Informática

CUADERNOS TÉCNICOS

dotNetManía



Programación en Silverlight 2.0

Marino Posadas

9

LUARNA

Este libro pertenece a:

Marino Posadas

-

Mposadas@mvps.org

Sinopsis: El propósito de este libro es suministrar al lector los conocimientos necesarios para empezar a construir aplicaciones con la versión 2.0 de Silverlight, abordando para ello todos los procesos fundamentales: la elección del entorno de trabajo,el análisis de la arquitectura y los modelos de desarrollo, la construcción de interfaces de usuario (desde **Visual Studio 2008** y también desde **Expression Blend 2.0 SP1**), el acceso a la información del sistema y a servicios Web para la lectura/escritura de datos (en sus diferentes opciones) y los mecanismos de instalación y puesta a punto de las aplicaciones finales.

Los **Cuadernos Técnicos de dotNetManía** son una serie de pequeños libros enfocados a temas concretos para programadores y arquitectos de software de la plataforma .NET. Cubren el hueco existente entre artículos muy específicos en una revista especializada como dotNetManía o los grandes libros sobre temas genéricos.



Marino Posadas es Redactor Jefe de **dotNetManía**. Trabaja como Consultor y Program Manager en **Alhambra-Eidos** y es Microsoft Most Valuable Professional en Visual C#, además de titulado MCSD, MCAD, MCT y MSFP. Antes de esta obra, ha colaborado con varios compañeros de trabajo en la elaboración de otros cuatro títulos, y esta es su tercera obra en solitario. Conferenciante en diversos eventos organizados por **Alhambra-Eidos**, **Microsoft** y varias universidades españolas y extranjeras, se interesó por .NET Framework desde las primeras versiones «alfa» y mantiene una Web de soporte de sus actividades en el sitio www.ElAveFenix.net

Luarna

Programación en Silverlight 2.0
© Marino Posadas
© De esta edición: 2009, Luarna Ediciones, S.L.
www.luarna.com

Madrid, marzo de 2009

ISBN: 978-84-92684-19-9
Versión: 1.0 (26-03-2009)

Queda prohibido, salvo excepción prevista en la ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (art. 270 y ss. Código Penal).

Edición en papel

Primera edición en papel: 2008
Programación en Silverlight 2.0
© Marino Posadas
© Netalia, S.L., 2008
ISBN: 978-84-934895-8-8

CUADERNOS TÉCNICOS
dotNetManía

Programación en Silverlight 2.0

Marino Posadas

nº 9



"*Creo que parte de mi amor a la vida se lo debo a mi amor a los libros*"
Adolfo Bioy Casares
A Milagros, por su paciencia y apoyo.

"*Los amigos son como la sangre, cuando se está herido acuden sin que se los llame*"
Anónimo
A Paco Morero y a Paco y Pilar, por su amistad.

"*Confiamos demasiado en los sistemas y muy poco en los hombres*"
Benjamin Disraeli
A Luismi, alguien de confianza.

"Fuera del perro, un libro es el mejor amigo del hombre; dentro del perro, probablemente esté demasiado oscuro para leer"
Groucho Marx

Agradecimientos

Varias personas han aportado, de una forma u otra, ideas, o me han apoyado en la medida de sus posibilidades. Vaya aquí mi agradecimiento a algunos de ellos, y los que —habiéndolo hecho— no estén en la lista, que lo achaquen a mi despiste habitual: "mea culpa".

Entre ellos, quiero recordar a mi grupo favorito en Microsoft Ibérica, "los chicos/as del DPE": **Alfonso Rodríguez** y **Beatriz Ordoñez** que confiaron en mí para este trabajo, **David Carmona**, que me sugirió ideas desde el inicio para la estructura del libro; **David Salgado**, a quien sigo en su blog, siempre al día, **Antonio Gómez**, que, pidiéndonos siempre renovación, nos mantiene alerta, y muchos otros, como **César de la Torre**, **Aurelio Porras**, **Isabel Gómez**... ¡sois geniales!

Al grupo de MVP españoles (especialmente los de desarrollo, con quienes tengo más contacto), con **Cristina González** a la cabeza. Muchas veces habéis apoyado mi trabajo, y quiero recordaros aquí: **Jorge Serrano**, **Octavio Hernández**, **Guillermo Som**, **Miguel Egea**, **Eladio Rincón**, **Salvador Ramos**, **Daniel Seara**, **José Manuel Alarcón**, **Miguel Jiménez**, **José Miguel Torres**, **Rodrigo Corral**, **Iván González**, **Unai Zorrilla**, **Alejandro Mezcua**, **Lluis Franco**… y no sigo que se me acaba el espacio… (y el tiempo).

Y a muchos otros compañeros de profesión de diversos sitios con los que me unen intereses comunes o ideas similares: **Adolfo Wiernik**, **Dino Esposito**, **Miguel Katrib**, **Brian Wheeler**, los chicos de **SecondNug**, **Yamil Hernández**, **Luis Fraile**, **Jesús López**, **José Luis Latorre**, y en general, a los compañeros de trabajo que me han apoyado, antiguos y actuales. Gracias a todos, perdón por las omisiones involuntarias, y espero que esto os sirva de introducción a Silverlight 2.0.

Índice

1. Introducción a Silverlight 2.0 16
A quién va dirigido este libro.....17
Requisitos previos 17
Buscando una definición 18
Aplicaciones RIA.....19
Otras plataformas RIA 21
Arquitectura de Silverlight.....23
Tecnología DeepZoom 26
Media Stream Source.....26
Comparativa de versiones 26
Silverlight y los lenguajes.....28
Lenguajes dinámicos 29
Sobre las herramientas de desarrollo para Silverlight.....29
Instalación offline o con escasos privilegios administrativos.....30
El papel de Expression Blend 32
Qué ofrece el modelo de objetos de Silverlight 33
El CoreCLR.....34
El modelo de seguridad del CoreCLR.....35

Las BCL (Base Class Libraries) en el CoreCLR.....37
Algunas características fundamentales.....37
Funcionamiento multiplataforma:
el Platform Adaptation Layer (PAL).....38
Platform Adaptation Layer (PAL).....39
El modelo desde el punto de vista
del desarrollador.....40
La creación del control Silverlight.....41
Ejemplo básico de programación
con Silverlight 1.0 usando JavaScript.....41
Proceso recomendado para la actualización
del código 1.0 a 2.0.....45
Conclusión.....45

2. El marco de desarrollo para Silverlight 2.0.....47
Visual Studio 2008 y los tipos
de proyecto Silverlight.....48
La aplicación Web de prueba.....50
Notas respecto a la ejecución en otros entornos.....51
Instanciación del objeto Silverlight en
los dos modelos (HTML / ASP.NET).....51
Manejo de distintas versiones del runtime.....53
El elemento <asp:Silverlight>.....54
La parte XAML de la aplicación Silverlight
predeterminada.....57
El flujo de procesos dentro de la aplicación
Silverlight.....59
Comprobación de la versión de Silverlight
instalada en el cliente.....62
El ciclo de vida de una aplicación Silverlight.....64

3. Expression Blend para desarrolladores 65
Desarrollo de una aplicación Silverlight básica con Expression Blend 2.0.....65
El IDE de Expression Blend 2.0.....68
Elementos principales del IDE.....69
Controles y geometrías.....69
Creación automática de elementos a partir de recursos gráficos.....70
Más sobre los Assets.....71
Categorías de controles.....72
Ejemplo Inicial.....73
Probando la solución inicial.....75
Trabajo con objetos de dibujo.....77
Cambio del punto de entrada de la aplicación 80
Trabajando con la Ventana de diseño 80
La superficie de diseño.....81
Brushes 83
ImageBrush 84
VideoBrush.....88
Otros Brushes.....90
Otras sub-ventanas de propiedades 92
Búsqueda de propiedades.....93
Un pequeño visor de fotografías.....94
Programando un procedimiento de evento.....96
Ajustando el control para efectos especiales.....98
Conclusión 100

4. XAML en Silverlight 2.0 101
El lenguaje XAML 102
Elementos estáticos: la interfaz de usuario básica ..104
Espacios de nombres en XAML 105

El inevitable “Hola Mundo” 107
Layout 108
Elementos contenedores 108
Propiedades adjuntas 110
Los otros contenedores: Canvas y StackPanel 112
Márgenes y alineación 118
Elementos de dibujo: Shapes y Geometries 119
Diferencias entre formas y geometrías 119
Formas (Shapes) 121
Los objetos Line, PolyLine y Polygon 125
Geometrías y el objeto Path 127
PathGeometry 131
Arcos 134
Curvas Bézier 135
Una nota sobre la conversión automática de formatos en ficheros XPS 138
Recortes mediante geometrías 139

5. Controles 141
Controles disponibles en Silverlight 2.0 142
Controles que muestran/admiten texto como función primaria 143
Peculiaridades del ListBox y los elementos colectivos 147
Controles para ayuda de la experiencia visual de uso directo 150
Una primera prueba con manejo de eventos 155
Controles multimedia 157
El resto de controles 158
ContentControl 159
Etiquetas flotantes (ToolTip y ToolTipService) 161

Los “Presenters” 163
InkPresenter 164
Thumb, ToggleButton y RepeatButton 164
Otros controles disponibles 166
Resumen 166

6. Plantillas, animaciones y Visual State Manager.....168
Dependency Properties y eventos 168
Propiedades de dependencia y propiedades del CLR 169
El sistema de eventos en Silverlight 2 170
Bubbling 171
Estilos y plantillas.....176
Uso de estilos para cambiar la apariencia de un conjunto de controles.....177
Asignación Late Binding de una plantilla.....182
ContentPresenter.....183
Edición de estilos y plantillas desde Expression Blend 185
Silverlight 2.0 Animation System 186
Animaciones 187
Animaciones en Silverlight 187
Transformaciones vs. animaciones.....188
Transformaciones.....188
Propiedades RotateTransform y ScaleTransform ..189
Creación de transformaciones mediante código 191
Matrices de transformación y TransformGroups...192
Animaciones 192
Triggers 193
El trigger más simple sólo en código XAML.....194

Respuesta a eventos mediante código 196
Creación de animaciones mediante código 197
Líneas de tiempo y KeyFrames.....199
Interpolación no-lineal 203
Visual State Manager 204
El modelo Parts-And-States 205
Partes 205
Estados y grupos de estado 206
Deconstruyendo a Button.....207
Utilización de Visual State Manager para la personalización de una interfaz de usuario 208
Creación de estados y transiciones propios con VSM 210
Construcción de un control personalizado.....219
Pasos típicos en la creación de un control de usuario 220
El control TextoDNI 221

7. El tratamiento de datos.....230
DataBinding en Silverlight 2 231
Sintaxis de DataBinding en XAML 232
DataContext e ItemSource 233
Ejemplo 233
La interfaz INotifyPropertyChanged.....234
Acceso a datos mediante servicios Web.....242
Acceso a datos mediante servicios WCF 242
Ejemplo 243
Acceso a datos mediante servicios Web SOAP (ASMX).....252
Utilización de servicios REst 255
Políticas de seguridad.....256

ADO.NET Data Services 260
ADS y los datos relacionales 261
Ejemplo de ADO.NET Data Services 261
DataBinding con Expression Blend 2.0 269
Vinculación a objetos de negocio 269
Enlazando a objetos de negocio 273

8. Arquitectura y distribución de aplicaciones 282
Arquitectura de aplicaciones Silverlight: recomendaciones 282
Marco tecnológico 283
Buenas prácticas en la construcción de aplicaciones Silverlight 2 284
Capas de presentación 285
Controles 285
Media y gráficos 286
Navegación 286
Gestión de estados 288
Validación de datos de entrada 288
Cachés 289
Accesibilidad 289
Comunicaciones entre capas 290
Seguridad 290
Acceso a datos 291
Excepciones y depuración 292
Distribución de las aplicaciones 292
Optimación de la experiencia inicial 293
La experiencia de instalación 293
Algunas técnicas recomendadas 295
Acceso a recursos de la aplicación 295

Tratamiento de recursos en tiempo de compilación 296
Manejo de fuentes adicionales 298
Configurando un servidor Web para el alojamiento de Silverlight.....299
Valores instalados de forma predeterminada en IIS 7.0 sobre Windows 302
Conclusión 302

Apéndice 1. Bibliografía e información on-line 304

1

Introducción a Silverlight 2.0

Suponemos que el lector ha oído hablar de Silverlight en el pasado. Al menos, de la primera versión (1.0), aparecida de forma oficial en septiembre de 2007. En el momento de escribir estas líneas acaba de aparecer la versión 2.0 final, y aunque iniciamos el trabajo con la primera beta, hemos ido revisando los cambios a medida que se producían, pudiendo, finalmente, contar con la RTW (*Release to Web*) con tiempo suficiente como para poder verificar todo de nuevo.

El propósito de este libro es suministrar al lector los conocimientos necesarios para empezar a construir aplicaciones con la versión 2.0 de Silverlight, abordando para ello todos los procesos fundamentales: la elección del entorno de trabajo, el análisis de la arquitectura y los modelos de desarrollo, la construcción de interfaces de usuario (desde **Visual Studio 2008** y también desde **Expression Blend 2.0 SP1**), el acceso a la información del sistema y a servicios Web para la lectura/escritura de datos (en sus diferentes

opciones) y los mecanismos de instalación y puesta a punto de las aplicaciones finales.

Para todo lo referente a la documentación *on-line*, así como los enlaces de descargas relacionados, Microsoft ha creado el sitio `http://silverlight.net`, donde se pueden encontrar las herramientas necesarias para la instalación del entorno de desarrollo: SDK, documentación adicional (en inglés, de momento) en forma de borradores de trabajo, ejemplos (vinculados a su versión correspondiente) y vídeos ilustrativos de su funcionamiento, dirigidos por algunos de los principales evangelistas y responsables del producto: **Jessy Liberty, Tim Heuer, Joe Stegman, Scott White**, etc. En la parte final de esta obra, se añaden un conjunto de referencias útiles, tanto desde el punto de vista bibliográfico, como de blogs recomendadas que mantienen información actualizada sobre esta tecnología.

A quién 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 (preferiblemente de la versión 2.0 o superior), que deseen evaluar las posibilidades de Silverlight y ver sus capacidades como plataforma de construcción de aplicaciones RIA. También pensamos que puede ser de interés para responsables de la toma de decisiones, que quieran valorar las posibilidades de adopción de Silverlight 2.0 en futuros proyectos.

Requisitos previos

Para la compresión del texto, se requiere el conocimiento —siquiera superficial— de los fundamentos de la progra-

mación con .NET Framework y su herramienta de desarrollo principal, Visual Studio, así como conocimientos generales de la programación para Internet, y tecnologías asociadas (incluyendo lenguajes de marcas, y concretamente, XML). Respecto a los lenguajes de programación, puede utilizarse cualquiera de los dos principales soportados por Visual Studio (C# y Visual Basic .NET), si bien nosotros utilizaremos C# en los ejemplos que ilustran esta obra, aunque existe más código XAML que C#.

Para una profundización más provechosa en el contenido, es aconsejable el conocimiento (al menos a nivel introductorio) de la tecnología de presentación introducida por .NET Framework 3.0: **Windows Presentation Foundation**, así como de los mecanismos implicados en el acceso datos mediante servicios Web (tradicionales o basados en **Windows Communication Foundation**).

Buscando una definición

Si buscamos en la Web una definición de la tecnología y seguimos lo indicado en la enciclopedia *on-line Wikipedia*, veremos que Silverlight *es un complemento para navegadores de Internet que agrega algunas funciones de Windows Presentation Foundation* —WPF desde ahora—, *como la reproducción de vídeos*, *gráficos vectoriales*, *animaciones y otros elementos*. Esta definición se refiere exclusivamente a la versión 1.0, y no contempla el profundo cambio que la versión 2.0 supone en la construcción de aplicaciones Web, por lo que preferimos enmarcar la tecnología en su justo contexto dentro de las aplicaciones RIA (*Rich Internet Applications*), y definirla como *un complemento multi-plataforma para diversos navegadores de Internet que incorpora las capacidades de*

Windows Presentation Foundation, y otras propias de .NET Framework, para permitir la construcción de aplicaciones RIA.

Aplicaciones RIA

Presentamos, por tanto, Silverlight como una tecnología para la construcción de una nueva generación de aplicaciones, denominada aplicaciones **RIA**, que pretende conjugar lo mejor del mundo Web con las ventajas de las aplicaciones de escritorio: una excelente experiencia de usuario y la distribución remota de la aplicación. La riqueza a la que hace mención el nombre es —sobre todo— una riqueza de recursos de interfaz de usuario: precisamente lo que más se echaba de menos en las aplicaciones Web. De hecho, en esta versión está disponible un importante subconjunto de la funcionalidad presentada por WPF, así como soporte de interoperabilidad, acceso a recursos, lenguajes dinámicos, multimedia, acceso a datos, etc.

En general, podemos enumerar las ventajas principales de las aplicaciones RIA como las siguientes[1]:

- Balance cliente/servidor.
- Eficiente comunicación asíncrona.
- Reducción del tráfico de red .
- No necesitan instalación (acceso vía Web) y las actualizaciones hacia nuevas versiones son automáticas.
- Están soportadas por las plataformas y navegadores más populares del mercado. En el caso de Silverlight, estas plataformas son Mac y Windows (y, a través del proyecto **Moonlight**, ligado a Mono

[1]. Citado de Wikipedia: `http://www.es.wikipedia.org/wiki/ Rich_Internet_Application.`

(`http://www. mono-project.com/Moonlight`), se está realizando la implementación para Linux). Los navegadores que lo soportan: Internet Explorer, Firefox *y* Safari.
- Además, el equipo de desarrollo de Opera trabaja con el de Silverlight, para que en una próxima versión, ya se encuentre disponible para este navegador y, respecto a Google Chrome, aunque las primeras pruebas daban fallos o funcionaban de forma más o menos aleatoria parece que ya la RTW tiene un alto nivel de compatibilidad con la versión para desarrolladores (Google Chrome Build 1251) y será totalmente compatible en próximas versiones de este navegador.
- Es menos probable la infección por virus, que utilizando, por ejemplo, programas ejecutables.
- Más capacidad de respuesta, ya que el usuario interactúa directamente con el *runtime*, sin necesidad de recargar la página.
- Ofrecen aplicaciones interactivas que no se pueden obtener utilizando sólo HTML, incluyendo arrastrar y pegar, cálculos en el lado del cliente sin la necesidad de enviar la información al servidor, etc.

Obviamente, las aplicaciones RIA también tienen sus inconvenientes, aunque se está trabajando en minimizar la mayoría de ellos:

- Ejecución en *SandBox* (depende de la configuración del cliente, aunque también puede considerarse una ventaja desde el punto de vista de la seguridad y las acciones permitidas en la máquina del cliente).

- Que esté deshabilitada la opción de "Scripting" en el navegador (esto afecta principalmente a la versión 1.0, basada en este lenguaje, aunque el tratamiento de errores predeterminado se gestiona mediante código JavaScript también en la versión 2.0).
- El tiempo de descarga de la aplicación (si bien es —normalmente— mínimo y comparable muchas veces al de la descarga de los gráficos de una página).
- Cierta pérdida de visibilidad en los motores de búsqueda (aunque se está trabajando muy positivamente en ese sentido).

Otras plataformas RIA

Silverlight no es la única plataforma que permite construir aplicaciones de esta clase, aunque sí una de las más ricas en posibilidades y fácil en su desarrollo. De hecho, existen varias propuestas de esta clase en el mercado:

- **Adobe Flash Player y Adobe Flex**
  - Instalado en más del 90% de los ordenadores mundiales.
  - Multiplataforma.
  - Lenguajes propietarios MXML y *ActionScript*.

- **Adobe AIR (Adobe Integrated Runtime)**
  - *Runtime* multi-plataforma.
  - No requiere conocimientos nuevos para un desarrollador Web.
  - No puede considerarse un sistema RIA completo.

- **Plataformas AJAX**
  - Basadas en JavaScript y el objeto `XmlHttpRequest`.
  - Los datos leídos pueden usar XML "formateado".
  - Multinavegador sin necesidad de complementos (*add-ins*).
  - Como otros RIA, no se adapta bien a la optimización de los motores de búsqueda.
  - Problemas con clientes con "Scripting" deshabilitado.

- **JavaFX**
  - Complemento de la familia de herramientas de Java.
  - Sirve para aplicaciones de escritorio, móviles y aparatos electrónicos (con ese soporte).
  - Toda la programación en Java. (necesita, lógicamente, la *JavaVM*).

- **Google Gears**
  - Acceso desconectado a servicios *on-line*.
  - Instala un motor de bases de datos basado en *SQLite* en el cliente, para "cachear" información de la aplicación y permitir su uso posterior.
  - Para cualquier operación, la actualización se difiere en caso de existir una conexión disponible.
  - No es un RIA, propiamente dicho, pero la promesa es que se acerque mucho, al solventar el problema de las conexiones.

Arquitectura de Silverlight

MSDN, en su página `http://msdn2.micro soft.com/en-us/library/bb404713.aspx`, presenta un esquema gráfico muy aclaratorio de los constituyentes principales de la arquitectura de Silverlight 2.0 comparada con la versión anterior.

Podemos distinguir en esta arquitectura tres elementos principales:

- Un núcleo de presentación (*Presentation Core*), que dispone de gestión de derechos digitales.

- Un subconjunto de .NET, que se denomina en el gráfico *.NET for Silverlight*, que, si bien no contiene toda la riqueza que se encuentra en .NET 3.5, asombra por la cantidad de características incluidas. Como puede verse en el gráfico, es la parte del complemento que más novedades presenta respecto a su antecesor (en acceso a datos, lenguajes dinámicos, controles WPF, soporte de WCF y muchas de las librerías básicas de .NET 3.5).

- El instalador, que se ocupa del proceso de implantación en el equipo cliente para usuarios iniciales y también de la actualización a nuevas versiones.

Como puede observarse, el número de elementos nuevos en la versión 2.0, y la riqueza de éstos, la convierten en una auténtica plataforma RIA, que además, se ve potenciada por la posibilidad de utilizar un IDE bien conocido, con todos sus recursos.

A tenor del gráfico, las diferencias entre ambas versiones son muy considerables, y hay autores que califican de

hecho la versión 1.0 como "tentativa", o primera aproximación, con poco más que algunas capacidades multimedia y de presentación. El otro inconveniente de la primera versión era el lenguaje de desarrollo (el de presentación era un pequeño subconjunto de XAML, muy inferior al actual). JavaScript no es —para la gran mayoría— el más idóneo, aunque Visual Studio 2008 haya hecho un esfuerzo en ese sentido, y por primera vez contemos con características avanzadas para la codificación con JavaScript como Intellisense y depuración.

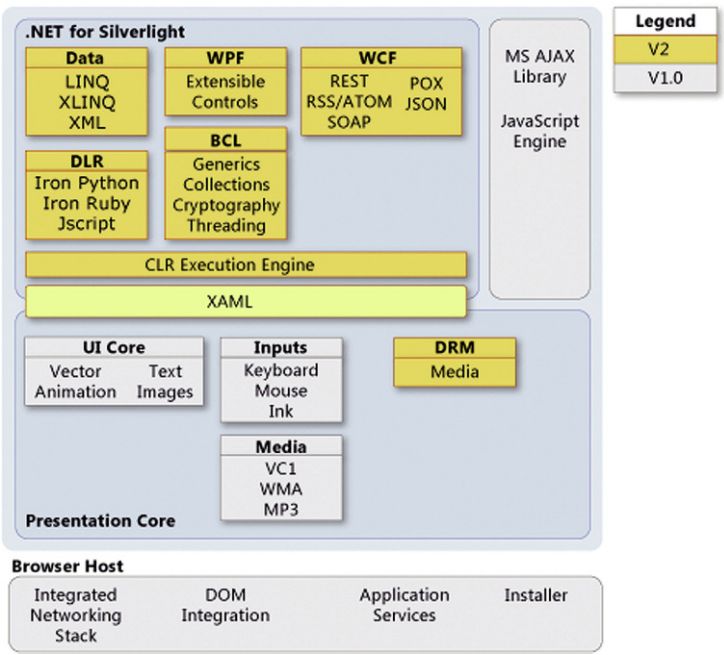


Figura 1-1. Arquitectura de Silverlight indicando las novedades de la versión 2.0.

La tabla siguiente es una explicación más detallada de los componentes principales de presentación descritos en el gráfico anterior:

Características	Descripción
Input	Maneja las entradas desde dispositivos hardware como el teclado, ratón, dispositivos de dibujo y otros sistemas de introducción de datos.
UI rendering	Interpreta gráficamente vectores, gráficos de bitmap, animaciones y texto.
Media	Controla la ejecución y gestión de varios tipos de ficheros de vídeo y audio, como los formatos .wmp y mp3.
Controls	Soporta controles extensibles que se pueden personalizar mediante estilos y plantillas.
Layout	Habilita el posicionamiento dinámico de elementos de la interfaz de usuario.
Data Binding	Habilita el enlace de objetos de datos a elementos de la IU.
DRM	Habilita la gestión de derechos digitales de los recursos multimedia.
XAML	Suministra un analizador de código para XAML.

Tabla 1. Capacidades principales de la versión 2.0 de Silverlight.

Tecnología DeepZoom

Otra novedad es la presencia 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 mega-píxeles hasta un giga-píxel, pero el usuario no tiene que esperar a la descarga completa, gracias a la estructura subyacente que permite la descarga concreta de las zonas necesarias. Internamente, utiliza el formato de fichero XML.

Media Stream Source

Se denomina así a una potente API que permite la descarga dinámica adaptable de contenidos multimedia. Así, la aplicación que reproduce el contenido multimedia selecciona la velocidad de descarga basándose en el ancho de banda del cliente y su potencia de CPU.

Comparativa de versiones

Sobre las diferencias concretas entre ambas versiones, baste ver la relación que incluimos en la tabla 2, para comparar entre ambas. Las mejoras de esta versión no son sólo en cantidad (que también) sino en calidad. A eso nos referimos la calificar a Silverlight 2.0 como otro producto; en total hay 16 características fundamentales nuevas, respecto a la anterior.

Características	Silverlight 1.0	Silverlight 2.0
Animación/Gráficos 2D	*	*
Soporte AJAX	*	*
Multi-navegador (Firefox, IE, Safari, Opera, Google Chrome)	*	*
Multi-plataforma: Windows, Mac (Linux en beta)	*	*
Lenguajes de .NET Framework (Visual Basic, Visual C#, IronRuby, Iron Python)	-	*
Integración HTML DOM	*	*
HTTP *Networking*	*	*
Almacenamiento Aislado	-	*
Soporte JavaScript	*	*
JSON, REST, SOAP/WS-*, POX, y RSS Web Services (y soporte para Sockets)	-	*
Acceso a redes entre dominios	-	*
LINQ to Objects	-	*
Soporte de posicionamiento en Canvas	*	*
Soporte de posicionamiento en *StackPanel*, *Grid* y *Panel*	-	*
Marco de controles administrados	-	*
Completo conjunto de controles (*TextBox*, *RadioButton*, *Slider*, *Calendar*, *DatePicker*, *DataGrid*, *ListBox*, y otros)	-	*
Integración *Deep Zoom*	-	*
Soporte de HTML administrado	-	*
Manejo de excepciones administrado	-	*
Multimedia: Protección de contenido	-	*
Multimedia: Video 720P de Alta Definición (HD)	*	*
Multimedia: Soporte Audio/Video (VC-1[2], H.264 WMV, WMA, MP3, AAC)	*	*
Multimedia: Soporte de imágenes (JPG, PNG)	*	*
Multimedia: *Markers*	*	*
Framework básico enriquecido (Genéricos, colecciones)	-	*

2. Tanto HD DVD como Blu-Ray han adoptado VC-1 como estándar de vídeo obligatorio, lo que significa que sus dispositivos de reproducción de vídeo deben ser capaces de decodificar y reproducir contenido de vídeo comprimido con formato VC-1. Windows Vista soporta reproducción de HD DVD, incluyendo el decodificador VC-1 y componentes relacionados que son necesarios para la reproducción de películasHD DVD codificadas con VC-1.2, y hay planes, igualmente, para un pronto soporte de Blu-Ray.

Características	Silverlight 1.0	Silverlight 2.0
Mejoras de seguridad	-	*
Controles Silverlight ASP.NET, como asp:media, asp:xaml, etc.	*	*
Verificación de seguridad de tipos	-	*
Soporte de Windows Media Server	*	*
Analizador (Parser) de XAML (basado en WPF)	*	*
Objetos *XMLReader* y *XMLWriter*	-	*

Tabla 2: Comparativa funcional entre las versiones 1.0 y 2.0 de Silverlight

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 fundamentalmente compuesto de una DLL donde radica nuestra aplicación compilada (más todos los requisitos que pudiera necesitar para su ejecución), y un fichero `.manifest` que define aspectos 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 2008 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, o VB.NET/C# en la versión 2.0). Veremos con más en detalle el modelo de ejecución en el capítulo 2.

Lenguajes dinámicos

Una de las novedades interesantes de esta versión es el soporte de lenguajes dinámicos: *JScript*, *IronPython 2* y *IronRuby*. Para ello, Silverlight 2.0 incorpora 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`). De hecho parece que la próxima versión de Visual Basic (VB 10 o VBx) ofrecerá soporte para el DLR[3].

El lector podrá encontrar abundantes ejemplos de utilización de estos dos lenguajes en Silverlight en el sitio oficial antes indicado.

Sobre las herramientas de desarrollo para Silverlight

En el momento de escribir esto, estamos utilizando Visual Studio 2008 SP1, Expression Blend 2.0 Service Pack 1 y .NET Framework 3.5 SP1, además del propio SDK de Silverlight. Todo el proceso de instalación (si no se da ninguna

[3]. Para más información, ver:
`http://en.wikipedia.org/wiki/Visual_Basic_.NET#Visual_Basic_.27VBx.27_.28VB_10.0.29`

incompatibilidad en las condiciones de instalación), es automático a partir del descargable disponible en el sitio Web citado al principio. La buena noticia es que existen herramientas de desarrollo gratuitas: podemos usar Eclipse (mediante un *plug-in* suministrado por Soyatec o Microsoft Visual Web Developer Express SP1).

Respecto al sistema operativo, se requiere Windows 2003 SP4 o superior (XP, Vista). En las demos, utilizaremos Visual Studio 2008 Team Suite Service Pack 1, debido a las ventajas que ofrece para el desarrollo con Silverlight respecto a sus antecesores, si bien pueden utilizarse otras ediciones de Visual Studio 2008: con solo instalar el SDK del producto, nos aparecerán dos plantillas de proyecto Silverlight (*Proyectos de Aplicación Silverlight* y *Librería de Controles Silverlight*) que quedan disponibles como un tipo de proyecto más en los dos lenguajes predeterminados.

El entorno es el usual de Visual Studio: tecnología Intellisense, tanto en la edición del código XAML, como en JavaScript (caso de usar este lenguaje), junto a las ya típicas capacidades de depuración para todos los lenguajes disponibles, soporte síncrono de diseño según se van editando elementos de la interfaz de usuario (y está anunciado un nuevo diseñador visual al estilo del que ya contamos para ASP.NET o Windows Forms), ayuda local y en línea, y los típicos *gadgets* que son el día a día de Visual Studio.

Instalación off-line o con escasos privilegios administrativos

En entornos de desarrollo donde no está permitido el acceso a Internet, o si se necesita hacer una instalación donde no se tiene acceso, existen dos formas de abordar el problema,

pero previamente, necesitamos las herramientas como dos ficheros separados:

1) Primero, se necesita el instalador "Silverlight Tools for Visual Studio 2008 SP1", disponible en la dirección:
   `http://go.microsoft.com/fwlink/?LinkId=129043.`
2) Descargamos el Silverlight 2.0 Developer Runtime desde la dirección
   `http://go.microsoft.com/fwlink/?linkid=129011.`

Y las dos opciones de instalación son:

a) Extraer y pegar.
   a. Abrir una ventana de comandos (`cmd.exe`) y navegar a donde esté el fichero `Siverlight_tools.exe.`
   b. Ejecutar el comando `silver light_tools.exe /x.`
   c. Una vez descomprimidos los ficheros, volcar en el mismo directorio el Silverlight 2.0 Developer Runtime y ejecutar `SPInstaller.exe.`

b) Colocar el Silverlight 2.0 Developer Runtime en el directorio `Temp` (no requiere descompresión de ficheros).
   a. Navegar al directorio `Temp` (o abrir una ventana de comandos y teclear la secuencia: `CD %Temp%.`
   b. Crear desde ahí un directorio para la instalación.
   c. Copiar `Silverlight.2.0_Developer.exe` a ese directorio y ejecutar el instalador `Silverlight_tools.exe.`

Con uno de estos procesos toda la maquinaria necesaria para el desarrollo, debiera de quedar lista para funcionar.

El papel de Expression Blend

No obstante, en lo referente al diseño (la parte visual) y la generación del código XAML, tenemos que insistir en las bondades de **Microsoft Expression Blend 2.0 SP 1** (hay descargable una versión de evaluación). Su uso no es requisito imprescindible para la construcción de aplicaciones Silverlight, pero —tenemos que insistir— sí que es **muy** recomendable. Y no es una mera cuestión estética.

A lo largo de esta obra, utilizaremos esta herramienta en repetidas ocasiones para simplificar la generación del código vinculado a los aspectos visuales de las aplicaciones. A partir del capítulo 3 y siguientes, veremos cómo utilizarlo en distintos escenarios tanto en el desarrollo de interfaces de usuario Silverlight, como en aspectos que tocan directamente la lógica de negocio: el enlace a datos, el acceso a objetos de negocio o la lectura de información basada en servicios.

Con esta herramienta podemos llevar hasta el último extremo el nuevo paradigma de desarrollo que propone Microsoft: la separación entre lo visual (fundamentalmente, la interfaz de usuario) y lo funcional. Trabajar conjuntamente con Visual Studio 2008, realizando el diseño visual con Expression Blend y la parte programática con nuestro IDE de siempre, de forma simultánea, configura el mejor entorno posible para el desarrollador de estas aplicaciones. Cuando lleguemos al análisis del nuevo gestor de estados visuales (*Visual State Manager*) comprenderá claramente el lector que, aunque Expression Blend no sea

imprescindible, la complejidad de las interfaces más sofisticadas (utilizando sistemas de animaciones y transformaciones) requiere de la facilidad de esta herramienta para editar esas interfaces complejas y el código XAML que se genera.

NOTA. En caso de que el usuario que lee estas líneas quisiera acceder al SDK e instalarlo antes de la aparición de la versión definitiva del producto, debe tenerse en cuenta que, para minimizar los problemas de instalación que pudieran surgir existen secuencias recomendadas de desinstalación de versiones anteriores y se invita a seguir la propia guía de instalación disponible en el sitio oficial antes citado.

Qué ofrece el modelo de objetos de Silverlight

El modelo de objetos de Silverlight o SOM (*Silverlight Object Model*)[4] define un conjunto de objetos que pueden ser utilizados en la creación de aplicaciones Silverlight como si se tratara de cualquier otra librería estándar.

Hasta ahora, la programación Web seguía patrones totalmente distintos a la programación tradicional: CSS, HTML/DOM, Java Script, etc. Si bien ASP.NET supone un gran avance en ese sentido, dependíamos del conocimiento de esas otras tecnologías para la puesta en marcha de sitios Web modernos. Con Silverlight 2, podemos aprovechar todo nuestro conocimiento de Windows Presentation Foundation, C#, Visual Basic .NET, XAML, etc., pudiendo construir sitios superiores en funcionalidad, presentación, capacidades y facilidad de programación.

4. No confundir con el popular webmaster Guillermo Som "el Guille"...).

Estas ventajas se basan fundamentalmente en el núcleo del *runtime* de Silverlight, también llamado *CoreCLR*. La forma en que ha sido construido y los problemas que se han resuelto en el proceso, ilustran muy bien el "carácter" de este *runtime* único y merecen un comentario más extenso.

El CoreCLR

El *CoreCLR* es un motor de ejecución de muy poco peso en términos de volumen de descarga (6 Mb), con una funcionalidad muy parecida a su versión superior. Las dos DLL principales de .NET (`mscorwks.dll` *y* `mscorelib.dll`), ya miden cada una casi lo mismo que todo el *CoreCLR*. La labor de reducción y simplificación de código ha sido muy notable, siempre con la mirada puesta en garantizar la máxima compatibilidad.

Precisamente por esta razón, el motor de ejecución y la máquina virtual son iguales. Eso incluye el sistema de tipos, el *Garbage Collector*, el compilador JIT, el *pool* de hilos de ejecución (*thread pool*), y otras partes fundamentales del motor de ejecución.

Como contrapartida, y dada la distinta naturaleza de las aplicaciones Web (más simples y de corta ejecución), el compilador JIT se ha enfocado principalmente en minimizar los tiempos de carga, en lugar de realizar otras optimizaciones más complejas. Y se ha actuado de la misma forma respecto a otros modelos de soporte y optimización.

Pero el código MSIL y los metadatos usados en las aplicaciones Silverlight son los mismos que en las aplicaciones de escritorio. Además, el hecho de que Silverlight no pretenda reemplazar al CLR principal, ha supuesto un gran

cambio en el motor: el *CoreCLR* puede ejecutarse paralelamente (*side-by-side*) al CLR estándar[5].

NOTA. Antes, era imposible ejecutar dos versiones del CLR desde el mismo proceso. Hay varias razones para esto, pero baste pensar que cada instancia del CLR asume que es la única capaz de acceder a sus datos estáticos. Si una variable existiera en dos versiones del CLR (pongamos 2.0 y 3.5), y ambas versiones fueran cargadas por el mismo proceso al mismo tiempo, ninguna de ellas podría cambiar datos en esa variable sin afectar al estado de la otra.

Por otra parte, hay poderosas razones para que ambas versiones del *runtime* (el estándar y el *CoreCLR*) puedan ejecutarse desde el mismo proceso: por ejemplo, sería imposible escribir una aplicación WPF que alojase un control *WebBrowser*, si éste navegase hacia una página que contuviera un control Silverlight. De esa forma, y para otros escenarios, se garantiza una compatibilidad completa.

El modelo de seguridad del CoreCLR

En .NET tradicional, los mecanismos de seguridad se basan principalmente en el modelo propuesto por CAS (*Code Access Security*), y las políticas asociadas con él. Esto habilita la creación de *sandboxes* (áreas protegidas de ejecución) donde se establecen claramente los permisos de usuarios y procesos. En Silverlight, solo se necesita una *sandbox*, equivalente a la de Internet Explorer cuando ejecuta scripts en una página Web. Debido a esto, se han podido eliminar las políticas de CAS en este modelo, y basarlo en el concepto de Transparencia de Seguridad (*Security Transparency*), que se introdujo en .NET 2.0.

El modelo divide el código en 3 clases:

- Transparente (*Transparent*),
- Crítico con seguridad (*SafeCritical*), y
- Crítico (*CriticalCode*).

Podemos asumir que el primero es el de más bajo nivel de permisos, no puede elevar privilegios, ni acceder a recursos sensibles del sistema o la plataforma. Todo el código de usuario estándar es de esta clase en Silverlight 2.

En el otro extremo, el código Crítico es el de más alto nivel de permisos y puede interactuar con el sistema a través de los mecanismos de invocación de plataforma (*Platform Invoke*), o incluso contener código no verificable. Pero, en Silverlight 2.0, **todo el código crítico debe ser parte de la plataforma**.

Por tanto, el código *SafeCritical* actúa como intermediario entre ambos. El símil que podemos hacer es sencillo: el código transparente sería como el código estándar de una aplicación Windows, el código Crítico serían las API de Windows, y el Crítico con Seguridad como la API entre el código de usuario y las API de Windows.

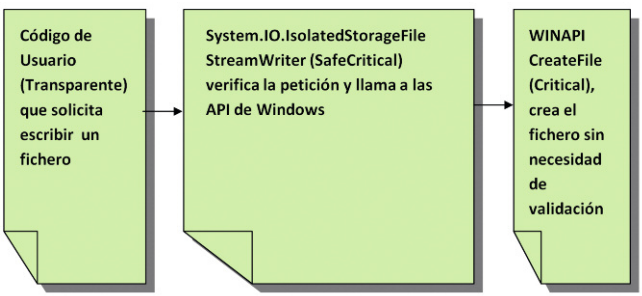


Figura 1-2. El proceso de llamadas a servicios críticos en Silverlight 2.0.

De esta forma, el código Transparente, solo puede llamar a otro código Transparente o a código Crítico con Seguridad. Y el código con Seguridad, solo puede llamar al código Crítico por encargo del código de Transparente (de usuario). Será labor del código Crítico con Seguridad, el mantener las salidas y entradas de forma controlada para mantener la seguridad del sistema.

Las BCL (Base Class Libraries) en el CoreCLR

Debido al enfoque de ejecución antes citado, muchas librerías de .NET no son necesarias aquí: por ejemplo, `System.Console`. Por otro lado, el **.NET Compact Framework** ha servido de inspiración en muchos aspectos, así como la experiencia de lecciones aprendidas en el proceso de minimización de un *runtime*, al tiempo que se mantuvo la compatibilidad con esa plataforma. Igualmente, se ha procurado evitar la duplicidad de elementos, y se ha modificado cierta funcionalidad para hacerla más sencilla, como los mecanismos de globalización.

Algunas características fundamentales

Dentro del *CoreCLR* hay algunas características muy importantes, como el soporte de tipos genéricos (veremos varios ejemplos a lo largo de este texto donde los utilizamos), la presencia de un solo hilo de ejecución para la interfaz de usuario y el hecho de que haya un solo objeto `Dispatcher` que maneje la cola de elementos en proceso para la IU. Esto es, utilizando el `Dispatcher`, podemos actualizar la interfaz de usuario desde un hilo de ejecución que no sea de interfaz de usuario (por ejemplo que ejecute otra acción en segundo plano).

Aunque por compatibilidad se soportan API de bajo nivel para el tratamiento de hilos de ejecución, tales como `System.Threading.ThreadPool.QueueUserWorkItem` y `System.Threading.Monitor.Enter`, se recomienda para esas tareas la utilización de `System.ComponentModel.BackgroundWorker`, ya que encapsula la actualización de la interfaz de usuario cuando concluye su tarea (dos trabajos en uno).

Otro aspecto importante de las BCL es la utilización del Almacenamiento Aislado (*Isolated Storage*). Es la forma adecuada para que una aplicación que se ejecuta en *sandbox* tenga acceso al sistema de ficheros y, aunque estaba presente desde la primera versión del Framework, aquí se potencia su utilización por razones de seguridad. Al igual que sucede con las *cookies* en aplicaciones Web, esto permite mantener estado entre distintas invocaciones en casos necesarios, y, aunque no está pensado para almacenamiento de información crítica como contraseñas y demás, su ubicación es segura, y su acceso se limita a la aplicación que posee dicho almacenamiento.

Funcionamiento multiplataforma: la Platform Adaptation Layer (PAL)

Hemos citado al comienzo la próxima disponibilidad del proyecto Moonlight, dirigido por **Miguel de Icaza**, para hacer que Silverlight esté disponible para Linux, lo que se está logrando a través de acuerdos firmados con Novell.

De la misma forma, Microsoft está trabajando en una versión de Silverlight para el sistema Symbian OS, de amplia aceptación en la industria de la telefonía móvil, así como para Windows Mobile. La versión móvil de Silverlight funcionará sobre el Compact .NET Framework,

mucho más ligera que el *CoreCLR*. Pero la versión para Mac OS X, será exactamente la misma que la del *CoreCLR*. ¿Cómo se ha conseguido esto?

Platform Adaptation Layer (PAL)

Esto se ha hecho posible gracias a la llamada Capa de Adaptación de Plataforma (PAL), que es una API escrita justamente para funcionar en distintas plataformas, y que sigue en evolución en los centros de investigación de Microsoft. Suministra abstracciones para el sistema de ficheros, gestión de errores, semántica de hilos de ejecución, servicios de redes, etc. Estas funciones comparten sus nombres con las de las API de Windows, pero difieren en su implementación.

Algunas simplemente pasan parámetros a funciones del sistema operativo (por ejemplo a OS X), mientras otras tienen que adaptarse para emular las mismas signaturas (definiciones formales de las funciones). En otros casos, existe funcionalidad en las API de Windows que no está presente en MacIntosh, por lo que las funciones deben ser implementadas en su totalidad. Hay que notar que buena parte de esta implementación se inspira en lo creado para *Rotor* (la API abierta de .NET, de nombre técnico SSCLI (*Shared Source Common Language Infrastructure*)6, que funciona en varias versiones de sistemas operativos tipo UNIX, así como en Windows.

Como consecuencia de sus objetivos más limitados, PAL solo soporta el subconjunto funcional de Win32 necesario para el funcionamiento de Silverlight. No hay necesidad de soporte de registro, GDI+, o COM/COM+, con lo que se mantiene pequeño y rápido. Incluso algunas de

las decisiones que se han tomado en la construcción del CLR podrán implementarse en la versión "superior", más adelante, como la ejecución en paralelo.

El modelo desde el punto de vista del desarrollador

Desde nuestro punto de vista, Silverlight, se expone como un control, que se crea como contenedor de un fichero XAML, cuyo nivel de soporte (respecto a WPF) dependerá de la versión. Por suerte, una buena parte de las capacidades de XAML para la definición de interfaces de usuario está presente en la versión 2.0 de Silverlight, con alguna excepción notable como la capacidad de crear interfaces 3D, aunque esto puede simularse para determinados procesos y ya existen algunos ejemplos al respecto en Internet.

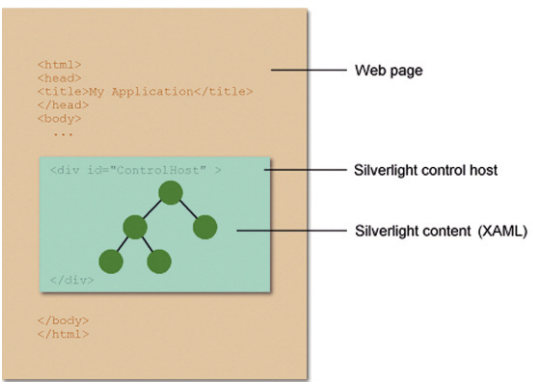


Figura 1-3. Modelo del esquema de objetos en una página que aloja un control Silverlight.

También es necesario tener en cuenta los formatos soportados para los elementos gráficos y multimedia que ahora

toleran una gama más amplia. Por tanto, el escenario de desarrollo, constará de 3 tipos de elementos, y la relación es de inclusión: *Página HTML -> Control "host" de Silverlight -> Contenido XAML dentro del con*trol, comportándose de una forma similar a como lo hacen los "Data Islands" de XML, solo que produciendo la salida interpretada visualmente (*rendered*) que se espera de XAML (la figura 1.3 muestra un gráfico de esta situación).

De forma que una página que contenga un complemento Silverlight, es una página Web clásica (HTML), con un elemento contenedor que sirve de *host* para el control Silverlight, y que establece en su definición la versión del *runtime* que requiere para su funcionamiento.

La creación del control Silverlight

El proceso de creación varía de una versión a otra: en la versión 1.0, tiene lugar llamando al *script* `Siverlight.js` mediante el método `createSilverlight`, donde se hace referencia al contenedor HTML del control (típicamente, una etiqueta `<DIV>`), así como a las dimensiones que se desean para éste. También es usual que se definan manejadores de eventos básicos de procesos, incluyendo el tratamiento de errores, las acciones en caso de que la versión requerida no esté presente y algunas más.

Concluimos esta introducción con un ejemplo simple de programación en la versión 1.0 de Silverlight.

Ejemplo básico de programación con Silverlight 1.0 usando JavaScript

Veamos cómo podemos implantar el modelo descrito para desarrollar una página que incluya un control Silverlight

1.0. Nos basta con un fichero HTML (o ASPX) que haga de *host* del control, y que deberá incluir una etiqueta contenedora responsable de albergarlo. Por otro lado, deberemos de incluir referencias a los ficheros de JavaScript que aporta el SDK; especialmente el fichero **`Silverlight.js`** que contiene la función responsable de la instanciación del control Silverlight, y recibe como argumentos los parámetros de configuración. Así pues, uno de los modelos típicos de página que tendríamos como punto de inicio sería el que muestra el listado 1-1.

También podríamos incrustar el código de creación del control en la propia página (en lugar de usar el fichero separado). El aspecto del código JavaScript para la instanciación del control puede verlo en el listado 1-2.

`Diseño.xaml` contendrá toda la interfaz de usuario a interpretar por el control. Finalmente, tendremos que indicar cuál es el fichero XAML a manejar por el complemento Silverlight que hemos creado. Para una prueba sencilla nos bastaría con un círculo con relleno, como se indica en el listado 1-3.

Listado 1-1. Documento HTML contenedor del control Silverlight

```
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
   <title>Demo de Silverlight con JavaScript</title>
   <script type="text/javascript" src="Silverlight.js"></script>
 </head>
```

```
 <body>

    <!-- Ubicación del control Silverlight -->
    <div id="SilverlightControlHost">
    </div>
    <script type="text/javascript">

      // Función que crea el control Silverlight en el navegador
      crearControlSilverlight();

    </script>
  </body>
</html>
```

Listado 1-2. Código fuente de la función que instancia el control Silverlight.

```
function crearControlSilverlight()
{
// Cacheamos el control contenedor, para pasarlo a la función de // // instanciación

var Contenedor = document.getElementById("SilverlightControlHost");
Silverlight.createObjectEx({
    source: "Diseño.xaml",      // Fichero fuente XA ML
    parentElement: Contenedor,  // Etiqueta DIV que aloja el control
    id: "SilverlightControl",  // ID único para el control
    properties: {               // Propiedades iniciales del control (se
       width: "100%", // incluye la versión requerida)
       height: "100%",
       version: "1.0"
    },
```

```
    events: { // Manejador del evento onLoad
      onLoad: Sys.Silverlight.createDelegate(Diseño, Diseño.handleLoad)
    }
  });
}
```

Listado 1-3. Código XAML sencillo, que dibuja una elipse con relleno en la superficie del control

```
<Canvas Width="300" Height="300"
       xmlns="http://schemas.microsoft.com/client/2007"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Ellipse Height="200" Width="200"Canvas.Left="30" Canvas.Top="30"
       Stroke="Black" StrokeThickness="10" Fill="LightBlue" />
</Canvas>
```

El resultado, muy simple en este caso, sería una imagen como la que vemos en la figura 1-4 (se asume que la figura dibujada es la inscrita en un cuadrado de 200 píxelesde ancho por otros 200 de alto, dando origen a un círculo y no a una elipse, al tratarse de un contenedor cuadrado):

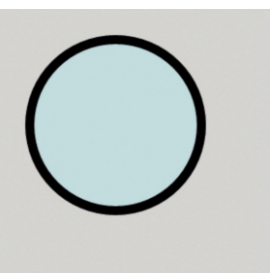


Figura 1-3. Imagen resultante de la salida en el navegador IE8 del código anterior.

Proceso recomendado para la actualización del código 1.0 a 2.0

En el caso de actualizar código de versiones anteriores, el lector debe de tener en cuenta que —en la versión 2.0— no es necesaria ya la presencia de ninguno de esos ficheros JavaScript. Como hay algunas modificaciones importantes, y lo recomendable es crear un nuevo proyecto desde Visual Studio 2008 y copiar a este proyecto los ficheros fuente del proyecto 1.0. Al pasar a depuración, aparecerán las incompatibilidades que pueden resolverse haciendo referencia a las librerías adecuadas, en la mayor parte de los casos, o cambiando algunos atributos que puede que no estén presentes en los elementos de esta versión. El conjunto de incompatibilidades posibles aparece listado en el manual oficial del SDK bajo el apartado "Breaking Changes in Silverlight 2". Además, el sitio Web oficial mantiene siempre la información actualizada incluyendo las listas de cambios entre versiones.

Si el usuario conoce Silverlight 2 Beta 1, podrá comprobar igualmente que el número de cambios en la versión final respecto a aquella es considerable, y puede que parte de lo hecho tenga que ser retocado, o la funcionalidad haya sido asumida por otros elementos, o eliminada por aproximaciones distintas al mismo problema.

Conclusión

Vamos a continuar a partir de este punto con el estudio de lo que ofrece la versión 2.0 exclusivamente. El esfuerzo de divulgación que está haciendo Microsoft a este respecto y

la repercusión social en la blogosfera, nos indica que es un camino a seguir, y —sin entrar en disquisiciones sobre diversas tecnologías competidoras— para los desarrolladores de .NET, el poder abordar este tipo de desarrollos con todas las herramientas y bagaje tecnológico al que estamos acostumbrados, es un motivo más para perderle el miedo.

2

El marco de desarrollo para Silverlight 2.0

Veíamos al final del capítulo anterior que la programación con la versión 1.0 se basa fundamentalmente en un fichero HTML o ASPX en el cual —mediante código Java Script— se instancia un control Silverlight, pasándole en la llamada los valores de configuración inicial. Una vez instanciado el control, éste aloja un fichero XAML que es interpretado visualmente según los patrones establecidos por el subconjunto de Windows Presentation Foundation presente en el *runtime* de Silverlight, disponiendo así de una riqueza visual de la que la carecíamos anteriormente.

También apuntábamos que el desarrollo con la versión 2.0 del producto iba a ser abordado utilizando Visual Studio 2008 Service Pack 1 y Expression Blend 2.0 Service Pack 1. Vamos a comenzar por analizar la estructura de una aplicación Silverlight tal y como la construye Visual Studio 2008 Service Pack 1, trabajando con la versión .NET Framework 3.5 Service Pack 1 (ambos en sus versiones finales).

Visual Studio 2008 y los tipos de proyecto Silverlight

Empezamos creando un proyecto Silverlight desde Visual Studio 2008, que genera toda la estructura de ficheros y directorios necesaria (en el capítulo siguiente, veremos el mismo proceso desde Expression Blend 2.0).

Una vez seleccionado el lenguaje que nos interesa, vemos que existen dos tipos de plantillas disponibles: "Aplicación Silverlight" y "Librería de Clases Silverlight". Esta última, suele usarse como parte de soluciones más complejas o para generar objetos de negocio separados que pueden ser reutilizados por otras aplicaciones y también para controles con una lógica especial que queremos que sean reutilizables.

La primera opción que debe decidir el usuario en este escenario, es la de si quiere hacer las pruebas mediante una página predeterminada que se crea dinámicamente, o prefiere que el IDE construya un sitio Web completo, con todo lo necesario para proceder con la depuración. Nosotros vamos a optar por la opción de la creación del sitio Web completo, para analizar la estructura de ficheros que genera el IDE. La caja de diálogo de selección que se le presentará al usuario es la de la figura 2-1.

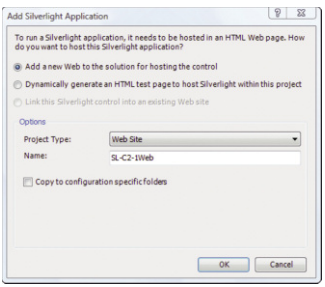


Figura 2-1. Ventana de selección de modelo de aplicación Silverlight.

Donde la opción "Copy to configuration specific folders" ofrece la posibilidad de copiar el fichero distribuible final (de extensión `.xap`), a una subcarpeta de configuración concreta (como *Debug* o *Release*), o bien a la carpeta de destino predeterminada (`ClientBin`).

Como resultado de este proceso inicial, contaremos con una solución que incluye dos aplicaciones: un sitio Web de prueba que utilizará `localhost` como servidor por defecto[1] y la aplicación Silverlight 2.0. Se genera un sitio Web en el sistema de archivos del equipo local que tendrá igual nombre que el de la aplicación Silverlight más el sufijo `_Web`, y se establecen las referencias necesarias. En cualquiera de los dos casos, el modelo de las plantillas de Silverlight implica la creación de un sitio Web para la depuración del componente; el hecho de que se prefiera trabajar con una simple página de pruebas, al estilo de la versión 1.0, no afecta a las capacidades de depuración desde Visual Studio.

Podemos ver en la figura 2-2 el resultado completo de todo este proceso.

La aplicación de inicio predeterminada será la del sitio Web creado para pruebas, donde encontramos 3 páginas Web: dos con el nombre de la aplicación más el sufijo `TestPage`, y extensiones `.aspx` y `.html`, más la página `Default.aspx` y un conjunto de referencias a librerías, incluyendo la de la aplicación Silverlight que está construyendo.

Además, es posible añadir aplicaciones Silverlight a sitios Web existentes, tanto si son aplicaciones ya hechas, como si queremos añadir una nueva para complementar el sitio.

---

1. Y usará aleatoriamente un puerto no estándar.

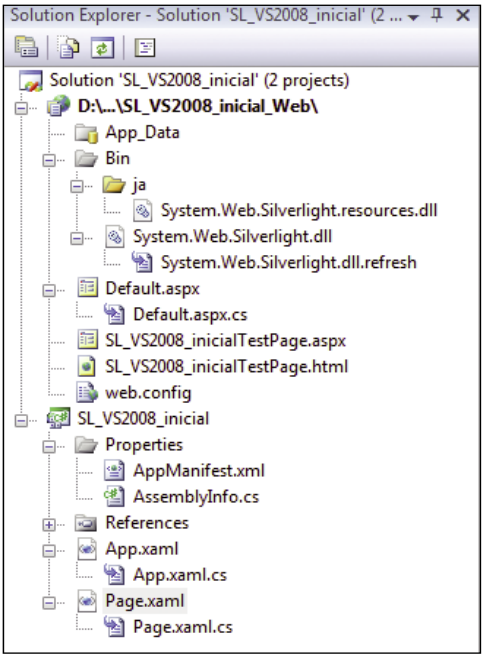


Figura 2-2. Estructura de una solución Silverlight 2.0 desdeVisual Studio 2008.

La aplicación Web de prueba

Establecidas las posibles variantes de trabajo, vamos a ver, siquiera someramente, la funcionalidad del sitio Web generado por defecto.

La página **`Default.aspx`** está vacía, solo con la estructura de los elementos básicos de una página Web, y los nombres de los dos ficheros generados para las pruebas llevarán el prefijo de la aplicación y las extensiones **`.aspx`** y **`.html`** (en este ejemplo, **`SL-C2-1TestPage.aspx`** y **`SL-C2-1TestPage.html`**).

Cualquiera de los dos puede servir para lanzar la aplicación y hacer funcionar el complemento Silverlight, solo que en el primer caso se hace uso de los recursos propios de ASP.NET y en el segundo solamente elementos HTML y JavaScript. Podemos hacer la prueba cambiando la página de inicio y viendo que, en ambos casos se ejecuta sin problemas. Además, en la versión final se añadió un fichero JavaScript (`Silverlight.js`), que no debemos confundir con el que se usaba en la primera versión para instanciar el complemento. Se trata de un fichero que incluye una firma digital y se encarga del control de la situación cuando el complemento no está instalado en el equipo del navegador y la instanciación se realiza desde la página HTML en vez de la página ASP.NET.

Notas respecto a la ejecución en otros entornos

Ya hemos indicado que las aplicaciones Silverlight se pueden usar en cualquier servidor Web. Esto incluye Apache sobre Linux, Oracle Web Server y muchos otros servidores populares en el mundo de la industria. Pero lo mismo es cierto para la página de alojamiento que puede ser cualquier página Web de servidor (PHP, Java, Python, Ruby, etc.).

NOTA. Sobre el tema de servidores y alojamiento, ver el capítulo final acerca de la configuración en servidores y otros aspectos relacionados.

Instanciación del objeto Silverlight en los dos modelos (HTML / ASP.NET)

En el primer caso (HTML/DOM, y valga para situaciones en las que el usuario necesite instanciar un elemento

Silverlight en una página existente), la instanciación se hace dentro de un elemento donde se sitúa una etiqueta Object que es la encargada de llamar al complemento y pasarle los parámetros iniciales. Este proceso se maneja con el código del listado 2-1.

Listado 2-1

```
<!-- Runtime errors from Silverlight will be displayed here.
   This will contain debugging information and should be removed or hidden
   when debugging is completed -->
<div id='errorLocation' style="font-size: small;color: Gray;"></div>
<div id="silverlightControlHost">
   <object data="data:application/x-silverlight-2," type="application/x-
       silverlight-2" width="100%" height="100%">
   <param name="source" value="ClientBin/SilverlightA pplication1.xap"/>
   <param name="onerror" value="onSilverlightError" />
   <param name="background" value="white" />
   <param name="minRuntimeVersion" value="2.0.31005.0" />
   <param name="autoUpgrade" value="true" />
   <a href="http://go.microsoft.com/fwlink/?LinkID=124807"
              style="text-decoration: none;">
         <img src="http://go.microsoft.com/fwlink/? LinkId=108181" alt="Get Microsoft
         Silverlight" style="border-style: none"/>
         </a>
    </object>
 <iframe style='visibility:hidden;height:0;width:0; border:0px'></iframe>
</div>
```

De todos estos parámetros pasados en la instanciación, algunos valores son fundamentales: data que establece el tipo de objeto que se desea (Silverlight), y type, donde le indicamos la versión del control. Lo recomendable es poner la versión mínima que se necesita para su funcionamiento, si bien podemos indicar este aspecto con más detalle en el parámetro de nombre minRuntimeVersion. Aquí, la secuencia application/x-silverlight-2 significa Siverlight 2.0 versión final, como podrá imaginarse el lector.

NOTA. La versión 1.0 se corresponde con el tipo MIME application/x-silverlight.

Manejo de distintas versiones del runtime

Esto permite, además, una aproximación más elegante al problema de las versiones, si es que se quiere tener en cuenta ese factor. Por ejemplo, podemos construir páginas distintas para cada versión del control. En este caso, el objeto Application puede comprobar cuál es la versión instalada del control y, en función de eso, lanzar una u otra página, o cargar distintas versiones de un control en la misma página (recordemos que el CoreCLR puede ejecutarse en paralelo).

No obstante, debemos tener en cuenta que la versión 2.0 soporta toda la funcionalidad de la anterior sin necesidad de cambios, por lo que un cliente con Silverlight 2.0 instalado puede ver perfectamente elementos enriquecidos con Silverlight 1.0 sin ningún aviso por parte del navegador. Lo contrario no es cierto, por supuesto.

NOTA. La presencia de la etiqueta IFrame vacía obedece a un comportamiento especial del navegador Safari. Se consigue así que Safarí no sitúe la página en caché, y por lo tanto permite que el control vuelva a cargarse si se retorna a una página Silverlight previamente visitada.

En el caso de la página ASP.NET, se utiliza una control especial de servidor para realizar la instanciación: `<asp:Silverlight>`, que se encarga de las mismas labores que el anterior, y además, permite otras posibilidades, como especificar qué queremos que vea el usuario en el caso de que Silverlight no se encuentre instalado en la máquina del cliente, y redirigir la salida hacia una interfaz amigable si se opta por la instalación del complemento, como veremos a continuación.

El código de esta etiqueta anterior tal como es creada por el IDE de Visual Studio es el siguiente:

Listado 2-2

El elemento <asp:Silverlight>

Hay varias ventajas añadidas en la utilización del control `<asp:Silverlight>`: primero la presencia del atributo `MinimumVersion`, que permite establecer la versión del complemento mínima para soportar la funcionalidad de nuestra página. Por supuesto, el atributo `Source` hace referencia al fichero una vez compilado y listo para distribución, pero podemos establecer todo un conjunto de propiedades visuales, como la anchura, altura, y muchas

otras, como se muestra en el Intellisense asociado al elemento, que vemos en la figura 2-3.

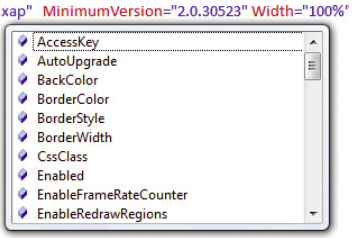


Figura 2-3. Distintos atributos asignables al elemento <asp:Silverlight>

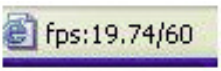
Y para los casos en los que el cliente no tenga instalado el complemento, si, optamos por convertir la etiqueta en contenedora de otros elementos (añadimos la etiqueta de cierre `</asp:Silverlight>`), veremos que el Intellisense nos ofrece un único elemento posible: `<PluginNotInstalledTemplate>` donde podemos colocar cualquier secuencia de controles ASP.NET (o HTML) para elaborar una interfaz de usuario personalizada y que sea ésta la que se le ofrezca al lector como alternativa en lugar de la predeterminada (el típico logo de la aplicación, con el enlace al sitio de descarga en Microsoft, etc.)[2].

NOTA. Observe igualmente que el fichero de configuración Web.config, contiene un buen número de referencias a librerías del CLR y otros valores de interés.

Existen otras propiedades a tener en cuenta, de las cuales me gustaría comentar las más interesantes:

[2]. Distintos atributos asignables al elemento **<asp:Silverlight>**

- **`EnableFrameRateCounter`**: permite visualizar en la parte inferior de la ventana del navegador los marcos por segundo que está procesando nuestro complemento Silverlight. Útil en depuración de ciertas situaciones. Aparece como en la siguiente captura de pantalla:



- **`EnableRedrawRegions`**: también útil para depuración. Si se establece su valor a True muestra cuando el control redibuja una región. Por ejemplo, para un círculo animado se obtendría una salida similar a la siguiente:

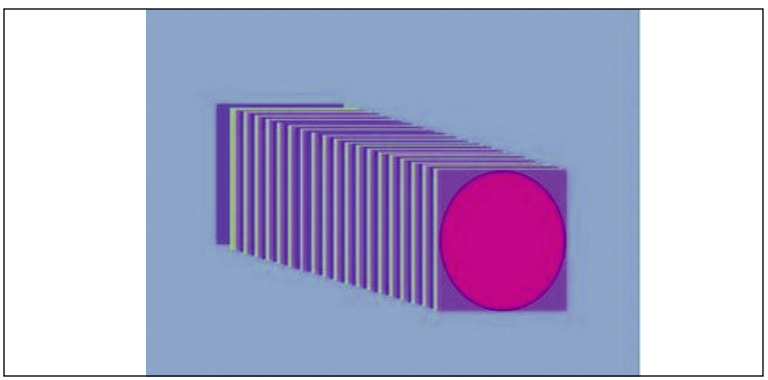


Figura 2-4. Salida de un gráfico Silverlight con la opción EnableDrawRegions habilitada

- **`SplashScreen`**: permite cambiar la ventana de splash que muestra Silverlight por defecto cuando está cargando el complemento. El proceso no es trivial y Microsoft ha publicado una página explicativa que lo detalla paso a paso:
  `http://www.silverlight.net/QuickStarts/BuildUi/SplashScreen.aspx`.

- `InitParameters`: Es la forma de pasar variables de inicialización. Por ejemplo, podemos incluir lo siguiente:

Listado 2-3

```
InitParameters="BaseUrl=http://live.com,ResourceUrl=http:   //www.microsoft.com"
 //Y, más adelante, en la clase A pp.xaml.cs…
private void A pplication_Startup(object sender, StartupEventArgs e)
     {
       string baseUrl = e.InitParams["BaseUrl"];
       string reasourceUrl = e.InitParams["ResourceUrl"];
       this.RootVisual = new Page();
}
// Los parámetros se pueden pasar en el constructor de la clase Page
```

La parte XAML de la aplicación Silverlight predeterminada

Una vez compilada esta aplicación inicial, veremos (figura 2-5) que hay dos ficheros XAML junto a sus correspondientes contrapartidas C# (o VB.NET): `App.xaml` y `Page.xaml`, y que el subdirectorio `Debug` contiene todo lo necesario como salida del proceso de compilación: la DLL y el fichero `.manifest`, que son el resultado principal, el fichero `.xap`, que es el resultado empaquetado y listo para distribuirse, la página de prueba HTML y un fichero `.pdb` que utiliza Visual Studio para soporte en la depuración. En principio para esta prueba no se necesita nada más.

De forma que el fichero `.XAP` al que referencia el control `<asp:Silverlight>`, es un entregable más de los producidos por el proceso de compilación. Esto es debido a que la DLL

solo alberga el código compilado, pero no otros recursos que podrían estar disponibles para el control Silverlight, como pueden ser imágenes o ficheros multimedia. Esto también dependerá de cómo declaremos el elemento. La figura 2-5[3] muestra la relación de inclusión e interoperabilidad de los componentes, junto a la estructura de ficheros.

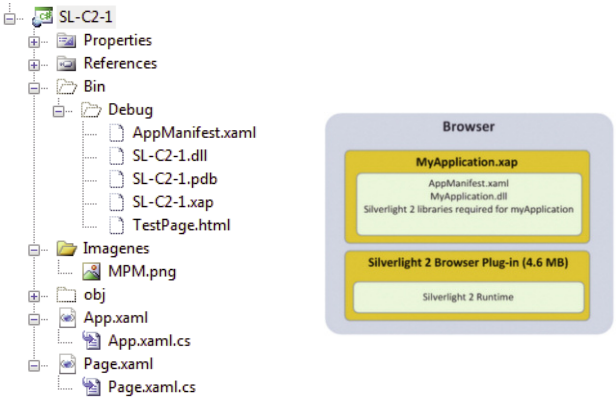


Figura 2-5. Estructura de ficheros y diagrama conceptual de una aplicación Silverlight

De hecho, ya comentamos que el fichero `.XAP` es un fichero comprimido ZIP con la extensión cambiada. Si renombramos la extensión a `.ZIP` veremos que se puede descomprimir y contiene la DLL más el fichero `.manifest` correspondiente. El resto de recursos que se usen en la aplicación se compilan junto con la DLL que se distribuye, y la forma en que pueden ser accedidos depende un tanto de la propiedad "Build Action" asociada a cada recurso. En el

[3]. Tomado del sitio oficial "SilverlightBlend" `http://silverlight blend.blogspot.com/2008/08/silverlightinteroperabilidad-de-java.html`

capítulo final dedicado a la instalación y configuración de aplicaciones, dedicamos un apartado a este tema.

El flujo de procesos dentro de la aplicación Silverlight

Por tanto, tenemos una página (HTML o ASP.NET) que hace referencia a un cierto fichero de extensión **.XAP** donde se encuentra toda nuestra aplicación Silverlight. A partir de ahí, entra en juego el runtime y el punto de entrada en la aplicación lo encontramos en el fichero **App.xaml.cs**, cuyo contenido puede ver en el listado 2-4.

Listado 2-4

```
public partial class A pp : Application
{
  public A pp()
  {
    this.Startup += this.A pplication_Startup;

    this.Exit += this.A pplication_Exit;
    this.UnhandledException+= this.Application_Un handledException;

    InitializeComponent();
  }

    private void A pplication_Startup(object sender, StartupEventArgs e)
  {
        this.RootVisual = new Page();
  }
    private void A pplication_Exit(object sender, EventA rgs e) {}
```

```
   private void A pplication_UnhandledException(object sender,
                           ApplicationUnhandledExceptionEventArgs e)
 {
   // If the app is running outside of the debugger then report the
   // exception using the browser's exception mechanism. On IE this will
   // display it a yellow alert icon in the status bar and Firefox will
   // display a script error.
   if (!System.Diagnostics.Debugger.IsAttached)
   {
     // NOTE: This will allow the application to continue running after
     // an exception has been thrown but not handled.
     // For production applications this error handling should be
     // replaced with something that will
     // report the error to the website and stop the application.
     e.Handled = true;
     Deployment.Current.Dispatcher.BeginInvoke(delegate    { ReportErrorToDOM(e);
     });
   }
 }

private void ReportErrorToDOM(ApplicationUnhandledExceptionEventArgs e)
{
   try
   {
     string errorMsg = e.ExceptionObject.Message + e.ExceptionObject.StackTrace;
     errorMsg = errorMsg.Replace('"', '\''). Replace("\r\n", @"\n");
     System.Windows.Browser.HtmlPage.Window.Eval(
               "throw new Error(\"Unhandled Error in Silverlight 2 Application " +
               errorMsg + "\");");
   }
   catch (Exception)
```

```
    {
    }
  }
}
```

El objeto App (que hereda de Application), define 4 eventos: **`OnStartup`**, **`OnExit`**, **`Application_UnhandledException`** *y* **`ReportErrorToDOM`**. El diseñador incluye el código para los manejadores de evento de todos ellos.

En el primero, (que recibe un argumento **`StartupEvetArgs`**) se define el contenido de la propiedad **`RootVisual`** (por así decirlo, el contenedor del control Silverlight, vacío de contenido) como el resultado de llamar al constructor de la clase **`Page`** (que hereda de **`UserControl`**).

`RootVisual` permite recuperar o establecer la interfaz de usuario principal de la aplicación. Tiene dos usos posibles en este contexto: por un lado, la opción predeterminada, que genera el fichero de pruebas citado antes, y por otro, podemos definir nuestra interfaz personalizada de usuario justo en el momento de la asignación inicial, pero, esto solo puede hacerse una vez (podríamos decir que es inmutable, al igual que las cadenas en .NET, una vez inicializado).

Además, en la versión final del producto, se incluye código para el tratamiento de excepciones mediante el procedimiento de evento **`Application_UnhandledException`**, que deberemos codificar expresamente. Este procedimiento llama de forma asíncrona a **`Report ErrorToDOM`**, de manera que la aplicación pueda seguir funcionando, aunque se nos indica que para aplicaciones de producción deberemos hacer nuestro propio tratamiento de error.

Por lo demás, esta forma de comienzo de la aplicación permite, entre otras cosas, asignar distintas instancias de una clase al objeto `RootVisual`, por ejemplo, dependiendo del contexto de ejecución o de otros parámetros que se consulten en el proceso inicial.

Si compilamos la aplicación tal cual está, el IDE nos indicará que la compilación se ha realizado con éxito, y la primera vez que se ejecuta, nos preguntará si deseamos habilitar la depuración Web (ver figura 2-6). Si es así, se incluirá automáticamente en el fichero `Web.config` la información necesaria para permitirlo (básicamente, el cambio de valor del atributo `Debug` a `true`, deshabilitado inicialmente). Por lo demás, no hay diferencias significativas en cuanto a las posibilidades de depuración de que se dispone respecto a otras aplicaciones clásicas (ya sean Windows Forms o ASP.NET).

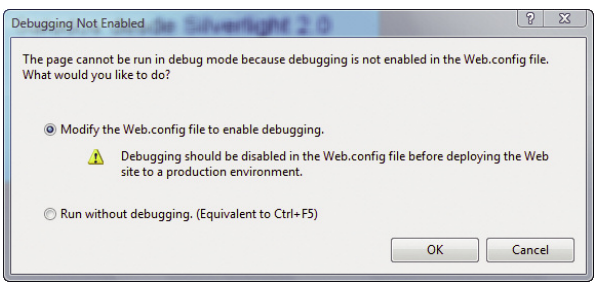


Figura 2-6. Ventana de selección de modo de ejecución

Comprobación de la versión de Silverlight instalada en el cliente

Ya hemos visto que, tanto la etiqueta `<object>` (desde la página HTML) como el control `<asp:Silverlight>` (desde la

página ASP.NET), incluyen mecanismos para requerir del cliente una versión mínima del complemento antes de lanzar la aplicación. En ambos casos, y salvo acción programada en contrario, el sistema detecta esa situación y redirige al usuario a la página de Microsoft adecuada para que proceda con la descarga del complemento.

Si queremos averiguar la versión de Silverlight presente en el equipo del cliente, existen varios métodos de comprobación. Quizá el más sencillo sea utilizar una llamada justo antes de crear el objeto **`RootVisual`** y verificar el soporte para una versión determinada. Incluso podríamos llamar a la clase **`MessageBox`**, para mostrar una caja de diálogo con esa información al iniciarse la aplicación:

```
MessageBox.Show(this.Host.IsVersionSupported("2.1") .ToString());
```

En este ejemplo, mostraríamos si una —hipotética— versión 2.1 está, o no, soportada, obteniendo un resultado **`False`**. Las opciones posibles en la actualidad son las que describimos en la tabla adjunta, con sus respectivas referencias en la Web, y números de versión.

Versión	Tipo MIME	Número	URL para instalación de versión
Silverlight 1.0	application/x-silverlight	1.0	http://go2.microsoft.com/fwlink/?LinkId=110408
Silverlight 2 Beta 1	application/x-silverlight-2-b1	2.0.30226	http://go2.microsoft.com/fwlink/?LinkId=108182
Silverlight 2 Beta 2	application/x-silverlight-2-b2	2.0.30523	http://go2.microsoft.com/fwlink/?LinkID=115261
Silverlight 2.0	application/x-silverlight-2	2.0.31005.0	http://go2.microsoft.com/fwlink/?LinkID=108181

Tabla de versiones disponibles de Silverlight.

El ciclo de vida de una aplicación Silverlight

El ciclo de vida de una aplicación Silverlight comienza cuando el control es instanciado por la página que lo aloja, y termina con cualquiera de los siguientes supuestos:

- El navegador se desplaza a otra página Web.
- Se cierra la ventana del navegador o la pestaña que muestra la página host.
- Se refresca la página Web que lo aloja (en ese caso comienza un nuevo ciclo de vida).
- El *script* de la página utiliza HTML DOM para eliminar el control Silverlight de la página.
- Se cierra la sesión de usuario del sistema operativo o se cierra éste.

En estos casos, la aplicación puede realizar acciones de monitorización, auditoría o grabado de información, aprovechando el evento **`OnExit`**, accesible como miembro del objeto **`Application`**.

NOTA. A este respecto hay una recomendación "oficial" con la que estamos totalmente de acuerdo: respetar el deseo del usuario de realizar la salida completa de la página, teniendo cuidado de no añadir en el manejador de ese evento código cíclico o re-entrante, tal como reiniciar la propiedad Source del control Silverlight: si el usuario quiere cerrar la ventana, evitemos la tentación de despedirnos con un "numerito".

3

Expresión Blend para desarroladores

Continuamos hablando de desarrollo en Silverlight 2, pero esta vez con Expression Blend 2.0[1] SP1 como herramienta. A partir de este momento, siempre que nos refiramos a Blend, estaremos haciendo mención de esta edición del producto. Empezaremos por hacer una pequeña aplicación que nos vaya mostrando las características de funcionamiento y comparando la organización de su interfaz de usuario (y la forma de operar en ella) con la de Visual Studio.

Desarrollo de una aplicación Silverlight básica con Expression Blend 2.0

Tras abrir Blend, seleccionaremos "New Project", y lo primero que vemos es que existen dos tipos de proyectos Silverlight, uno para cada versión: 1.0 ó 2.0 (ver figura 3-1).

La diferencia es que —para el primer caso— se trata de un "Sitio Silverlight", mientras que en el segundo es una

[1]. El soporte de aplicaciones Silverlight 2 lo suministra el Service Pack 1 de Blend 2.0

"Aplicación Silverlight". Para la versión 1.0, Blend construye un sitio básico predeterminado, con una página desde donde se carga el control Silverlight 1.0; para la versión 2.0, se trata de toda una solución, con las referencias a las librerías .NET necesarias para el desarrollo, etc.

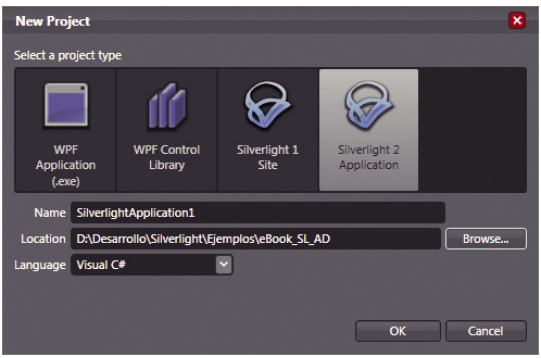


Figura 3-1. Oferta inicial de proyectos en Blend 2.0 tras la instalación del SDK 2.0 de Silverlight

En la ventana del Explorador de proyectos se muestra la estructura de ficheros exactamente igual que si lo hubiésemos creado con Visual Studio 2008, a excepción del sitio Web de soporte, que en Blend no se genera, pues para probar los proyectos, siempre crea una página dinámica.

En la figura 3-2, vemos la ventana "Project": la misma estructura de ficheros y directorios que cabría esperar de una aplicación Silverlight 2 iniciada con Visual Studio, incluyendo las referencias a las librerías (ver la sección "References" de la figura 3-2), el fichero `Application.manifest`, que establece comportamientos de la aplicación, y, claro

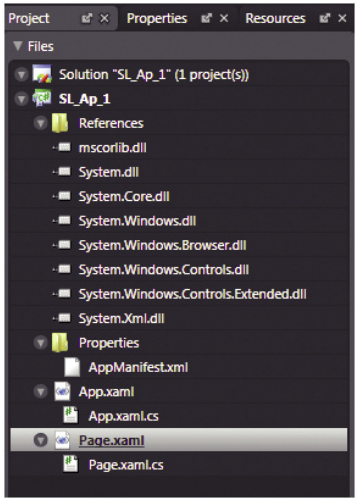


Figura 3-2. Elementos típicos de una aplicación Silverlight, en la solapa Project

está, los ficheros principales `App.xaml` y `Pa ge.xaml`, con sus contrapartidas de clases en C#.

Veremos la diferencia respecto al proyecto creado con Visual Studio al abrir uno de los dos ficheros C#, ya que se lanza inmediatamente Visual Studio 2008, cargando el fichero, y manteniendo ambos sincronizados al cambiar de un entorno a otro (siempre tras un aviso al usuario de que el fichero se ha modificado en otra aplicación exterior). Esto es así, porque, por el momento, Blend no soporta la edición de ficheros para C# o VB.NET.

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 2.0

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. En la figura 3-3 se aprecia el aspecto general del entorno de diseño con los elementos que indicamos:

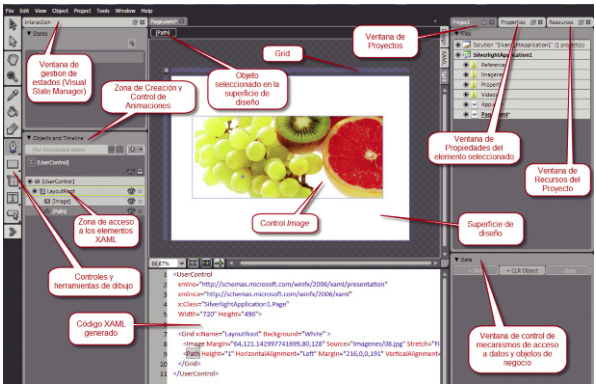


Figura 3-3. Entorno de trabajo de Blend, con descripciones de sus elementos predeterminados.

A la izquierda, en lugar de la típica barra de herramientas con controles, disponemos de una serie de iconos que despliegan (con el botón derecho) menús laterales agrupados por acciones comunes de diseño y dibujo. Ahí se encuentran las referencias a los controles principales, por categorías, junto con una ventana denominada “Objects & Timeline”, que nos ayuda a seleccionar los objetos y crear animaciones en el diseño. En el capítulo 6, veremos cómo

manejar esta opción y la creación de Transformaciones y Animaciones.

Elementos principales del IDE

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 muy accesible.

Controles y geometrías

El desarrollador de Visual Studio está acostumbrado a que la Caja de herramientas contenga todo lo que la IU de su aplicación pueda mostrar. Blend dispone (por el momento) de menos controles, pero, como contrapartida, permite crear otro tipo de objetos de IU: las Geometrías. Son elementos de dibujo de carácter vectorial creados por el usuario en una "sesión de dibujo", y por lo tanto no pueden estar predeterminados. Blend habilita para ello las herramientas Pen (dibujo a mano alzada), Pencil (dibujo vectorial asistido), Ellipse, Rectangle y Line.

El resultado de estas sesiones de dibujo se expresa como elementos XAML vectoriales y por tanto de alta calidad gráfica. El trabajo conjunto con ambos elementos permite conseguir interfaces creativas con gran sencillez.

NOTA. El lector puede comprobar esto último practicando con los objetos **`Pen`** y **`Pencil`**.

¿Y dónde están los controles de interfaz de usuario? Al final de la lista de herramientas, en el apartado "Assets", encontramos una ventana desplegable con la oferta de todos los controles disponibles, teniendo en cuenta que es de esperar que sigan creciendo paulatinamente a medida que el producto evolucione (así lo han anunciado oficialmente los jefes de producto, y —de hecho— ya está disponible el nuevo Silverlight Toolkit, como apuntamos al final del capítulo 5.).

Seleccionando la opción "All controls" en 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 3-4.

NOTA. En Visual Studio 2008, no es posible arrastrar un control Silverlight directamente a la ventana de diseño. De momento, ha de hacerse en la ventana de código, y el XAML generado se interpreta visualmente en la ventana de diseño.

Con Expression Blend disponemos de todos los recursos típicos de una herramienta gráfica, cosa que comprobamos con solo arrastrar un control a la ventana de diseño y observar los utensilios visuales que se activan automáticamente y el código XAML generado por el editor.

Creación automática de elementos a partir de recursos gráficos

Cuando se selecciona un recurso de la aplicación que tiene asociado un elemento visual (como es el caso de una imagen, asociada al control Image, o un vídeo y su correspondiente

`MediaElement`), Blend crea automáticamente ese elemento y le asigna las propiedades predeterminadas, más las propias del recurso (anchura, altura, etc.), de forma que podemos verlo directamente en la Ventana de diseño, listo para funcionamiento. En el caso de que se trate de un vídeo, téngase presente que la propiedad de reproducción automática estará activada por defecto.

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

Más sobre los Assets

Dentro de la ventana "Assets", podemos dividir los recursos disponibles por categorías, de forma 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", los controles de usuario en "Custom Controls" (por el momento solo aparece la clase `Page`), y los usados más recientemente, los tendremos en la solapa "Recent". También podemos optar por el modo de vista "Details", que nos muestra la ubicación física de las DLL a las que pertenece cada control, y el espacio de nombres en que está definido.

Una característica muy importante de los controles (o "elementos de diseño de IU") es su total flexibilidad. Cualquier control puede ser modificado en sus propiedades o en las plantillas que definen su aspecto y comportamiento visual, de forma que se adapte, literalmente, a cualquier entorno imaginado por el usuario.

NOTA. Aunque no tiene que ver directamente con el desarrollo en Silverlight, la herramienta Expression Design 2, permite exportar diseños complejos en un formato utilizable por Blend. Y lo mismo sucede con algunas herramientas de terceros como Adobe Illustrator 3.

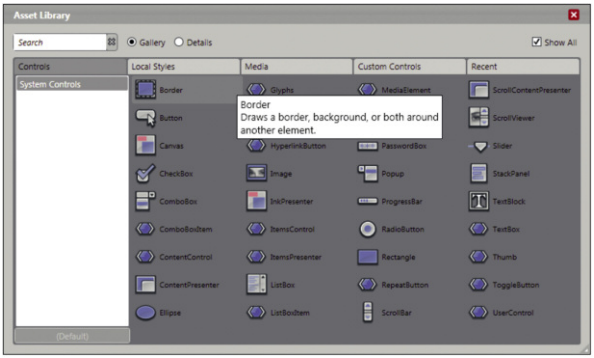


Figura 3-4. Librería de controles (Asset Library) disponible para una aplicación Silverlight 2.0.

Categorías de controles

Desde el punto de vista iconográfico, existen dos categorías de controles: aquellos que poseen su propio glifo diferenciador, y los que comparten un solo icono general (), como es el caso de **`HyperLinkButton`**, **`MediaElement`**, y otros. En todos los casos, el IDE generará el código XAML correspondiente, si bien la opción de Intellisense, se reduce —por el momento— al editor XAML de Visual Studio 2008.

En realidad, esta diferenciación iconográfica no está hecha al azar. Solo poseen icono propio los elementos que, de forma predeterminada, muestran una interfaz bien definida. 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 para trabajar con él, la ventana de propiedades muestra los valores que ese objeto tiene asignados: tanto explícitamente (por haberlo dibujado en la ventana de diseño), como los predeterminados. Y esto es aplicable también a otras capacidades, como el enlace a datos a través de atributos `DataContext` y similares.

Con estos sencillos principios puede el lector comenzar a hacer sus diseños visuales en Blend 2.0, y observar cómo el IDE siempre responde actualizando automáticamente los cambios hechos en cualquiera de las ventanas de edición.

Ejemplo Inicial

Empezamos con una prueba muy simple. Hemos creado un par de directorios en la aplicación para albergar gráficos y vídeos. Podemos hacerlo desde el menú "Project" y seleccionar "Create New Folder" o directamente sobre el Explorador de proyectos con el botón derecho, tal y como muestra la figura 3-5.

Una vez creado el directorio, copiamos y pegamos algunas imágenes para hacer referencia a ellas al programar el evento `Click`. Esto es así, porque, si seleccionamos la opción "Add Existing Item", lo que vamos a obtener es una referencia al fichero en cuestión, pero éste seguirá estando su ubicación original, y no dispondremos de una copia local para distribuir. Más adelante volveremos en detalle a los recursos embebidos.

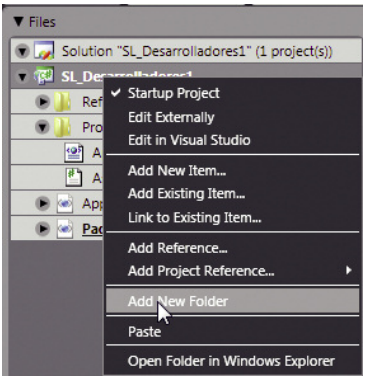


Figura 3-5. Explorador de proyectos mostrando el menú contextual

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 cómo 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.

NOTA. Hay que recordar, que —literalmente— 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 2 (por ejemplo, **`.Gif`** no lo está, —de momento— por problemas de integración de sus capacidades de animación, que Microsoft está considerando).Ver el capítulo 1 para más detalles sobre formatos soportados.

Probando la solución inicial

De hecho, si probamos la aplicación (con [F5] o la opción de menú "Project"-> "Test Solution") veremos que, sin haber escrito una línea de código, tenemos un control con una imagen y un vídeo funcionando. Blend 2.0 generará una nueva página para probar el control (llamada **`Default.htm`**), que contiene lo básico para la instanciación de un control Silverlight 2.0 tal y como se ha visto en el capítulo anterior. Si el lector quiere verificar cuál es exactamente el código que se genera, puede hacerlo lanzando la aplicación y abriendo el código fuente desde el navegador.

Con lo hecho hasta el momento, el código producido tiene el aspecto que puede ver en el listado 3-1 (téngase en cuenta que incluye transformaciones como la rotación del vídeo).

Listado 3-1

```
<UserControl
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="SilverlightA pplication1.Page"
   Width="720" Height="496" Background="#FFD 5D237">
   <Grid x:Name="LayoutRoot" Background="#FFBDBC9D" >
```

```
    <Image Margin="352,144,-89.19899749755 86,156.057998657227"
          Source="Imagenes/08.jpg" Stretch= "Fill"
          RenderTransformOrigin="0.5, 0.5">
      <Image.RenderTransform>
        <TransformGroup>
          <ScaleTransform/>
          <SkewTransform/>
          <RotateTransform A ngle="89.616 "/>
          <TranslateTransform/>
        </TransformGroup>
      </Image.RenderTransform>
    </Image>
    <MediaElement Margin="52.2299995422363, 94.7099990844727,232,112"
        x:Name="Bear_wmv" Source="Videos/Be ar.wmv" Stretch="Fill"
                                      RenderTrans formOrigin="0.5,0.5">
      <MediaElement.RenderTransform>
        <TransformGroup>
          <ScaleTransform/>
          <SkewTransform/>
          <RotateTransform Angle="-19.41"/>
          <TranslateTransform/>
        </TransformGroup>
      </MediaElement.RenderTransform>
    </MediaElement>
   </Grid>
</UserControl>
```

Y, si ejecutamos la aplicación, obtendremos la salida visual del control Silverlight mostrado por la página en Internet Explorer 8 beta 2 que puede ver en al figura 3-6.



Figura 3-6. Resultado de la ejecución del programa anterior en Internet Explorer 8.0 Beta 2.

Vemos que Blend es muy intuitivo de manejar, y casi siempre nos garantiza una aproximación al problema gráfico a resolver desde el punto de vista del 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: dibujo, modificación de propiedades en la Ventana de diseño, y por último, ajuste fino en la Ventana de propiedades.

Vamos a seguir con el mismo proyecto, lo que nos dará una excusa para ver la forma en que se puede disponer de varias páginas y lanzar una u otra manejando el código C# subyacente. 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 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 en el listado 3-2.

Listado 3-2

```
<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/ex pression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   mc:Ignorable="d" x:Class="SilverlightAppli cation1.Dibujos"
   d:DesignWidth="640" d:DesignHeight="480">

   <Grid x:Name="LayoutRoot" Background="White" >
     <Ellipse HorizontalA lignment="Stretch" Ver ticalAlignment="Stretch"
       Fill="#FF652DC2" Stroke="#FF000000"
       RenderTransformOrigin="3.20000004768372 ,2.48000001907349"
       Margin="112,56,144,56"/>
```

```
    <Rectangle HorizontalA lignment="Stretch" Margin="176,168,208,168"
        VerticalAlignment="Stretch" Fill="  #FFD0C590" Stroke="#FF000000"
        RadiusX="40" RadiusY="40"/>
    <TextBox Margin="216,224,240,216" Text="Text Box" TextWrapping="Wrap"
                                                          FontSize="24"/>
    </Grid>
</UserControl>
```

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 controles contenedores (elementos que hereden de **Panel**), por lo que la visibilidad en condiciones de solapamiento es la predeterminada en la, propiedad **z-order**, esto es, el primero que se dibuja va más al fondo, y los demás se sitúan por encima, y así sucesivamente. Esto puede cambiarse en cualquier momento.

En ejecución, como en este ejemplo el control central es un **TextBox**, el usuario puede modificar el texto inicial, pero esa es toda la funcionalidad disponible (ver figura 3-7).

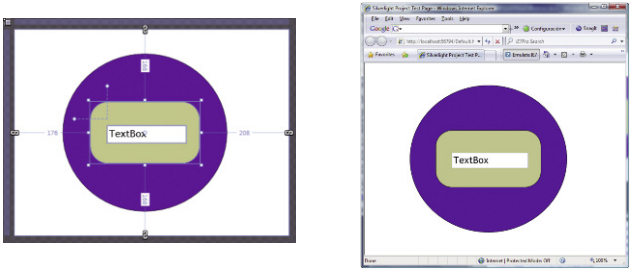


Figura 3-7. Ventana de edición y salida del ejemplo de gráficos (la salida es idéntica para FireFox y Safari).

No hemos codificado nada para recoger la entrada del usuario en el `TextBox`. Se trata, simplemente, de demostrar que el aspecto y el funcionamiento es equivalente a sus contrapartidas en Windows Presentation Foundation.

Cambio del punto de entrada de la aplicación

Sin embargo, 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`, y que sirve como punto de entrada, crea un objeto `Page` para instanciar la propiedad `RootVisual`. Basta con cambiar la instanciación a `Dibujos`, para tener el ejemplo funcionando (listado 3-3).

Una vez hecho esto, deberemos ver la salida correcta en el navegador.

Listado 3-3

```
private void OnStartup(object sender, StartupEventA rgs e)
{
  // Load the main control here this.RootVisual = new Dibujos();
  //this.RootVisual = new Page(); (modificamos esta entrada)
}
```

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. 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.

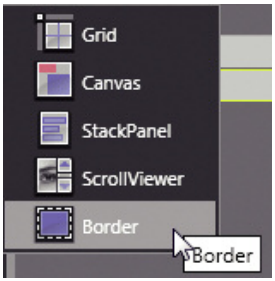


Figura 3-8. Ventana de edición y salida del ejemplo de gráficos (la salida es idéntica para FireFox y Safari).

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 otro elemento `UserControl`) que presenta 2 modos operativos: modo `Canvas` y modo `Grid`, que pueden alternarse pulsando en el símbolo .

La diferencia estriba en la forma de ubicar los elementos dentro del `Grid`. 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.

NOTA. Mediante la combinación de teclas [CTRL]+[Rueda de Ratón] podemos cambiar el tamaño del elemento UserControl en edición. De la misma

forma, si pulsamos la barra espaciadora, el cursor se convierte en una mano con la que podemos mover la superficie diseñada 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 control 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 3-9.

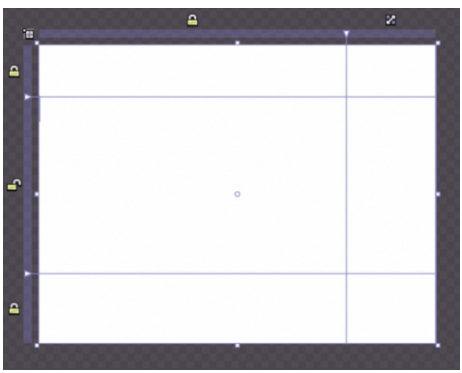


Figura 3-9. Resultado del proceso de edición de filas y columnas.

Las marcas anteriores producen como salida el código XAML del listado 3-4 (tengamos en cuenta que los números de las filas y columnas en un *grid* comienzan por 0).

Listado 3-4

```
<Grid x:Name="LayoutRoot" Background="White" >
   <Grid.ColumnDefinitions>
       <ColumnDefinition Width="496"/>
       <ColumnDefinition Width="Auto" MinWidth="144"/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
       <RowDefinition Height="83.179"/>
       <RowDefinition Height="*"/>
       <RowDefinition Height="111.952"/>
   </Grid.RowDefinitions>
   <Grid Height="39" HorizontalAlignment="Left" Margin="0,-0.179,0,0"
                    VerticalAlignment="Top" Width="0" Grid.Row="1"/>
</Grid>
```

Brushes

En parte, Silverlight sigue los patrones de diseño visual que establece la especificación de documentos XPS[2], basada en XAML y presente en Windows 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 2.0.

Silverlight 2.0 presenta cinco[3] tipos de *brushes*: **SolidColorBrush**, **LinearGradient Brush**, **RadialGradienBrush**, **ImageBrush** y **VideoBrush**, pero vamos a empezar por los más "mediáticos":

2. Ver **http://en.wikipedia.org/wiki/XML_Paper_Specification**

3. En realidad existe un sexto tipo: **TileBrush**, pero su uso parece dar algún problema de memoria.

ImageBrush y VideoBrush, y comprobaremos lo sencillo que resulta su uso desde este entorno.

Nombre	Descripción
Solid Color Brush	Rellena una región con un color sólido
Image Brush	Rellena una región con una imagen
Visual Brush	Rellena una región con un dibujo
Linear Gradient	Brush Rellena una región con un gradiente lineal
Radial Gradient Brush	Rellena una región con un gradiente radial

Tabla 1: Tipos de Brushes.

ImageBrush

ImageBrush permite dibujar la superficie de un control o geometría utilizando para ello una imagen de la que dispongamos como recurso en nuestra 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, de manera que podamos 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 (elemento "Image") seleccionada, 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 siguiente:

Listado 3-5

```
<UserControl.Resources>
   <ImageBrush x:Key="ImageBrush1" ImageSource="Gra ficos/deepzoomcomposer.png"/>
</UserControl.Resources>
```

La comprobación visual aparecerá en la solapa "Resources" de la ventana superior derecha, como se ve en la figura 3-10.

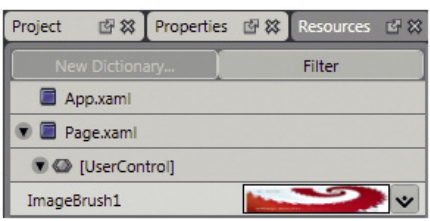


Figura 3-10. Recurso de imagen en Blend.

Posteriormente, resultará muy sencillo hacer que cualquier control adopte esa imagen como fondo utilizando una propiedad adecuada del control (como **Background**), y estableciendo un enlace dinámico con él. Veremos todo esto con más detalle más adelante, pero el código para vincular uno a otro es intuitivo y tiene esta forma:

Listado 3-6

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

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 disponible al lado de cada propiedad que está señalado por un pequeño cuadrado (ver figura 3-11), y seleccionar el recurso (aquí, **`ImageBrush1`**):

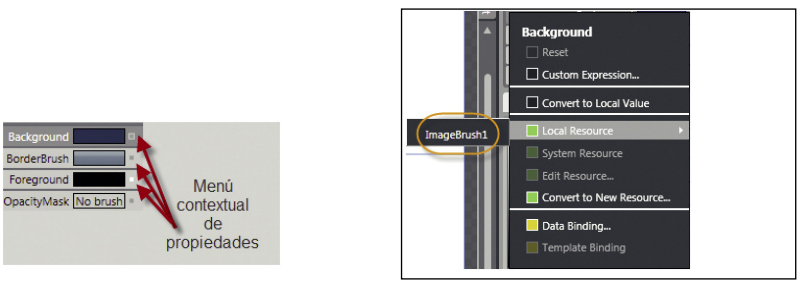


Figura 3-11. Acceso al menú contextual de propiedades y propiedades del atributo **`Background`**.

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 queremos 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 no lo hacemos como un recurso de la aplicación,sino indicándolo directamente en el código XAML del objeto `ImageBrush`.

Volveremos al tema del las geometrías en el capítulo 4, pero el aspecto que tiene la vinculación por código es el siguiente:

Listado 3-7

```
<Path Stroke="#008000">
   <Path.Fill>
     <ImageBrush ImageSource="Graficos/Silverlight2.0.png" Stretch="UniformToFill"/>
   </Path.Fill>
   <Path.Data>
            <PathGeometry>
        <PathFigure IsClosed="True" StartPoint="150,50">
            <LineSegment Point="150,275" />
            <LineSegment Point="500,50" />
                 <LineSegment Point="500,275" />
        </PathFigure>
     </PathGeometry>
  </Path.Data>
</Path>
```

Donde `Path` es el objeto tipo `Geometry` a usar en el dibujo, 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 3-12:

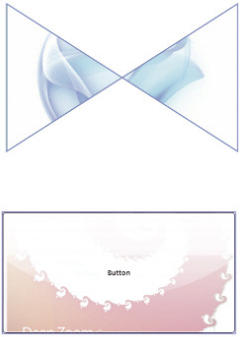


Figura 3-12. Utilización de ImageBrush en controles y geometrías

VideoBrush

Como extensión de lo anterior, una capacidad de esta forma de relleno de elementos gráficos es la de hacer que la brocha a utilizar 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 fondo es válido, incluyendo un `TextBlock`.

Listado 3-8

```
<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="Maro on" BorderThickness="12" Corne-
 rRadius="24" />
 <MediaElement Source="Videos/B.wmv" x:Name="Video_ Brush" Opacity="0" />
 <TextBlock Text="dot Net Mania" TextWrapping="Wrap" FontSize="120"
        FontFamily="Arial" Margin="15,20, -210,0"
        FontWeight="Bold" Width="640" Horizon talAlignment="Center" >
            <TextBlock.Foreground>
              <VideoBrush SourceName="Video_Brush" Stretch="Fill" />
            </TextBlock.Foreground>
       </TextBlock>
   </Grid>
</UserControl>
```

El único truco está en no mostrar el vídeo a pesar de que lo declaramos como `Media Element`; para ello, establecemos su valor de `Opacity` a `0`. Lo demás, resulta bastante evidente a la vista del código, y similar al ejemplo anterior.

En la salida, las letras del texto «**dotNet-Manía**» son pintadas con un popular vídeo de demostración. Volveremos sobre el tema de los recursos más adelante, pero sirva esta demo para comprobar las posibilidades de esta técnica. 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 (todas sincronizadas, ver figura 3-13).

Otros Brushes

Si optamos por rellenar un elemento mediante colores, el proceso es también muy intuitivo. Solo tenemos que escoger el objeto que queremos colorear, y observar el Panel de propiedades a la derecha. En la figura 3-14 puede verse 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 utilizable igualmente, en la solapa "Resources").



Figura 3-13. Letras cuyo fondo es un vídeo en formato .wmv

La oferta de propiedades a colorear 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*).

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

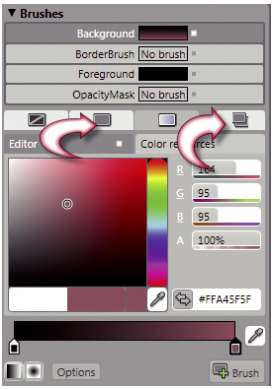


Figura 3-14. Editor de colores basados en objetos Brush

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 `GradienStop`, 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 inferior del gráfico, se presentan los botones de selección de tipo de gradiente (puede ser lineal o radial). Disponemos también de una gama de opciones dentro de la ventana de edición de brochas, para establecer las pautas de dibujo de los gradientes.

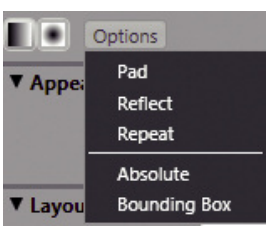


Figura 3-15. Selección del tipo de gradiente y opciones de gradiente

El resultado de estas operaciones puede tener un aspecto similar al siguiente:

Listado 3-9

```
<Grid.Background>
   <LinearGradientBrush EndPoint="320,480" StartPoint="320,0"
                        SpreadMethod="Pad" MappingMode="A bsolute">
      <GradientStop Color="#FF000000"/>
      <GradientStop Color="#FF484259" Offset="1"/>
      <GradientStop Color="#FF4616A 8" Offset="0.268 00000667572021"/>
      <GradientStop Color="#FF22906A " Offset="0. 48199999332427979"/>
      <GradientStop Color="#FF6D316F" Offset="0. 69599997997283936"/>
   </LinearGradientBrush>
</Grid.Background>
```

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 3-16, "Layout" nos ofrece todas las propiedades disponibles, reconociendo dónde se encuentra ubicado el control.

Los nombres de las propiedades resultan explicativos y —en muchos casos— son idénticos a sus contrapartidas en Windows Forms. No obstante, tenga en cuenta que la flecha que aparece en la parte inferior de

esa sub-ventana, despliega otra (continuación de ésta) que permite hacer un ajuste más fino de otras propiedades: anchuras y alturas máximas y mínimas, y propiedades del contenido del botón (el texto).

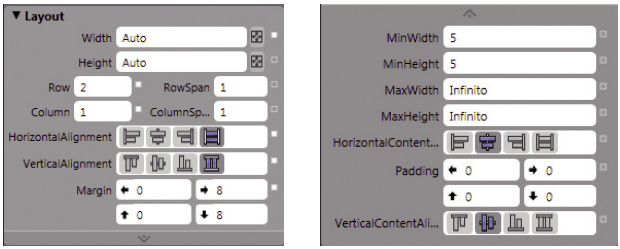


Figura 3-16. Ventanas del editor de posiciones de un elemento (Layout), simple y desplegada

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 del panel: 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 3-17).

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 (figura 3-18).

Resulta igualmente intuitivo establecer el formato del párrafo ( ) y ajustar las características de sangrado ( ).

Todos los elementos visuales que podemos utilizar aquí se encuentran representados por estas ventanas.

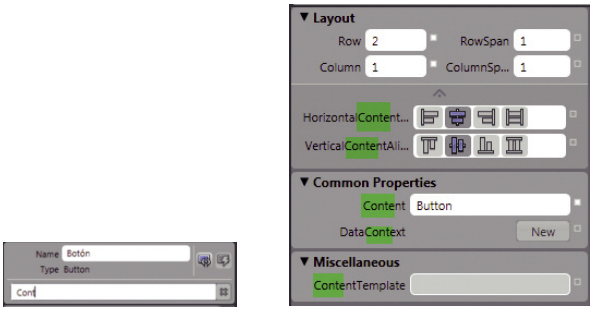


Figura 3-17. Caja de selcción de propiedades en la parte superior del panel y resultados de una búsqueda.

Finalmente, la sub-ventana "Miscelánea", contiene referencias a un conjunto de propiedades varias que pueden ser muy distintas para cada control.

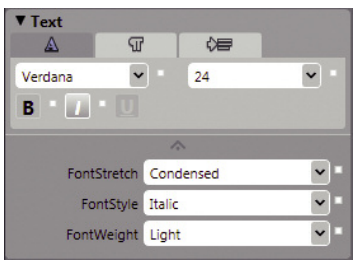


Figura 3-18. Sub-ventana de selección de las propiedades del texto de un elemento

Un pequeño visor de fotografías

Llegados a este punto vamos a programar un pequeño Visor de portadas de la revista **dotNetManía**, de forma

que se carguen las imágenes de dinámicamente. Para ello, es conveniente contar con una carpeta de aplicación separada para las imágenes, donde copiamos unas cuantas portadas.

La interfaz llevará un elemento `Image` para mostrar las portadas que nos interesen, y un botón para ir cambiando las fotos que veamos, según pulsamos. Pondremos el texto "Pulse" en el botón (el código para el cambio de fotografías se añadirá en la clase de soporte).

La verdad es que podemos usar varios controles distintos para mostrar una imagen. Desde el control `Image`, que —supuestamente— es el recomendado, hasta `Buttons`, `Rectangles`, `Borders`, `ListBoxes`, etc. Bastará con que dibujemos el control correspondiente en la superficie de diseño, le asignemos el tamaño y la ubicación adecuada, y más adelante, asignemos la pulsación sobre el botón en el código subyacente.

El aspecto del control después de añadir un botón y ubicarlo en la parte inferior derecha del `Grid` podría ser similar al de la página siguiente.

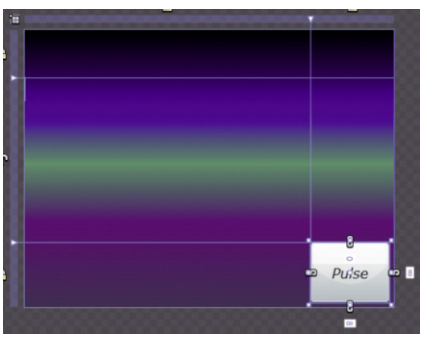


Figura 3-19. Ventanas del editor de posiciones de un elemento (Layout), simple y desplegada

Puede ver el correspondiente código XAML generado hasta el momento por Blend en el listado 3-10.

Para asignar el evento correspondiente a la pulsación del botón, seleccionaremos el símbolo  , que abre la Ventana de eventos del control. En el evento **`Click`**, y nada más escribir el nombre del manejador (**`Siguiente_Portada`**), veremos cómo se abre Visual Studio 2008 con el mismo proyecto y la ventana de código correspondiente a **`Page.xaml.cs`**, donde ya aparece creado el procedimiento de evento.

Programando un procedimiento de evento

Para codificar el cambio de imagen tenemos que tener en cuenta dos cosas: primera, que necesitamos un objeto URI que defina sin ambigüedad la ubicación del gráfico; segunda, que nos hace falta generar en memoria un gráfico compatible con el control **`Image`**. Un buen candidato es el objeto **`BitmapImage`**, y el proceso puede simplificarse hasta llegar a lo que puede ver en el listado 3-11.

Listado 3-10

```
<UserControl xmlns=http://schemas.microsoft.com/winfx /2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    x:Class="SL_Desarrolladores1.Page" Width="640" Height="480">
  <Grid x:Name="LayoutRoot">
    <Grid.Background>
            <LinearGradientBrush EndPoint="320,480" StartPoint="320,0"
                        SpreadMethod="Pad" MappingMode="Absolute">
      <GradientStop Color="#FF000000"/>
      <GradientStop Color="#FF484259" Offset ="1"/>
```

```
      <GradientStop Color="#FF4616A 8" Offset ="0.26800000667572021"/>
      <GradientStop Color="#FF22906A " Offset ="0.48199999332427979"/>
      <GradientStop Color="#FF6D316F" Offset ="0.69599997997283936"/>
    </LinearGradientBrush>
  </Grid.Background>
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="496"/>
      <ColumnDefinition Width="144"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
      <RowDefinition Height="83.179"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="111.952"/>
  </Grid.RowDefinitions>
 <Button HorizontalAlignment="Stretch" Margin="0,0,8,8" Width="Auto
  Content="Pulse" Grid.Row="2" Grid.Co lumn="1" VerticalAlignment= "Stretch"
  x:Name="Botón" FontFamily="Verdana" FontSize="24" FontWeight="Light"
  FontStretch="Condensed" FontStyle= "Italic" BorderThickness="3,3,3,3"
  TextAlignment="Center"/>
</Grid>
</UserControl>
```

Listado 3-11

```
//Declaraciones de variables de control
int TotalPortadas = 7;
int contador = 0;
….
private void Siguiente_Portada(object sender, RoutedE ventArgs e)
  {
    contador++;
```

```
  if (contador == TotalPortadas + 1) contador = 1;
  Uri uri = new Uri("Imagenes/portada00" + conta dor.ToString() + ".jpg",
        UriKind.Relative);
  ImageSource logo = new System.Windows.Media. Imaging.BitmapImage(uri);
  Portada.SetValue(Image.SourceProperty, logo);
 }
```

Normalmente, los atributos de imágenes se relacionan con elementos del tipo **Bitmap** representados por la clase **BitmapImage**. Pero en el caso del elemento **Image ("Portada")**, se solicita directamente una propiedad **Source**, que vemos con el nombre de **SourceProperty** —una propiedad de dependencia[4]—, con la que poder generar el bitmap. A su vez, **ImageSource** requiere una ubicación válida para la que el código tenga permiso de acceso (Uri).

Más adelante veremos otros métodos similares, aunque alternativos, de asignar imágenes a controles de la interfaz de usuario.

Ajustando el control para efectos especiales

Supongamos que —al igual que hemos visto en muchas demos o por Internet— no nos resistimos a probar el efecto de reflejo sobre una superficie. Para ello, tendremos que jugar con la imagen dinámica cargada, y la clave del proceso estará en el contenedor. En lugar de 1, tendremos ahora 2 controles **Image**. El segundo, situado a continuación del primero, a poca distancia de su borde inferior.

4. Ver capítulo 6 para una explicación detallada

Para conseguir los efectos de reflejo parcial, tendremos que invertir el sentido de presentación de la imagen y establecer la opacidad (visibilidad) del control a un valor cercano al 0.

Una vez asignadas las propiedades, nuestra ventana puede tener un aspecto inicial similar al de la figura 3-20.



Figura 3-20. Ventanas del editor de posiciones de un elemento (Layout), simple y desplegada

Lo único que falta en el código es rellenar la imagen del reflejo (también de forma dinámica), y recordar que cada nueva imagen será mostrada con las características visuales de su contenedor.

Listado 3-12

```
//Imagen Reflejada
ImageSource logo2 = new System.Windows.Media.Imaging. BitmapImage(uri);
Portada_Copy.SetValue(Image.SourceProperty, logo2);
```

En el código fuente definitivo del evento `Siguiente_Portada`, bastará con añadir un par de líneas para cargar otro Bitmap que pueda ser usado por el control `Portada_Copy` que se encarga del reflejo.

Téngase en cuenta que la misma imagen no puede servir de origen de información para los dos controles.

Conclusión

Hay 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 proveedores de datos. En el capítulo 6 veremos lo concerniente a Animaciones y Transformaciones y en el capítulo 7 dedicado al tratamiento de datos, cómo utilizar Blend para generar interfaces que enlacen con datos de una forma visual.

4

XAML en Silverlight 2.0

Ya hablamos en el primer capítulo del tremendo esfuerzo de simplificación y miniaturización que supone haber reducido todo el potencial presente en el lenguaje XAML de Windows Presentation Foundation hasta conseguir un *runtime* de poco más de 4 Mb y de las experiencias aprendidas en el proceso.

Esto no solo es un trabajo de optimización, sino que requiere la reducción o eliminación de algunas características que no están presentes en la versión de XAML que presenta Silverlight 2 (como los gráficos 3D, por ejemplo). No obstante, el desarrollador acostumbrado a WPF, quizá se sorprenda de la cantidad de elementos disponibles aquí, para lo reducido del complemento.

Por otro lado, en los tres primeros capítulos, casi todo el código XAML que hemos utilizado ha sido generado por Expression Blend o el propio Visual Studio. Nos hemos centrado más en las herramientas y su capacidad de producción de código que en el código en sí.

Vamos, por tanto a revisar el potencial de la versión de XAML disponible para Silverlight 2, dividiendo sus elementos en dos grupos principales: los del propio

lenguaje y su capacidad de expresión para representar elementos de la IU, y en el capítulo siguiente, aquellos que ya están construidos y presentan una funcionalidad encapsulada en forma de control.

El lenguaje XAML

Se trata, en muchos sentidos, de un lenguaje de marcas similar a los estándares utilizados en Internet: HTML, XHTML, etc. Un fichero XAML es, simplemente, un fichero de texto plano con sintaxis XML, y el único requisito formal inicial es que sea un documento XML bien formado[1].

Un documento XAML puede tener cualquier extensión, aunque, por convención, asumimos que `.xaml` es la más descriptiva. XAML, por otra parte, es un lenguaje orientado a objetos (no olvidemos que cada elemento representa una clase definida en las librerías del *runtime*), y usualmente (pero no obligatoriamente) dispone de una clase parcial complementaria, escrita en un lenguaje .NET válido: C#, VB.NET o un lenguaje dinámico, aunque los dos primeros son los predeterminados. Cualquier elemento definido en la parte XAML es reconocido por su clase subyacente.

También es perfectamente posible, y en ocasiones se hace así, crear elementos de la interfaz de usuario utilizando exclusivamente código .NET. No obstante, la

1. Los documentos XML admiten dos niveles de conformidad para garantizar su coherencia. Se dice queun 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 contenidocon un esquema que lo define, sea del tipo XML–Schemas o DTD (Document Type Definition).

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.

Otra cuestión a tener en cuenta 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 pérdidas de calidad por el cambio de tamaño (escalados) en ninguno de los dos sentidos. De eso se encarga el núcleo *Presentation Core* que podemos ver en la figura 4-1, junto al resto del *framework* de Silverlight, sirviendo de base a todos los elementos visuales.

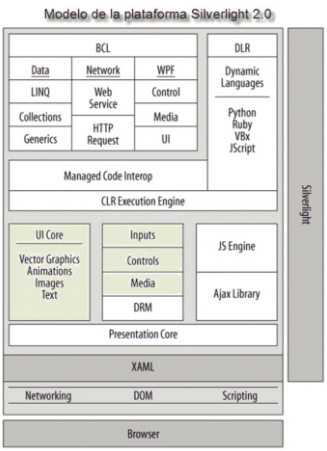


Figura 4-1. Ventanas del editor de posiciones de un elemento (Layout), simple y desplegada

En la lista de ítems disponibles para la construcción de la interfaz de usuario también 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

Si recordamos lo dicho en capítulos anteriores, el fichero **.XAP** que generamos al crear una aplicación Silverlight 2, tiene como contenido principal una DLL cuya entrada inicial 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 **Page** al objeto contenedor de toda la parte visual: **RootVisual**.

Un vistazo al código generado por Visual Studio 2008 para esa clase **Page**, nos mostrará una parte XAML y otra CS (o VB.NET). En la parte XAML, el elemento principal de toda la interfaz es un objeto **UserControl**, con unas pocas propiedades asignadas:

Listado 4-1

```
<UserControl x:Class="SL_C2_1.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White"> </Grid>
</UserControl>
```

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

Un `Grid` definido de esa forma es como una tabla que contiene una sola celda que ocupa todo su espacio disponible, al que podemos hacer referencia en el código subyacente por el nombre asignado en su atributo `x:Name`, y que —tal y como está— tiene fondo de color blanco, por lo que no resulta visible aunque lancemos la aplicación (obtendremos una ventana del navegador aparentemente vacía).

Espacios de nombres en XAML

En XAML, todos los elementos se corresponden con una clase de .NET incluida en el *CoreCLR*. Y los atributos de éstos, expresados en sintaxis XML, tienen su contrapartida (*mapean*, decimos a veces) como 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, especialmente para referirnos a objetos de negocio.

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 un secuencia única de caracteres (debido a esto, se utilizan recursos

URI muchas veces) que se asocian con una especificación o una librería y sirve para definir sin ambigüedad qué elementos están permitidos en un fichero XML (XAML en este caso).

En la definición del control principal, notamos la presencia de dos declaraciones de espacios de nombres (`xmlns`): el predeterminado, que apunta al núcleo de presentación y uno secundario (prefijo `x`) que define extensiones de marcado y que utilizamos para nombrar elementos, asignar claves (*Keys*) que permitan referenciar posteriormente estilos u otros recursos programables, etc.

El IDE de visual Studio ha evolucionado para incluir 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, y si hemos compilado la aplicación, incluirá la DLL generada en nuestro proyecto (ver figura 4-2).

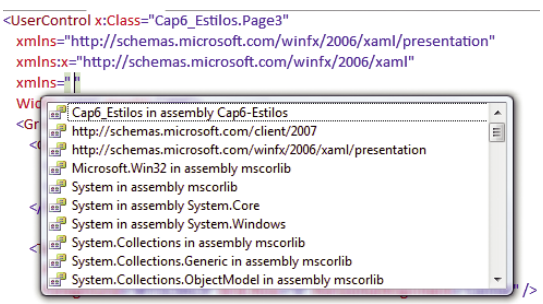


Figura 4-2. Intellisense en la asignación de espacios de nombresdesde Visual Studio 2008

El inevitable "Hola Mundo"

Según esto, y si queremos seguir la tradición de escribir un "Hola Mundo" como primer programa, al estilo que originó **Charles Petzold**, solo necesitaremos un elemento capaz de mostrar algún texto para tener esa primera aplicación. Bastará un vistazo a la Ventana de herramientas de Visual Studio para detectar dos posibles candidatos (ver figura 4-3):

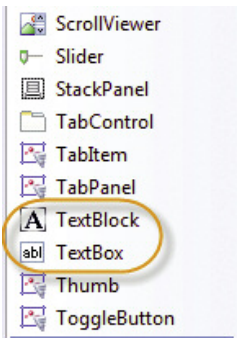


Figura 4-3. Fragmento de la Ventana de herramientas.

Cualquiera de los dos nos sirve, pero no es una mala práctica que nos acostumbremos a utilizar cada cosa para el propósito para el que fue creada. Ya que el texto que deseamos es de "solo-salida" (el usuario no interactuará cambiándolo), el más apropiado es el `TextBlock`.

Introducimos la siguiente línea dentro del elemento `<Grid>`:

```
<TextBlock Text="Hola desde Silverlight 2.0" FontFa mily="Arial" FontSize="24"
           VerticalAlignment="Center" HorizontalAlignment="Center"/>
```

La compilación producirá el fichero `.xap` correspondiente, el servidor de IIS local asignará un puerto para

la ejecución, y el `.xap` será cargado por la página que hayamos establecido como inicial. El resultado… evidente (nos podemos ahorrar un gráfico).

Layout

Una revisión del código anterior nos lleva a una primera diferenciación entre los elementos XAML de Silverlight 2: 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 solo admiten un elemento contenido (aunque este pueda ser, a su vez, 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.

Elementos contenedores

Un control Silverlight solo puede albergar un elemento. Para conseguir interfaces con múltiples elementos, existen los contenedores (también llamados *layouts*). Se trata de los objetos que heredan de `System.Windows.Controls.Panel`.

A continuación mostramos la jerarquía de elementos que heredan de `Panel`, tal y como la muestra la herramienta `Reflector`. Los 3 contenedores disponibles son **Grid**, **Canvas** y **StackPanel** (con Silverlight Toolkit, se dispone de 2 más: `WrapPanel` y `DockPanel`):

Si analizamos un poco más el elemento `TextBlock` utilizado, veremos que dispone de un número de atributos programables, relativos al texto a mostrar, a su diseño visual, y a sus relaciones con el entorno. Este modelo

se repite en la inmensa mayoría de los controles y elementos visuales, y hay algunos de ellos que varían dependiendo del contexto del control (de su contenedor).

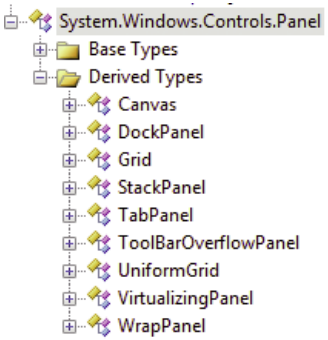


Figura 4-4. Jerarquía de objetos derivados de Panel.

Recordemos que podemos abrir cualquier fichero XAML en Blend. Si seleccionamos el menú contextual del fichero `Page.xaml`, tenemos la opción de abrirlo como aparece en la figura 4-5.

Avancemos algo más. Supongamos que queremos también un pequeño título, un fondo distinto del blanco y que nuestro letrero se muestre en uno de esos bonitos rectángulos con los bordes redondeados. En este caso no hay filas y columnas definidas pero bastaría definir dos filas para poder separar título de contenido. Convendría igualmente, hacer que el título estuviera en una fila inicial de tamaño fijo, y el texto, ocupando la otra fila, centrado. El código del listado 4-2 se encarga de ello.

Listado 4-2

```
<Grid.RowDefinitions>
        <RowDefinition Height="56.1"/>
        <RowDefinition Height="*"/>
</Grid.RowDefinitions>

<TextBlock Text="Primera aplicación Silverlight 2.0" FontFamily="Arial"
         FontSize="18" VerticalAlignment="Center" Horizontal Alignment="Center"
        Grid.Row="0"/>
<Border CornerRadius="25" Margin="40,87.9,40,84" Grid.Row="1" BorderThickness="5,5,5,5"
   Background="Blue" BorderBrush="Yellow" Horizontal Alignment="Stretch"
   VerticalAlignment="Stre tch"/>
<TextBlock Text="Hola desde Silverlight 2.0" FontFamily ="A rial" FontSize="24"
  VerticalAlignment= "Center" HorizontalA lignment="Center" Grid. Row="1"
  Foreground="#FFE4EC12"/>
```

Propiedades adjuntas

Adviértase la presencia de atributos que representan propiedades relativas al contenedor de ese elemento. Se trata del concepto de Propiedades Adjuntas (*Attached Properties*). En el elemento contenido se establece una propiedad que, realmente, compete al contenedor. Concretamente, `Grid.Row` establece en qué fila se mostrará el elemento (siempre comenzando por 0), y de igual forma podemos asignar atributos `Grid.Column`, en caso de haber definido columnas. El código es bastante auto-explicativo.

Y un apunte sobre el efecto de biselado: se consigue con la propiedad `CornerRadius` del objeto `Border`, y el mismo

efecto se pueden obtener igualmente en otros elementos (`Rectangle`, etc.).

El resultado gráfico (esta vez sí), debiera ser el siguiente:



Figura 4-6. Salida gráfica del código anterior.

Las rejillas (*grids*) se utilizan por su similitud con las tablas 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:

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

Además, estos modos siempre pueden mezclarse para conseguir la combinación para un escenario concreto, y de forma muy similar a como sucede en HTML con las tablas, 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 complejo, y lo mejor para su manejo es utilizar el diseñador de estados Visual State Manager, por lo que remitimos al lector interesado a la documentación del producto.

Los otros contenedores: 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 para ciertas áreas por su capacidad de agrupar sus contenidos en una sola línea que puede ser orientada vertical u horizontalmente.

La gran diferencia de **Canvas** con respecto a **Grid** es que sus elementos contenidos se pueden ubicar en cualquier posición que se desee (incluyendo solapamientos), como ya vimos en el capítulo 3. Esto se consigue mediante posicionamiento absoluto, en forma similar a la propiedad CSS equivalente para HTML. De hecho, el código paralelo para la salida anterior sería muy similar, excepto que tendríamos que decirle exactamente dónde debe situarse cada uno de los dos literales (y el objeto **Border**), cosa que realizamos mediante las propiedades **Canvas.Top** y **Canvas.Left**, de cada control:

Listado 4-3

```
<Canvas x:Name="LayoutRoot" Background="Azure">
  <TextBlock Text="Primera aplicación Silverlight 2.0" FontFamily="Arial"
FontSize="18" Verti calAlignment="Center" HorizontalAlignment= "Right" Can-
vas.Left="63.844" Canvas.Top= "16"/>
   <Border CornerRadius="25" Margin="0,0,0,0" Border Thickness="5,5,5,5"
       Background="Blue" Border Brush="Yellow" HorizontalAlignment="Stretch"
VerticalAlignment="Stretch" Canvas.Left="16"  Canvas.Top="128" Width="360"
Height="66"/>
    <TextBlock Text="Hola desde Silverlight 2.0" Font Family="Arial" FontSize="24"
VerticalAlignment= "Center" HorizontalA lignment="Center"
Foreground="#FFE4EC12" Canvas.Top="148.402" Canvas .Left="63.844"/>
</Canvas>
```

NOTA. Recuerde que en Expression Blend 2.0, la ventana de diseño muestra dos modos de trabajo (modo Grid y modo Canvas), entre los que conmutamos pulsando sobre el icono de la parte superior izquierda de esa ventana.

Con este código, obtendríamos una salida idéntica a la anterior, y el posicionamiento pasa a ser absoluto, pudiendo precisar un gran número de decimales para la ubicación exacta de cada elemento.

Al objeto de ganar en flexibilidad a la hora de organizar los elementos visuales cuando trabajamos con contenedores **Canvas**, un objeto **Canvas** puede estar contenido dentro de otro y ser —a su vez— contenedor de un conjunto diverso de controles.

NOTA. En situaciones reales, cuando es necesario averiguar el tamaño actual (altura y anchura) de un elemento, las propiedades Width y Height no son de gran ayuda, ya que solamente expresan la altura y anchura deseadas, pero por razones de cambio de tamaño de otros elementos es muy posible que esos valores no coincidan con los actuales. Los valores requeridos son suministrados por las propiedades ActualHeight y ActualWidth.

Esto puede verse en éste sencillo ejemplo que nos suministra la documentación oficial:

Listado 4-4

```
<Canvas Wid[0" 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>
```

Que produce el resultado de la figura 4-7.

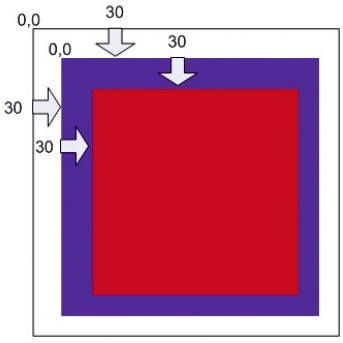


Figura 4-7. Objeto Canvas anidado dentro de otro que contiene un rectángulo (Gráfico de MSDN).

En situaciones de diseños complejos, lo mejor es jugar con ambos mundos, ya que pueden convivir perfectamente (un `Grid` puede contener objetos `Canvas` en cualquiera 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 contenida ocupe toda su superficie. En el caso del `Canvas` (Fila 1, Columna 0) su contenido se ubica de forma relativa al contenedor directo (el propio `Canvas`, y no al `Grid` que hace de contenedor general (el código corresponde al del listado 4-5).

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 listado 4-5 se corresponde con la figura 4-8.

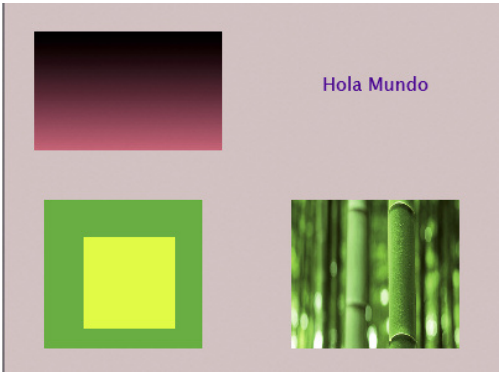


Figura 4-8. Salida del código anterior mostrando un Grid con cuatro celdas y un objeto distinto en cada una de ellas.

Listado 4-5

```
<UserControl x:Class="SL_C2_1.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/ xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/200 6/xaml"
    Width="400" Height="300"
    xmlns:d="http://schemas.microsoft.com/expression /blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/mar kup-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>

<Canvas Canvas.Left="30" Background="#FF0FA22 B"
   Canvas.Top="30" HorizontalAlignment= "Stretch" Margin="32,16,40,20"
   Grid. Row="1" Grid.Column="0" >
 <Rectangle Canvas.Left="32" Canvas.Top="30" Fill="#FFFFF500"
             Width="74" Height="74" />
</Canvas>

<TextBlock Margin="0,0,0,0" Text="Hola Mundo" TextWrapping="Wrap"
 Grid.Column="1" Fore ground="#FF3512A E" Width="Auto" Height="33"
   HorizontalAlignment="Center"/>

<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.RowDefinitions>
   <RowDefinition Height="0.48*"/>
   <RowDefinition Height="0.52*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="A uto"/>
   <ColumnDefinition Width="0.5*"/>
</Grid.ColumnDefinitions>
```

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

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 manera que ocupe todo el espacio disponible en el control `Image` que lo carga.

Márgenes y alineación

El control de la posición de los elementos que no se ubican mediante posicionamiento absoluto, se realiza mediante las propiedades `Margin`, `HorizontalAlignment` y `Vertical Alignment`, 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 estriba en 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).

Elementos de dibujo: Shapes y Geometries

Obviamente, ni los elementos vistos hasta ahora, ni los controles que veremos más adelante, son adecuados para la creación de dibujos complejos de tipo vectorial. Esa es una parte funcional muy importante de WPF y un subconjunto de ella está presente en Silverlight 2.0. En el capítulo anterior, vimos cómo Blend divide esos conceptos: lo que se "dibuja" en una sesión de usuario y no tiene forma predeterminada, y lo que cae dentro del apartado del control, que sí lo tiene.

También veíamos que existen dos tipos de elementos para conseguir estos objetivos: las formas, que heredan de `Shape` (`Ellipse`, `Rectangle` y `Line`, `Path`, `Polygon` y `Polyline`) y las geometrías (`Geometries`), objetos de sufijo `Geometry` que no deben confundirse con los anteriores (existe un `RectangleGeometry`, un `EllipseGeometry`, un `PathGeometry`, etc). Ambas categorías tienen cosas en común y varias de ellas pueden producir efectos muy similares, pero se diferencian en varios aspectos.

Diferencias entre formas y geometrías

Las `Shapes` son objetos que heredan `Shape` (y por tanto, de `UIElement`), mientras que las geometrías no. Debido a esto,

las formas pueden interpretarse visualmente a sí mismas (*render*), mientras que las geometrías no. Esto significa que disponen de propiedades como `Opacity`, `OpacityMask` y otras cualidades gráficas de las que carecen las geometrías. En el otro lado de la balanza, 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 la interpretación visual (por ejemplo, un objeto `PathGeometry` —del tipo `Geometry`—, puede definir todos los puntos de su contenido en su atributo data, pero usará un objeto `Path` —tipo `Shape`— para que lo dibuje).

Además de poder dibujarse a sí mismas, y por su posición en la jerarquía de clases, las formas tienen otras ventajas programáticas, como la capacidad de poder utilizar cualquier contenedor válido (igual que cualquier `UIElement`) y, sobre todo, la posibilidad de soportar los mismos eventos que sus homólogos.

Todos los objetos de tipo `Shape` heredan de la clase `System.Windows.Shapes.Shape` y el diagrama de la jerarquía completa de clases es el siguiente:

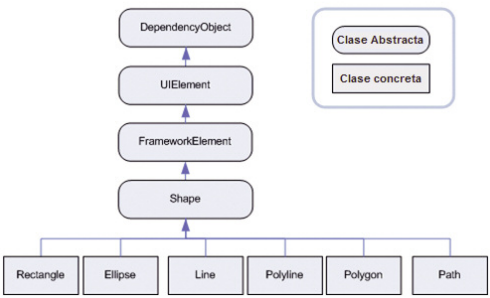


Figura 4-9. Diagrama jerárquico de clases de los objetos Shape.

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

Formas (Shapes)

En el apartado de las formas distinguimos los siguientes objetos:

- `Ellipse`
- `Line`
- `Path`
- `Polygon`
- `Polyline`
- `Rectangle`

Con la adecuada combinación de los objetos `Shape`, podemos construir cualquier dibujo que nos propongamos, siendo especialmente útil para las formas irregulares el objeto `Path` que, literalmente, puede expresar cualquier combinación visual.

Todos ellos disponen de un conjunto de propiedades comunes entre las que destacamos las siguientes:

- `Stroke`: que describe cómo se dibuja el borde de la forma.
- `StrokeThickness`: que describe el grosor del borde de la forma.
- `Fill`: que describe como se rellena el interior de las formas.

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

Las formas suelen estar ubicadas dentro de objetos **Canvas**, ya que el posicionamiento absoluto es, a menudo, necesario para el dibujo, pero no hay problema para que se encuentren incrustadas en otros contenedores válidos. Ya hemos visto antes el código para dibujar un rectángulo según estos principios, y podemos aplicarlo también al resto de elementos. Por ejemplo para obtener una elipse sin relleno, (dentro de un **Grid**), nos bastaría con esto:

Listado 4-6

```
<Grid x:Name="LayoutRoot" Background="White" >
     <Ellipse Margin="128,104,208,224" Fill="#FFFFFFFF" Stroke="#FF000000"
     StrokeThickness="3"/>
</Grid>
```

Y si queremos ubicar varios objetos **Shape** en un contenedor tipo **StackPanel**, para que los presente verticalmente repartiendo el espacio, el problema no es muy complicado:

Listado 4-7

```
<Grid x:Name="LayoutRoot" Background="White" ShowGrid Lines="True" >
   <Grid.ColumnDefinitions>
           <ColumnDefinition Width="0.5*"/>
```

```
          <ColumnDefinition Width="0.5*"/>
   </Grid.ColumnDefinitions>
   <StackPanel Margin="8,8,8,8"Orientation="Vertical">
     <Ellipse Fill="#FF354A 91" Stroke=" #FF000000" StrokeThickness="3"
             Height="150" Width="275" />
    <Ellipse Fill="#FF9F5913" Stroke="#FF00 0000" StrokeThickness="3"
         Width="100" Height="100" />
    <Ellipse Fill="#FF88C634" Stroke="#FF0 00000" StrokeThickness="3"
         Width="150" Height="200" />
  </StackPanel>
</Grid>
```

Y tendríamos una salida en el navegador similar a la siguiente:

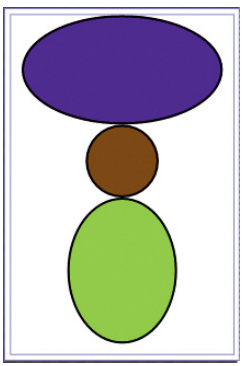


Figura 4-10. Disposición final de los 3 objetos elipse del código anterior.

También podemos utilizar la propiedad **`Stretch`** para determinar la manera en que la forma aprovecha el espacio de que disponga. Sus valores pueden ser:

- **None**: Ninguno
- **Fill**: Relleno completo del contenedor modificando las propiedades **Width** y **Height**.
- **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 este ejemplo, donde 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 4-8

```
<Grid x:Name="LayoutRoot" Background="White" ShowGrid Lines="True" >
   <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
   </Grid.RowDefinitions>

   <Ellipse Grid.Row="0" Fill="#FF354A 91" Stroke="#FF000000"
         StrokeThickness="3" Stretch="Fill" />
   <Ellipse Grid.Row="1" Fill="#FF9F5913" Stroke="#FF000000"
          StrokeThickness="3" Stretch="Uniform" />
   <Ellipse Grid.Row="2" Fill="#FF88C634" Stroke="#FF000000"
          StrokeThickness="3" Stretch="UniformToFill" />
</Grid>
```

La diferencia resultante puede apreciarse perfectamente incluso en la ventana de diseño de Visual Studio (figura 4-11).

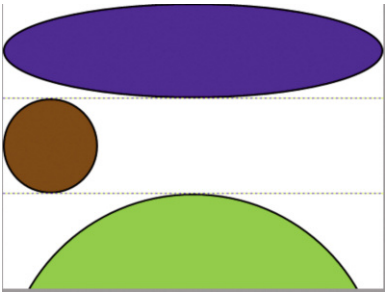


Figura 4-11. Las 3 elipses mostrando el efecto de aplicar su atributo Stretch.

Como vemos en el gráfico, la elipse inicial establece un crecimiento no uniforme hasta alcanzar los límites de la fila; como ésta es un rectángulo, obtenemos la elipse inscrita correspondiente. La elipse del centro, ha crecido uniformemente en ambas propiedades, lo que ha resultado en un crecimiento radial a partir de un foco único, convirtiéndose en un círculo. Y, por último, la tercera, es como una mezcla de las dos anteriores: el crecimiento es uniforme (radial), pero en la figura resultante solo se muestra la parte que cabe en su contenedor.

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 (`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`), y algunas más. 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 el ejemplo adjunto. La otra opción disponible (`NonZero`), lo hubiera rellenado como el resto de secciones.

Listado 4-9

```
<Canvas x:Name="LayoutRoot" Background="White">
   <Line Stroke="Maroon" StrokeThickness="5"
      X1="10" Y1="10" X2="220" Y2="100"></Line>
   <Line Stroke="Red" StrokeThickness="5"
       X1="0" Y1="0" X2="110" Y2="100" Canvas.Left="25" Canvas.Top="100"></Line>
   <Line Stroke="Blue" StrokeThickness="5"
     X1="25" Y1="100" X2="110" Y2="100"></Line>
   <Polyline Stroke="Navy" StrokeThickness="5"
```

```
     Points="265,50 400,200 375,100 500,150"> </Polyline>
   <Polyline Stroke="Navy" StrokeThickness="5"
     Points="300,300 415,370 475,200 500,250 300,300"></Polyline>
    <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>
```

La salida del código anterior es la que se muestra en la figura 4-12:

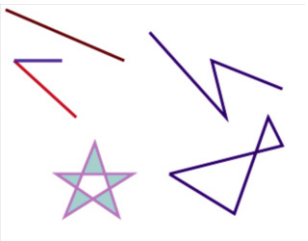


Figura 4-12. Ejemplo de líneas, poli-líneas y polígono.

Hemos dejado para el final el objeto `Path` por ser el más complejo, pero también el más completo de todos, ya que —técnicamente hablando— un `Path` puede dibujar cualquiera de los objetos anteriores más otros (como la construcción de curvas) que, aquellos, no tienen posibilidad de hacer.

Geometrías y el objeto Path

El funcionamiento del objeto `Path` es distinto a lo anterior. Se basa en que contiene una propiedad `Data` que permite

definir cualquier elemento gráfico por complejo que sea. En realidad, la propiedad `Data` acepta un objeto `Geometry` que es donde se define la forma a dibujar. De forma similar, los objetos `UIElement`, aceptan elementos `Geometry` en su propiedad `Clip`, para definir recortes.
Pero un objeto `Geometry` no puede usarse directamente, porque es una clase abstracta. Debemos utilizar alguna de sus clases derivadas o heredar nosotros de `Geometry`. Las clases derivadas son las siguientes:

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

Según esto, la forma de dibujar un rectángulo mediante un objeto `Path` que use un elemento `Geometry` será similar a esta:

Listado 4-10

```
<Path Fill="Yellow" Stroke="Blue">
  <Path.Data>
```

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

Y la mejor manera de combinar dos geometrías, utilizando un elemento **Geometry Group**:

Listado 4-11

```
<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>
```

De igual forma, podemos usar objetos **UIElement** para aplicar un objeto **Geometry** a su propiedad **Clip**, como mecanismo de recorte. Por ejemplo, para recortar una imagen, utilizando un objeto **Image** y esta técnica, podríamos construir lo siguiente:

Listado 4-12

```
<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>
```

Que nos ofrecería el resultado visual de la figura 4-13. 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**.



Figura 4-13. Imagen con recorte utilizando un objeto Geometry

PathGeometry

Es el objeto de dibujo por excelencia. 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 *E*xpression Blend, Expression Design o Adobe Ilustrator, pueden almacenar sus datos en formato XAML y permitirnos usarlos después de forma muy sencilla. Al final de este capítulo, comentaremos cómo generar estos ficheros más complejos, pero vamos a ver, al menos, 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 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 esta 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 4-13

```
<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>
```

Obteniendo esta salida en la Ventana de diseño.



Figura 4-14. 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 arco que se va a dibujar, 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 obtener 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 4-14

```
<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>
```

Produciría un único arco con la forma que se aprecia en la figura 4-15a.



Figuras 4-15a y 15 b. 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.

Hay dos propiedades que se han asumido de forma implícita para seleccionar (en este ejemplo) 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 por parte del sistema el valor `ArcSegment.IsLargeArc` predeterminado (o sea, falso), con lo que se seleccionó el arco más corto. Incluso así, hay dos formas de dibujar ese arco entre los 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 4-15b.

Curvas Bézier

Se trata de una curva paramétrica estudiada en Análisis Numérico, muy importante en computación gráfica y sus campos relacionados[2]. 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 y uno final y dos puntos de control, podemos construir una variedad enorme de curvas suaves (continuas o derivables). La imagen de la figura 4-16 da una idea —siquiera intuitiva— del proceso a seguir en la construcción de este tipo de curvas.

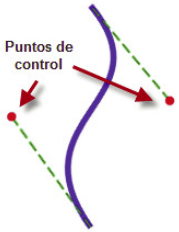


Figura 4-16. Curva Bézier con sus puntos de control.

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 (**`BezierSeg ment.Point3`**). En el siguiente ejemplo (listado 4-15), 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 4-17.

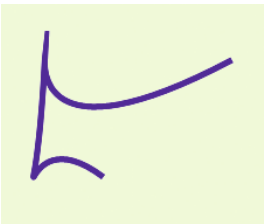


Figura 4-17. Dos curvas Bézier concatenadas.

Listado 4-15

```
<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>
```

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, como apuntábamos antes, herramientas pensadas para ello, como Expression Design, o cualquier otra capaz de exportar sus datos a formato XAML, como Adobe Illustrator.

NOTA. De hecho, Expression Design tiene la capacidad de leer ficheros generados por Adobe Illustrator con extensión -.ai-, que, luego, pueden ser exportados fácilmente a XAML.

Precisamente con el propósito de manejar estos elementos cuando la complejidad de los gráficos crece, se ha creado un "mini-lenguaje" especial para los objetos **`Path Geometry`**, que permite abreviar la descripción de los

gráficos pensando en las herramientas citadas. 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 por 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. Pues bien, Word (u otro programa de Office 2007) 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.

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 descomprimimos, veremos un montón de carpetas con los recursos y datos que contiene.

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. Solo hay que hacer dos pequeños retoques: eliminar el elemento raíz (llamado `FixedPage`), 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 4-18, 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 minilenguaje.

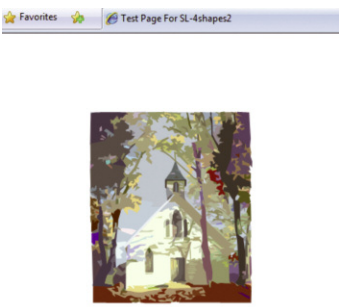


Figura 4-18. Fichero *j0149407.wmf* de la Galería de Windows, convertido a XAML y mostrado por Silverlight.

Recortes mediante geometrías

Antes de concluir este capítulo, un recordatorio respecto a los recortes (*Clipping*). Hemos visto un ejemplo de cómo recortar una imagen utilizando la propiedad `Clip` de que disponen todos los elementos, y donde usábamos un objeto `EllipseGeometry` para obtener ese efecto.

Lógicamente, la máxima potencia se obtiene cuando se utilizan objetos `PathGeometry` para definir el recorte, ya que este puede tener cualquier forma imaginable (que seamos capaces de dibujar) y, por lo tanto, permitir que la parte visual del elemento adopte formas muy complejas, tal y como sucede con algunas aplicaciones populares de reproducción de vídeo y audio, o software utilizado en redes sociales.

Vemos, pues, que no hay límites a lo que se puede representar con imágenes vectoriales en Silverlight 2, salvo las características no soportadas, como la tridimensionalidad. En el próximo capítulo, estudiaremos los controles como mecanismos ya construidos y preparados para reproducir interfaces de usuario más propias de aplicaciones de gestión y control.

5

Controles

Hasta ahora hemos visto los elementos XAML más simples y también hemos indicado cómo gestionar parte de la funcionalidad que presentan al usuario. En el capítulo anterior, también vimos la creación de nuevos elementos visuales mediante las posibilidades de dibujo que ofrecen los conjuntos de elementos Shapes y Geometries.

Si bien hemos usado controles sencillos, como `TextBlocks` o `Buttons`, es momento de revisar más a fondo el apartado de los controles. Son componentes terminados (pero abiertos), que generalmente contienen una estructura visual más elaborada, tanto en su comportamiento gráfico, como en el funcional: eventos programables, propiedades, etc.

La creación de interfaces de usuario modernas asume la existencia de estos controles a los que todos estamos acostumbrados y de los que conocemos su propósito (gráfico y operativo). Cajas de texto (`TextBoxes`), cuadros combinados (`ComboBoxes`), botones de opción (`OptionButtons`), `TreeViews` y rejillas de datos (`DataGrids`) son elementos típicos de estas interfaces y Silverlight no podía obviar una hecho así.

Con la versión 2.0 del producto, se introduce un conjunto muy completo de ellos, que se encuentra en constante crecimiento. Existen planes de continuar la creación de controles para Silverlight 2 y además, compañías de terceros han iniciado ya la creación de sus propias *suites* de controles[1] para esta plataforma, y algunas, anuncian ya juegos completos, que semejan y, muchas veces, mejoran sus contrapartidas disponibles para Windows.

Controles disponibles en Silverlight 2.0

En el manejo de los dos IDE que utilizamos, habrá observado el lector que podemos acceder a listas de controles disponibles tanto en Visual Studio 2008 (Caja de herramientas), como en Expression Blend 2.0 (Ventana Assets, en la parte inferior izquierda de este IDE), aunque hay presencias y ausencias en cada uno de ellos debidas, simplemente, al hecho de que algunos controles se encuentran en las librerías del SDK de Silverlight, y no forman parte del *runtime*, como sucede con `Datagrid`, `Calendar`, `DatePicker` y otros..

Hay elementos visuales que Expression Blend agrupa con los anteriores, sin que realmente se puedan considerar controles, como los `Glyphs` o los `PopUp`, que heredan de `FrameworkElement`, pero que aparecen vinculados con la interfaz de usuario, por tener un componente visual innegable. La figura 5-1 muestra la lista de controles disponible en Visual Studio 2008 SP1.

---

1. 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, 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.

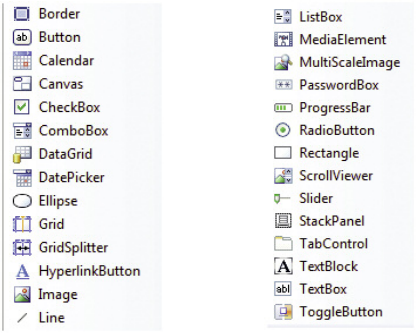


Figura 5-1. Conjunto de controles disponibles en Silverlight 2.0, presente en la Caja de herramientas de Visual Studio 2008.

Lo positivo de esta aproximación es que todos ellos tienen una sintaxis homogénea y una manera afín de declarar sus atributos y contenidos. De forma que vamos a dividirlos por grupos funcionales y veremos su comportamiento a través de ejemplos que agrupen varios de ellos en una misma demo.

Controles que muestran/admiten texto como función primaria

Hay unos cuantos… `TextBox`, `TextBlock`, `PasswordBox`, `CheckBox`, `RadioButton`, `DatePicker`, `HyperLinkButton`, `ListBox/ ListBoxItem` y `ComboBox/ComboBoxItem` (aunque estos dos últimos pueden mostrar cualquier cosa). Es cierto que otros controles también pueden mostrar texto, pero esa no es su función primaria (botones, `TabControl`, etc.).

También existen otros elementos XAML que no pueden considerarse controles estrictamente hablando, pero que intervienen en procesos relacionados con el manejo

y visualización de texto, como **LineBreak** (salto de línea), **Run** (renglón), **Inline** (e **Inline Collection**), **Glyphs**, y **FontSource**. Todos estos pertenecen al espacio de nombres **System. Windows.Documents.**

NOTA. Obviamos el caso del DataGrid, ya que, por ser uno de los más importantes en el tratamiento de datos, lo trataremos aparte, en el capítulo 7.

Estrictamente hablando, solo los tres 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 (la documentación se refiere a ellos como *headered controls*), o que muestren conjuntos discretos y bien definidos de texto (como **DatePicker**).

Veamos un ejemplo en el que ponemos en funcionamiento un grupo de ellos, creando un formulario de entrada de datos, al que —por completar el contexto— le añadimos un botón (sin funcionalidad ninguna). El código completo puede verlo en el listado 5.1.

El resultado visual es el que vemos en la figura 5-2. Observe el lector la presencia de un espacio de nombres que hemos añadido manualmente a la declaración del **UserControl: System.Windows.Controls.** Aunque el IDE de Visual Studio muestra todos los controles, los de fecha y algunos otros se encuentran en esas librerías que no se referencian por defecto en el proyecto.

Listado 5-1

```
<UserControl x:Class="Cap5_1.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:ctlEx="clr-namespace:System.Windows.
          Controls;assembly=System.Windows.Controls"
Width="420" Height="320">
<Border BorderThickness="12" BorderBrush="Fire brick" CornerRadius="12"
           Width="420" Height="300">
   <Grid x:Name="LayoutRoot" Background="Gainsboro">
     <Grid.RowDefinitions>
       <RowDefinition Height="50" />
       <RowDefinition Height="30" />
       <RowDefinition Height="75" />
       <RowDefinition Height="40" />
       <RowDefinition Height="50" />
     </Grid.RowDefinitions>
     <TextBlock x:Name="txbNombre" Horizontal Alignment="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" HorizontalA lignment="Left"
           VerticalAlignment="Center" Margin="20, 0,0,0" Width="261.695"
           Content="Coche propio" IsThreeState="False"
           Grid. Row="1"/>
     <TextBlock x:Name="txbPWD" Horizontal Alignment="Left"
     VerticalAlignment= "Center" Height="24" Width="70" Text="Contraseña:"
     TextWrapping= "Wrap" Margin="135,0,0,0" Grid.Row="1" />
     <PasswordBox Grid.Row="1" Width="90" Height="20"
          VerticalAlignment= "Center" Margin="120,-10,0,0"
          Background= "Aqua" Foreground="Black" />
```

```
      <ctlEx:DatePicker Text="Seleccione Fecha de entrada" Grid.Row="2"
         Horizontal Alignment="Left" VerticalAlignment ="Center"
         Margin="20,0,0,0" Width="160" />
      <ListBox Grid.Row="2" Height="64" Width= "170" Margin="200,0,0,0">
            <ListBoxItem>
                <TextBlock>Analista Senior</Text Block>
            </ListBoxItem>
            <ListBoxItem>
              <TextBlock>Programador Junior</Text Block>
                 </ListBoxItem>
                 <ListBoxItem>
            <TextBlock>Especialista en IT</Text Block>
                 </ListBoxItem>
               </ListBox>
               <Border BorderBrush="Firebrick" BorderThickness="2"
                     CornerRadius="7"
               Grid.Row="3" Margin="20,10,20,0" />
            <RadioButton Grid.Row="3" Content= "Soltero" Margin="50,15,0,0">
                </RadioButton>
            <RadioButton Grid.Row="3" Content= "Casado" Margin="170,15,0,0">
                                              </RadioButton>
            <RadioButton Grid.Row="3" Content= "Viudo" Margin="280,15,0,0">
                                              </RadioButton>
            <HyperlinkButton Content="Pulse aquí para visitar dotNetMania"
                NavigateUri="http://www.dotnet mania.com"
                 TargetName="_blank" Margin="20,15,0,0"
                 Grid.Row="4" Height="12" Ver ticalAlignment="Center" />
      <Button Content="Enviar" Grid.Row="4" Width="100"
                HorizontalAlignment ="Right" Margin="0,15,20,0" />
        </Grid>
    </Border>
</UserControl>
```

NOTA. Hay otra opción más sencilla que consiste en arrastrar el control desde la Caja de herramientas hasta el editor de código, con lo que éste hace esa labor por nosotros asignando al espacio de nombres "my" de forma predeterminada.



Figura 5-2. Primer ejemplo de controles.

Peculiaridades del ListBox y los elementos colectivos

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 (como `ComboBox`), disponen de funcionalidades que eran difíciles de implementar en aquellos. En el caso del `ListBox`, los contenidos son elementos `ListBoxItem`, que al heredar de `ContentControl`, pueden albergar prácticamente cualquier cosa, así que las posibilidades son innumerables (En el `ComboBox`, cada elemento hijo es del tipo `ComboBoxItem`, y su forma de programación muy parecida).

NOTA. Recuerde que la propiedad `Content` de un `ListBoxItem` (o `ComboBoxItem`) 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 unos populares *bloggers* y también su nombre. Ambos elementos (`Image` y `TextBlock`) se han ubicado en sendos controles `StackPanel` con orientación horizontal (listado 5-2).

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

El resultado en la Ventana de Diseño de VS2008 sería el de la figura 5-3:



Figura 5-3. Salida del listado 5-2 con un ListBox personalizado.

Listado 5-2

```
<UserControl x:Class="Cap5_Bloggers.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/ presentation"
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Width="300" Height="230">
  <Grid x:Name="LayoutRoot" Background="Gainsboro" ShowGridLines="True">
    <Grid.RowDefinitions>
     <RowDefinition Height="20" />
     <RowDefinition Height="210" />
    </Grid.RowDefinitions>
  <TextBlock Text="Bloggers" Grid.Row="0" HorizontalAlignment="Center"
         VerticalAlignment="Center" FontSize="21"/>
  <ListBox Grid.Row="1" Width="240" Height ="160" Background="AliceBlue" >
    <ListBoxItem>
      <StackPanel Orientation="Hori zontal">
       <Image Source="Fotos  /LMB.png" />
       <TextBlock VerticalAlign ment="Center" Margin="7">
             Luis Miguel Blanco</TextBlock>
                       </StackPanel>
    </ListBoxItem>
    <ListBoxItem>
       <StackPanel Orientation="Horizon tal">
         <Image Source="Fotos/Alarcon.png"  Width="48" />
         <TextBlock VerticalAlign ment="Center" Margin="7">
             José Manuel Alarcón</TextBlock>
     </StackPanel>
        </ListBoxItem>
        <ListBoxItem>
     <StackPanel Orientation="Horizon tal">
         <Image Source="Fotos/Dino.png" Width= "48"/>
        <TextBlock VerticalAlignment ="Center" Margin="7">
            Dino Esposito</TextBlock>
     </StackPanel>
  </ListBoxItem>
```

```
    </ListBox>
  </Grid>
</UserControl>
```

Controles para ayuda de la experiencia visual (de uso directo)

Dentro de ese grupo, muchos de ellos son realmente controles complementarios de otros, a los que sirven de soporte, y que se usan para la construcción de controles personalizados. Visitamos primero algunos de los usuales en el mundo del desarrollo con Windows Forms que tienen equivalente aquí.

Son los siguientes: **GridSplitter**, **ScrollBar**, **Slider**, **TabControl** y **ProgressBar**. La funcionalidad de cada uno queda resumida en la tabla 1.

Nombre	Funcionalidad
GridSplitter	También considerado como un sub-control ya que su funcionalidad depende del objeto Grid que lo aloja. Permite redistribuir el espacio entre filas y columnas de un control Grid.
ScrollViewer	Clase que representa de forma más genérica un área desplazable que contenga otros elementos visibles.
Slider	Representa un control que permite al usuario seleccionar un valor entre un rango controlado, mediante el desplazamiento de un marcador entre los límites inicial y final.
TabControl	Control que maneja solapas (TabItems) representando cada uno de ellos una superficie contenedora de otros controles.
Progress	Bar Consta de un rectángulo con un indicador que puede ir mostrando el progreso de una operación.

Tabla 5-1. Controles para ayuda de la experiencia visual.

Vamos a crear otro ejemplo para ver la operativa de este grupo entero de controles (excepto **ProgressBar**). Para ello, creamos una página con un **Grid** con dos filas y dos columnas. En la primera fila, colocaremos un **ScrollViewer** (columna 0), que ocupe dos columnas. En la segunda fila, situaremos cuatro **Sliders** con un elemento **Rectangle** cuyo color de fondo se modificará según los valores ARGB que van proporcionado cada uno de los **Sliders**. En la última celda, situamos un **TabControl** con 3 solapas (**TabItems**), cada uno conteniendo una información diferente (ver la figura 5-4 con el resultado en ejecución para tener una idea de conjunto).

Listado 5-3

```
<UserControl xmlns:my="clr- namespace:System.Windows .Controls;
  assembly=System.Windows.Controls.Extended" x:Class=
        "SL_Controles1.Exp_Visual"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="600" Height="500"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
         mc:Ignorable="d">
  <Border BorderThickness="12" BorderBrush="Fire brick" CornerRadius="12"
         Width="595" Height= "495">
<Grid x:Name="LayoutRoot" Background="Gainsboro" ShowGridLines="True" >
  <Grid.RowDefinitions>
    <RowDefinition Height="0.352*"/>
    <RowDefinition Height="0.648*"/>
  </Grid.RowDefinitions>
```

```
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="0.4*"/>
   <ColumnDefinition Width="0.6*"/>
</Grid.ColumnDefinitions>

<!—ScrollViewer -->
<ScrollViewer HorizontalScrollBarVisibility="Auto"
   VerticalScrollBarVisibility="Auto" Grid.Column ="0" Grid.Row="0"
   Grid.ColumnSpan="2" >
  <TextBlock Text="dotNetManía" FontFamily="Arial Black" FontSize="120"
            VerticalAlignment="Center" Margin="20, 0, 20, 0" />
</ScrollViewer>

<!—Sliders -->
<StackPanel Orientation="Horizontal" Grid.Column ="0" Grid.Row="1">
   <Slider x:Name="SldA lpha" Width="150" Verti calAlignment="Top"
       Margin="20,10,0,0" Minimum="0" Maximum="255"
       ValueChanged="Cambio_Valor" />
  <TextBlock FontFamily="Arial" FontSize="12" Text="A"
         VerticalAlignment="Top"
         Margin= "20,10,0,0"/>
</StackPanel>
<StackPanel Orientation="Horizontal" Grid.Column= "0" Grid.Row="1">
   <Slider x:Name="SldRed" Width="150" VerticalAlignment="Top"
      Margin="20,60,0,0" Minimum="0" Maxi mum="255"
      ValueChanged="Cambio_Valor" />
   <TextBlock FontFamily="Arial" FontSize="12" Text ="R"
               VerticalA lignment="Top" Margin="20,60,0,0"/>
</StackPanel>
<StackPanel Orientation="Horizontal" Grid.Column ="0" Grid.Row="1">
   <Slider x:Name="SldGreen" Width="150" Verti cal Aligment="Top"
```

```
      Margin="20,110,0,0" Minimum="0" Maximum="255"
      ValueChanged= "Cambio_Valor" />
   <TextBlock FontFamily="Arial" FontSize="12" Text="G" Vertical
              Alignment="Top" Margin= "20,110,0,0"/>
</StackPanel>
<StackPanel Orientation="Horizontal" Grid.Column= "0" Grid.Row="1">
   <Slider x:Name="SldBlue" Width="150" VerticalAlignment="Top"
     Margin="20,160,0,0" Minimum="0" Maximum="255"
     ValueChanged= "Cambio_Valor" />
   <TextBlock FontFamily="Arial" FontSize="12" Text= "B" Vertical
              Alignment="Top" Margin= "20,160,0,0"/>
</StackPanel>
   <Rectangle Stroke="Navy" x:Name="Rec" Grid.Row ="1" Grid.Column="0"
      Margin="0,160,0,0" Height="50" Width="82" Fill="White"/>

   <!–TabControl -->
    <my:TabControl Grid.Row="1" Grid.Column="1" Margin="15,15,15,15">
      <my:TabItem Header="D. Esposito">
        <StackPanel Orientation="Vertical">
          <Image Source="Fotos/Dino.png" Margin= "10, 50, 0, 0"
                VerticalAlignment= "Bottom" Stretch="None"/>
          <TextBlock Margin="100, 70, 0,0">
                <Run Text="El Blog de Dino Esposito"/>
          </TextBlock>
        </StackPanel>
      </my:TabItem>
    <my:TabItem Header="L.M. Blanco">
      <StackPanel Orientation="Vertical">
        <Image Source="Fotos/LMB.png" Margin= "10, 50, 0, 0"
          VerticalAlignment ="Bottom" Stretch="None"/>
        <TextBlock Margin="100, 70, 0,0">
```

```
          <Run Text="El Blog de Luis Miguel Blanco"/>
        </TextBlock>
       </StackPanel>
      </my:TabItem>
   <my:TabItem Header="J.M. A larcón">
       <StackPanel Orientation="Vertical">
        <Image Source="Fotos/alarcon.png" Margin="10, 50, 0, 0"
              VerticalAlignment ="Bottom" Stretch="None"/>
        <TextBlock Margin="100, 70, 0,0">
              <Run Text="El Blog de J.M. A larcón"/>
        </TextBlock>
       </StackPanel>
    </my:TabItem>
  </my:TabControl>
 </Grid>
</Border>
</UserControl>
```

Hay varios aspectos que merece la pena recalcar de este código. Primero, que hemos dejado la propiedad **ShowGridLines** del **Grid** activada para que se vea mejor la estructura del contenedor y donde está ubicado cada elemento. Además, estamos utilizando dos columnas para un solo control (**ScrollViewer**) mediante la propiedad **Grid.RowSpan**.

El comportamiento de **ScrollViewer** es simple: en función de su tamaño habilitará sendas barras de desplazamiento (vertical, horizontal o ambas) para permitir que el usuario tenga acceso a su contenido. Aquí, es un **TextBlock** con un tipo de letra muy grande el que fuerza a que sea así, pero podría ser cualquier otro.

Veamos la última celda, que maneja 2 controles de la familia de los `Tab*`. `TabControl` es un control contenedor de objetos `TabItem` que definen áreas en pantalla, de las que sólo una es visible en un momento dado. Los `TabItem` (solapas) pueden tener una cabecera con texto, y cualquier contenido (una foto y un texto, representado aquí por un objeto `Run`, ya que su única función es mostrar algún texto con formato).

Por lo demás, la funcionalidad de los `TabControl` es automática, como sucede con sus homólogos de Windows Forms, esto es, no hay que codificar nada separadamente para que muestren u oculten su contenido (al pulsar sobre otra solapa).

El objeto `TabPanel` (no presente aquí), pertenece a la misma familia, y se utiliza para la creación de plantillas personalizadas de visualización, y en esos casos, es el encargado de interpretar visualmente el contenido definido en las plantillas. Como ya hemos apuntado, en el próximo capítulo veremos más sobre creación de estilos y plantillas.

Una primera prueba con manejo de eventos

Y hemos dejado para el final el caso de los `sliders` por haber usado una técnica distinta (su mera presencia en el código no aclara demasiado el funcionamiento, aunque sea visualmente muy intuitivo). Utilizamos 4 controles `Slider` y cuatro `TextBlock`, para ilustrar qué canal de color estamos manipulando, y manejar los 4 valores ARGB del color de fondo (propiedad `Fill`) del rectángulo inferior.

Para adaptarse al rango de valores permitido se han inicializado los `Slider` con valores que van entre 0 (`Minimum`)

y 255 (`Maximum`) y ningún valor explícito (o sea, se asume 0), excepto el primero (que controla el canal *Alpha*), y que iniciamos con un valor de 255 (por que nos interesa que el color sea totalmente opaco, de entrada).

La idea es que el usuario cambie el color de fondo del rectángulo al mover los `Slider`. Al hacerlo, se produce el evento `Value_Changed`, y nosotros nos hemos suscrito a él declarando un manejador de evento común para cuando uno de los 4 cambie de valor. Al mover uno cualquiera, se pasará por el evento indicado y generaremos un nuevo color de fondo a partir de la clase `Color` del Framework. Como la acción es la misma en los 4 casos, y en todos ellos tenemos acceso a los valores de los `Slider`, solo necesitamos un procedimiento de evento con este código:

Listado 5-4

```
private void Cambio_Valor(object sender, RoutedProperty ChangedEventArgs<double> e)
   {
   Color colRect = Color.FromArgb((byte)SldAlpha. Value, (byte)SldRed.Value,
                   (byte)SldGreen. Value, (byte)SldBlue. Value);
   Rec.Fill = new SolidColorBrush(colRect);
   }
```

El procedimiento de evento regenera el valor de la propiedad `Fill` del rectángulo con cada cambio en uno de los controles `Slider`. Para ello, creamos un objeto `Color` que responda a los cambios de estado, y lo asignamos dinámicamente. La ejecución de este ejemplo ofrece una salida como la de la figura 5-4.

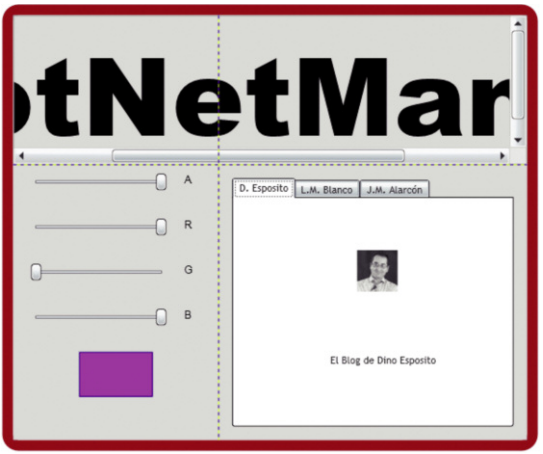


Figura 5-4. Salida por pantalla del código de ejemplo anterior.

Controles multimedia

En capítulos anteriores hemos visto cómo utilizar dos de los 3 controles disponibles específicamente para mostrar este tipo de información: `Image` y `MediaElement`. Y, no tema el lector, no vamos a crear otro reproductor de vídeo o visor de fotografías. Demasiados hay ya por la red, bien explicados y, seguramente, resueltos con más conocimientos de diseño de lo que podríamos hacer aquí. Su uso es bastante sencillo, como hemos podido comprobar en capítulos anteriores.

Pero merece la pena citar un tercer control creado expresamente para dar soporte a una tecnología que hace su debut con esta versión de Silverlight, llamada `DeepZoom`. Permite ampliar un conjunto de imágenes de tamaños cualesquiera, incluyendo alejamientos y aproximaciones espectaculares, y manteniendo un rendimiento excelente en

todo el proceso. Además, admite jugar con escenarios tridimensionales formados a partir de fotos individuales de un entorno, hasta formar un contexto de rotación de 360º.

El tratamiento de las imágenes de gran tamaño se realiza mediante la creación de una pirámide de imágenes, cuyo tamaño y resolución individual puede ser generado con la herramienta del SDK **Deep Zoom Composer**, teniendo en cuenta que ésta herramienta solo soporta el proceso de imágenes cuyo formato sea el manejado por la clase `BitmapImage`. Para la tarea de creación de escenarios tridimensionales, también se puede contar con la herramienta de Microsoft **PhotoSynth** (ver `http://photosynth.com`).

La programación de estos efectos se consigue con el control `MultiScaleImage`, que permite almacenar un número indeterminado de objetos `MultiScaleSubImage`. La documentación oficial es bien explícita sobre el funcionamiento de esta técnica e incluye ejemplos, junto a una solución completa descargable.

El resto de controles

Otros elementos de la interfaz de usuario que no aparecen en la caja de herramientas de Visual Studio pero sí en Blend, tienen ciertas peculiaridades respecto a los vistos hasta aquí. O no poseen un aspecto visual predeterminado (sino que son meros contenedores de otros), o su tipo de contenido es específico, o son utilizados para formar 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 funcionalidad por esta vía. Vamos a repasar algunos de ellos.

ContentControl

Se trata de un control muy especial dentro de esa jerarquía, ya que sirve de base a muchos otros controles y sub-controles. Esto podemos verlo más detalladamente en el diagrama 1. `ContentControl` hereda de `Frame workElement`, y —aunque no es lo más usual— llega a convertirse en origen de clases abstractas, como `CalendarButtonBase`. Entre la jerarquía a que da lugar, cabe destacar una buena cantidad de versiones de la idea de `Button`, que a su vez da origen a otros elementos, etc.

No es lo más normal utilizarlo directamente en una interfaz, pero no hay inconveniente para ello. El ejemplo siguiente muestra dos objetos `ContentControl` individuales en funcionamiento (listado 5-5).

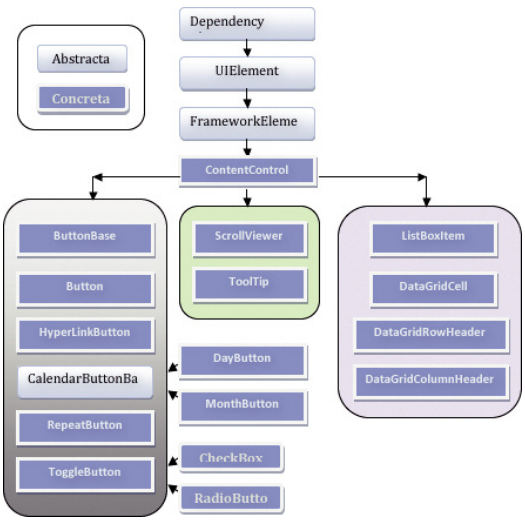


Diagrama 1. Jerarquía de clases vinculada a ContentControl.

Listado 5-5

```
<Grid x:Name="LayoutRoot" Background="Gainsboro" ShowGridLines="True">
   <Grid.RowDefinitions>
      <RowDefinition Height="A uto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
   </Grid.RowDefinitions>

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

   <ContentControl Margin="3" Grid.Row="1" Background="Black"
              BorderBrush="Red"
    VerticalAlignment="Center" Content="Control contenedor con texto simple."
         HorizontalA lignment="Center" />
   <ContentControl Margin="3" Grid.Row="2">
      <CheckBox Content="Este contiene un control CheckBox"
         HorizontalAlignment ="Center" Background="Gainsboro" />
   </ContentControl>
</Grid>
```

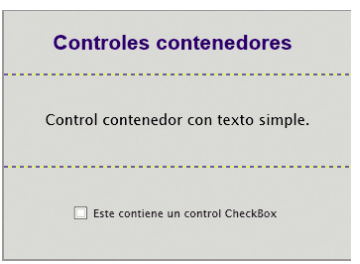


Figura 5-5. Salida por pantalla del código de ejemplo anterior.

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 5-5).

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

a) Debe ser un elemento que herede de `UIElement`.
b) Puede incluir otros objetos (pero entonces, simplemente, se llamará al método `ToString()` de cada objeto contenido).
c) Puede incluir otros objetos con plantillas: 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: sus capacidades se suministran mediante un servicio (al estilo de otros de Windows Forms), llamado `ToolTipService`. Este servicio, provee de métodos estáticos para mostrar una etiqueta flotante.

Su funcionamiento predeterminado es a partir de `ToolTipService` (que dispone de un único miembro, `ToolTip`), y se declara incluyéndole dentro del elemento sobre el que queremos que aparezca la etiqueta. Hemos modificado el `TextBlock` del elemento anterior para que muestre una simple etiqueta flotante con un texto, mediante al siguiente código:

Listado 5-6

```
<TextBlock Text="Controles contenedores" Margin=" 0,20,10,20" FontFamily="Arial"
         FontSize="21" FontWeight="Bold" Foreground="Navy"
         Horizontal Alignment="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>
```

Esto 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, solamente incluye el elemento a mostrar:

Listado 5-7

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

```
  </Border>
</ContentControl>
```

Así pues, tenemos 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, con la inclusión de un elemento `ToolTip` dentro de la declaración de `ToolTipService`. Si se hace así, podremos crear toda una interfaz compleja como parte de la etiqueta.

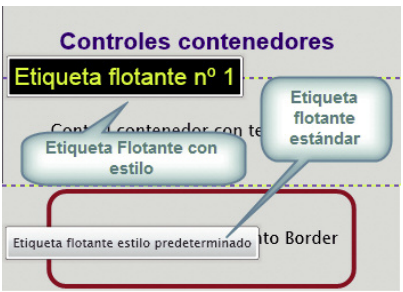


Figura 5-6. Distintos formatos de etiquetas flotantes.

Los "Presenters"

Aunque lo parece, no se trata de un grupo musical, sino del conjunto de controles que lleva ese sufijo, y que comparten algunos elementos comunes. Son 4: `ContentPresenter`, `InkPresenter`, `ItemsPresenter` y `ScrollContentPresenter` (que hereda directamente del primero). Todos pertenecen al espacio de nombres `System.Windows. Controls`[2].

2. Existe otro conjunto de "Presenters", todos ellos ligados al control DataGrid.

Están pensados para servir de contenedores de otros, y especialmente, para suministrar una infraestructura común sobre la que aplicar estilos a los controles con los que se asocian, pero 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 PDAs, PocketPCs, TabletPCs, 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 individualmente, sin jugar el papel de mero "ladrillo" constructor de otros controles más complejos.

Thumb, ToggleButton y RepeatButton

Concluimos este capítulo con una descripción de 3 típicos sub-controles. 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 otros controles como `ScrollBar`, donde adopta la forma de un rectángulo desplazable a lo largo de la

superficie limitada en ambos extremos por dos controles **RepeatButton**, pero puede utilizarse independientemente, sobre todo, en la construcción de controles de usuario (ver figura 5-7).

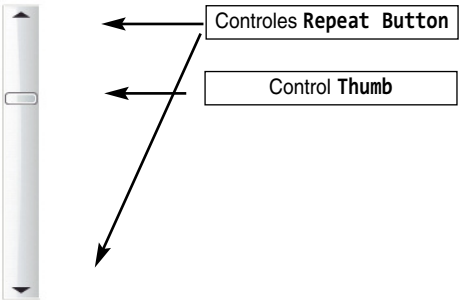


Figura 5-7. Controles Thumb y RepeatButton como constructores de un Scrollbar.

La gestión del proceso *Drag&Drop* 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 suele ser otro de los controles que nos quedan por comentar: **RepeatButton**.

RepeatButton sirve 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**.

Por su parte, **ToggleButton** es un botón con dos estados visuales estáticos: presionado y liberado (normal). Dispone de una definición de plantillas que permiten distinguir

claramente cuándo se encuentra en un estado u otro y una propiedad `IsChecked`, que permite determinar su estado. Además, puede configurarse para que se comporte como un control tri-estado, igual que `CheckBox`.

Otros controles disponibles

En el momento de cerrar la maquetación de esta obra, Microsoft acaba de anunciar la disponibilidad de un nuevo paquete de controles, componentes y herramientas accesibles públicamente en el sitio de **CodePlex**, bajo el nombre de "Silverlight Toolkit" (`http://www.codeplex.com/Silverlight`).

Incluye un paquete de controles, (algunos están en fase beta y otros son estables), herramientas para gráficos empresariales (*charting*), soporte de pruebas unitarias, soporte de temas de fondo (*themes*), nuevos mecanismos de apoyo para la visualización de datos y algunas cosas más.

A destacar, varios controles que estaban disponibles en WPF pero no en Silverlight, como `DockPanel` y `WrapPanel` (paneles), `Expander`, `Label`, `TreeView`, `ViewBox`, `NumericUpDown` y algunos otros. La presencia de estos nuevos controles, incrementa, además, la compatibilidad con el código existente en aplicaciones WPF de navegador.

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 de WPF.

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 la funcionalidad básica proporcionada por sus "ladrillos" constructores.
En el próximo capítulo veremos ejemplos de cómo usar estas familias de controles constituyentes creando escenarios visuales personalizados mediante estilos y plantillas. También veremos cuál es el proceso de creación de un control reutilizable.

NOTA. A la conclusión de esta obra, Microsoft ha hecho público un Silverlight Toolkit, con nuevos controles y herramientas, disponible en CodePlex: **`http://www.codeplex.com/silverligh.`**

6

Plantillas, animaciones y Visual State Manager

Dependency Properties y eventos

Sabemos por otros entornos de desarrollo que un evento es un mecanismo de comunicación entre 2 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 S*ilverlight 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, en Silverlight 2.0 se 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 2, las propiedades se exponen, típicamente, como propiedades clásicas del CLR. Podríamos estar utilizando muchas de ellas sin saber que se trata en realidad de propiedades 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 la parte final de este capítulo, mostramos esta técnica como parte de la definición de un control de usuario. La sintaxis de declaración de una **DependencyProperty** es la siguiente:

Listado 6-1

```
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); }
    }
```

La clase estática **DependencyProperty** registra la propiedad **Text**, anotando su tipo, su clase, la cadena que se utilizará para nombrarla y —eventualmente— un 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 esa propiedad se modifique.

El sistema de eventos en Silverlight 2

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 HTML[1]. Silverlight, por el momento, solo soporta unos pocos 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 *trigger* —que veremos más adelante a propósito de las animaciones— y es el único evento cuyo manejador puede ser declarado en puro lenguaje XAML. En Silverlight 2, la principal razón para la existencia de `RoutedEvents` es la compatibilidad con WPF.

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` (se indica que el evento ha sido manejado).

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

1. 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*".

eventos tengan un único manejador común. Veíamos 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.

Listado 6-2

```
<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"
          Horizontal Alignment="Right" MouseEnter= "Thumb1_MouseEnter" />
    </StackPanel>
  </Border>
</Grid>
```

Listado 6-3

```
public Page()
  {
```

```
    InitializeComponent();
    Thumb1.MouseLeave += new MouseEventHandler (Thumb1_MouseLeave);
    Thumb2.MouseLeave += new MouseEventHandler (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, Mouse EventArgs e)
  {
    Brush Fondo = new SolidColorBrush(Colors.Blue);
  Thumb1.Fill = Thumb2.Fill = Fondo;
  }
```

En los listados 6-2 y 6-3, mostramos 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#:

Para indagar un poco más en profundidad lo que sucede, un vistazo con .NET Reflector[2] a la clase **Rectangle**, nos muestra algunas cosas interesantes, tal y como refleja el listado 6-4.

[2]. El producto ha sido adquirido recientemente por Red Gate quien se ha comprometido a continuar su desarrollo. No obstante, sigue siendo gratuito (www.RedGate.com).

Listado 6-4

```
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; }
}
```

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 6-5

```
<Grid x:Name="LayoutRoot" Background="White">
  <Canvas Width="400" Height="150" Background="Blanched Almond"
                          KeyUp="Control Teclado">
    <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, programamos el evento **KeyUp**:

Listado 6-6

```
void ControlTeclado(object sender, KeyEventArgs e)
   {
        if (e.Key != Key.Unknown)
        {
             String msg = "La tecla " + e.Key.ToString();
             msg += " se pulsó cuando el foco estaba en " +
             (e.Source as Framework Element).Name;
             txtInfo.Text = msg;
        }
   }
```

Con lo que averiguamos cuál de los `TextBox` se encuentra activo (figura 6-1). El truco reside en que no utilizamos el argumento `Sender`, sino la propiedad `Source` del 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.

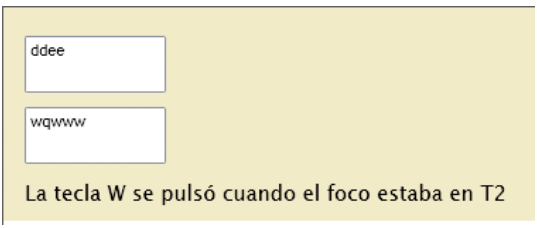


Figura 6-1. Salida del código anterior.

Estilos y plantillas

Una de las herramientas más poderosas para diseñar el aspecto visual de un elemento en Silverlight 2, 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 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. Más adelante, podemos 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 define la apariencia y el comportamiento para todos sus estados posibles. Siempre que se trate de un elemento que herede de `FrameworkElement`, podemos asociarle un estilo (`Style`).

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`, para indicar a qué controles debe aplicarse. No obstante, para aquellos lectores que conozcan WPF, hay algunas diferencias en la forma de utilizar estilos con Silverlight 2:

- En Silverlight, se deben usar atributos `x:Key` en los estilos personalizados y referenciarlos como recursos estáticos. No se soportan estilos implícitos aplicados mediante el atributo `TargetType`.
- Los estilos no pueden basarse en otros estilos para crear jerarquías mediante el atributo `BasedOn`.
- Podemos definir estilos para sobrescribir otros predeterminados, pero si se intenta aplicar el mismo estilo otra vez provocaremos una excepción. Sin embargo, sí 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.

Puede ver cómo cambiar el estilo de un elemento simple en el listado 6-7.

Listado 6-7

```
<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>
```

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 6-8

```
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));
}
```

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. Podemos crear un estilo que reescriba el contenido de la propiedad `ControlTemplate`, y, al aplicarlo a un control, habremos cambiado totalmente su aspecto gráfico.



Figura 6-2. 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 6-9

```
<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" Vertical Alignment="Center"/>
    </Border>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

Esto redefine la plantilla del botón, y por tanto, anula la 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 significativamente a la apariencia gráfica de cada botón):

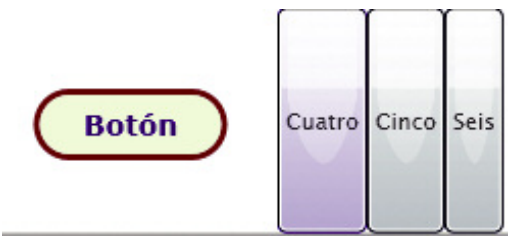


Figura 6-3. Modificación en el botón anterior comparada con el aspecto de los otros botones.

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

Asignación Late Binding de una plantilla

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 6-4:



Figura 6-4. La nueva plantilla, utilizando la extensión *TemplateBinding*.

Ahora, el tercer botón tiene un fondo individualizado, aunque toma el resto de sus propiedades de la plantilla. Además, `Control Template` 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.

ContentPresenter

Relacionado con esta operativa está 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: 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 6-10

```
<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"
     Font Weight="Bold" Foreground="#FF5C9A C9" Grid.Row="0"
     HorizontalAlignment="Center" />
  <Button Content="Botón" FontFamily="Verdana" FontSize="18" Grid.Row="1"
    Background= "BurlyWood" Width="230" Height="90"
    Horizontal Alignment="Center" Vertical Alignment="Center">
      <Button.Template>
        <ControlTemplate TargetType="Button" x:Name="ButtonTemplate">
          <Grid>
            <Border BorderThickness="5" BorderBrush ="Navy" CornerRadius="16"
                    Back ground="{TemplateBinding Back ground}" />
```

```
        <ContentPresenter x:Name="Button Presenter"
                  Content="{Template Binding Content}"
                     HorizontalA lignment="Center"
                     VerticalAlignment ="Center" />
        <Image Source="Graficos/serv.png" HorizontalAlignment="Left" VerticalAlign-
        ment="Center" Margin="15,12,0,12"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
 </Button>
</Grid>
```

Este código produce la salida de la figura 6-5, 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 6-5. Demo 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**.

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 ventana 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 6-6:

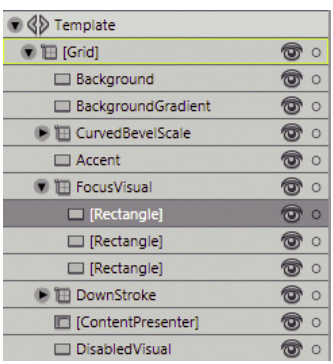


Figura 6-6. Estructura interna de un botón.

La estructura muestra que, internamente, un botón está formado por un `Grid` (que, a su vez, contiene otros) más una serie de rectángulos (`Background`, etc.), `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.

Silverlight 2.0 Animation System

Hasta aquí, hemos tratado con elementos estáticos. Si queremos incluir aspectos dinámicos, disponemos de tecnologías especialmente habilitadas para ello. El conjunto de todas ellas recibe el nombre de Silverlight Animation System, e incluye, como parte del sistema, una herramienta muy poderosa para manejar cambios de estado, llamada Visual State Manager.

El sistema de animaciones de Silverlight agrupa un conjunto de objetos de soporte para expresiones diná-

micas que Microsoft ha incluido en el *runtime*, y que se apoya en las capacidades de su hermano mayor (WPF).

Animaciones

Las nuevas interfaces de usuario se caracterizan por su dinamismo. Ese ha sido uno de los elementos diferenciadores de **Adobe Flash**, y ésa es una de las grandes virtudes de esta versión de **Silverlight**. 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*. El mecanismo de creación de vídeo no es más que un sistema de animaciones de fotogramas. Pero no vamos a ocuparnos aquí de los vídeos, sino de las animaciones de gráficos generadas por nosotros.

Animaciones en Silverlight

En Silverlight, *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` o `Height`. Si queremos que se oculte poco a poco, vamos reduciendo el valor de su propiedad `Opacity`, y así sucesivamente. La versión de XAML soportada por Silverlight contiene muchos objetos cuyas propiedades son susceptibles de animarse, en este sentido.

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.

Transformaciones vs. animaciones

Otra de las características relacionadas con las animaciones para la que tenemos soporte en Silverlight es el concepto de transformación. Una transformación establece cómo corresponder puntos de una coordenada del espacio a otra utilizando una matriz de transformación que provee un mecanismo de conversión simple. En el lenguaje XAML de Silverlight 2.0, existen predefinidas 4 clases 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 transformaciones[3].

Vamos a comenzar por éstas últimas.

Transformaciones

Las transformaciones se crean aplicando propiedades de transformación a distintos tipos de objetos. Dependiendo del elemento, las transformaciones soportadas

3. 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://enteos2.area.trieste.it/ russo/LabInfoMM2005-2006/ ProgrammaEMaterialeDidattico /daStudiare/009-FourierInSpace.html**.

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 los 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 vemos en la figura 6-7.



Figura 6-7. Código fuente y resultado visual del fragmento de código correspondiente a una rotación.

Como puede verse en el código de la figura adjunta, añadimos el atributo **Render Transform** al cuerpo del control (en este caso, el `TextBlock`), y especificamos una transformación por rotación, con un ángulo de 45º empezando el proceso en las coordenadas 150, 30, a partir de la esquina superior izquierda del control. Una imagen visual para darse cuenta exactamente del proceso, consiste en "clavar un alfiler" mentalmente en las coordenadas indicadas, y rotar los grados que se indiquen toda la superficie del control afectado.

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 `x` de un rectángulo (tenga en cuenta que `RenderTransform` solo se puede establecer una vez):

Listado 6-10

```
<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>
```

De la misma forma, para aplicar las Transformaciones de Elongación y Translación la propiedad `RenderTransform` aplicada al mismo rectángulo podría tomar los siguientes valores:

Listado 6-11

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

Para una elongación o estiramiento de 45 grados, y...:

Listado 6-12

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

...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 definir 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 6-13

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

Animaciones

Indicábamos al comienzo de este apartado que las animaciones se basan 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. 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`.

Triggers

Una animación es un proceso que tiene lugar siempre como respuesta a un evento. Y, en algunos casos, este evento puede ser expresado únicamente en Silverlight XAML mediante un desencadenador o *trigger*, representado por un atributo más del elemento y programado como parte de su definición de estilo. A diferencia de lo que sucede en WPF, solo existe un tipo de *trigger* soportado, el `EventTrigger` (en WPF disponemos de `DataTriggers`, `MultiTriggers` y otros elementos similares).

Además, como un elemento puede responder a más de una acción, se dispone de una colección de `Triggers` (de ratón, de teclado, etc), de forma que si queremos que, como respuesta a alguna acción por parte del usuario o del sistema, se produzca la reacción correspondiente, debemos asignar al atributo un manejador adecuado.

El trigger más simple sólo en código XAML

Vamos a ver un ejemplo sencillo de esto partiendo de nuestro código del rectángulo. Supongamos que queremos que se produzca una animación consistente en desplazar ese 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 *siempre y cuando queramos que la animación se produzca en la carga del control*. Si lo que queremos es que 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 6-14

```
<UserControl x:Class="A nimacioni2.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.Target Name="rect"
                        Storyboard.TargetPro perty="(Canvas.Left)"
                        AutoReverse ="True" Duration="0:0:5" To="300"/>
                   </Storyboard>
               </BeginStoryboard>
           </EventTrigger>
       </Rectangle.Triggers>
   </Rectangle>
 </Canvas>

</UserControl>
```

Lo único que hemos hecho es añadir un *trigger* a la colección de *triggers* del elemento **Rectangle**, indicar al *trigger* que se vincule con el evento **Loaded** (como dijimos antes, el único posible si no queremos añadir código paralelo para manejarlo), y establecer sus propiedades. Aquí hemos incluido el objeto **DoubleAnimation** dentro de un elemento **Storyboard**, y, éste, dentro de un **BeginStoryboard**, encargado de distribuir las animaciones que contiene a los objetos y propiedades adecuadas (por el momento no existe un objeto-contrapartida llamado **EndStoryboard**).

Respuesta a eventos mediante código

Si se pretende que sean eventos de usuario los que lancen los procesos de animación, tenemos que recurrir a los **Resources** asociados con el elemento. El código fuente para lanzar la respuesta 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. Damos un nombre a cada animación para referenciarlas desde el código C# (o cualquiera que sea el lenguaje). Por ejemplo, para modificar el proceso anterior de forma que se disparase al pulsar sobre el rectángulo, el código XAML se convertiría en:

Listado 6-15

```
<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>
```

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 6-16

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

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 vuelta 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 un botón en lugar de un rectángulo, o cualquier otro *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, bautizamos como **Crea YEjecutaAnimation()**. El código sería el siguiente:

Listado 6-17

```
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.A dd(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
}
```

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 lineal[4] utilizado internamente para la construcción de la animación.

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.

El esquema del siguiente ejemplo queda ilustrado en la figura 6-8.

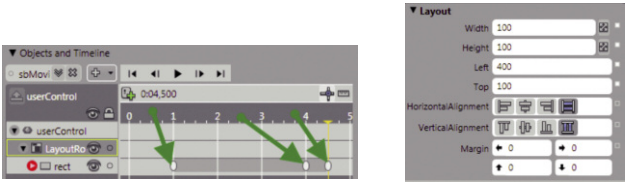


Figura 6-8. La línea de tiempo abarca un total de 4.5 segundos,y está dividida en 3 segmentos de duración variable.

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/seg., en la segunda 200 en 3 segundos (va más despacio) y en la tercera 100 píxeles en 0,5 seg. (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 and 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.



Figuras 9a y 9b. Editor de líneas de tiempo en Expression Blend mostrando los 3 puntos segmentación de la animación. A la derecha, sub-ventana de edición de posiciones de objetos 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, 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:

Listado 6-18

```
<UserControl.Resources>
  <Storyboard x:Name="sbMovimiento">
    <DoubleA nimationUsingKeyFrames 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>
```

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 6-10a 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 6-9.

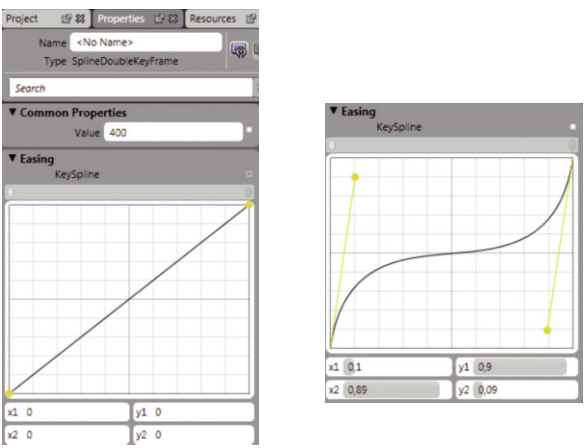


Figura 6-10a y 6-10 b. Representación gráfica de la interpolación lineal y curva propia del uso de KeyFrames.

La curva que describe la interpolación (el cálculo de valores intermedios entre dos dados 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 de la figura 6-10-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, en realidad. De hecho, 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 6-19

```
<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.

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 respecto 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*** (**Partes**): los elementos internos constituyentes del control.
- ***States & StateGroups*** (**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*** (**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 6-11, 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, en la escasa literatura al respecto (literatura digital, de momento), no se explica suficientemente un aspecto del funcionamiento de VSM como es

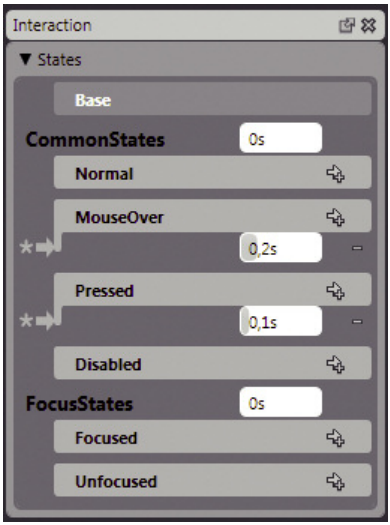


Figura 6-11. Ventana de interacción para definir transiciones de estado.

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.

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 2, utilizando Blend 2.0 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 otra apariencia 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, 45º cuando se encuentre en su estado `Pressed`.
- 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 6-12:



Figura 6-12. 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 identificación 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 6-13:



Figura 6-13. 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, como no se cambia esta vez el contexto visual pre-programado por el modelo, tenemos que intervenir en el código fuente para llamar al cambio de estado, cuando se haya producido el cambio lógico. 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 6-20

```
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);
   }
```

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 6-14).

Y hemos añadido una etiqueta flotante para que el usuario sepa que el icono le lleva nuevamente al estado inicial.

El código XAML completo puede verlo en el listado 6-21.

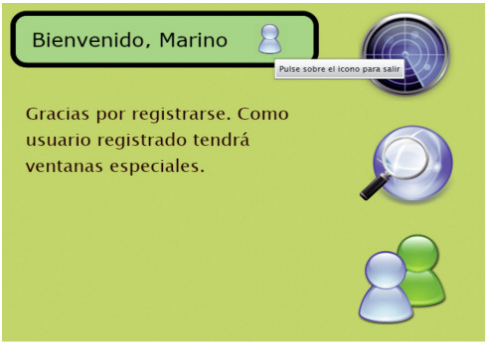


Figura 6-14. Resultado de la transición mostrando una etiqueta flotante.

Listado 6-21

```
<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)">
      <SplineDoubleçKeyFrame 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>
```

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 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 construidos 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 que 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`). Depen-derá 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— que los contenidos sean de tipo numérico y

calcular la letra asociada a ese número suponiendo que se trata de un DNI y 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 vacio 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 6-22

```
<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">
```

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, dejandolo 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 6-23

```
<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>
```

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, 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 `Dependency Property`, 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 6-24

```
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); }
}
```

A partir de este momento, se podría asignar el valor de la propiedad `Text` de `TextoDNI` desde el código XAML igual que otra propiedad cualquiera que ofrezca el sistema de Intellisense.

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 del listado 6-24, que representa el estado final de la clase).

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. El código de pruebas puede verlo en el listado 6-25. Con él obtendrá una salida como la de la figura 6-14.

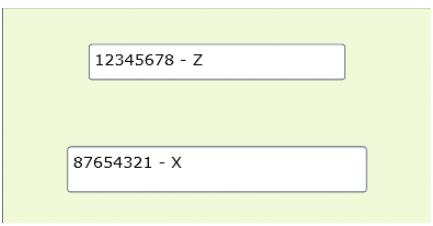


Figura 6-14. .El control TextoDNI en funcionamiento.

Listado 6-24

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Text.RegularExpressions;

namespace ControlesDNM
{
 public class CajaTextoDNI : ContentControl
  {
     string Letras = "TRWA GMYFPDXBNJZSQVHLCKET";
     Regex reNum = new Regex(@"^\d+$");
     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); }
     }

     public CajaTextoDNI()
     {
       this.DefaultStyleKey = typeof(CajaTextoDNI);
       this.Loaded += new RoutedEventHandler(CajaTextoDNI_Loaded);
       this.LostFocus += new RoutedEventHandler
                   (CajaTextoDNI_LostFocus);
     }

     void CajaTextoDNI_LostFocus(object sender, RoutedEventArgs e)
```

```
    {
      string Texto = (e.Source as TextBox).Text;
      bool esNumero = reNum.Match(Texto).Success;
      if (Texto != "" && esNumero)
      {
                (e.Source as TextBox).Text += " - " +
                Letras[Int32.Parse(Texto) % 23];
      }
      else { (e.Source as TextBox).Text = "Error en el DNI"; }
  }

  void CajaTextoDNI_Loaded(object sender, RoutedEventArgs e)
  {
      bool esNumero = reNum.Match(Text).Success;
      if (Text != "" && esNumero)
      {
          Text += " - " + Letras[Int32.Parse(Text) % 23];
      }
      else { Text = "Error en el DNI"; }
   }
 }
}
```

Listado 6-26

```
<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>
```

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.

7

El tratamiento de datos

El modelo de ejecución en *sandbox* que propone Silverlight supone que los mecanismos de acceso a datos no pueden seguir los patrones típicos de las aplicaciones de escritorio, y, en buena parte, de las aplicaciones Web tradicionales. En su lugar, Silverlight plantea un modelo de acceso a datos basado en servicios Web, en los que podemos utilizar servicios tradicionales ASMX basados en SOAP, servicios WCF, servicios REst y **ADO.NET Data Services** (antes llamados proyecto Astoria).

En el momento de escribir esto, ya existe una versión de ADO.NET Data Services disponible que forma parte del Service Pack 1 de Visual Studio 2008. Puede considerarse una extensión de **Data Entity Framework**, pero especialmente pensada para su uso en aplicaciones RIA.

En cualquier caso —incluyendo el acceso a objetos de negocio locales que pudiéramos tener en nuestra aplicación—, es imprescindible tener claro el concepto de *DataBinding* en Silverlight, de manera que empezaremos por ver cómo funciona este mecanismo y continuaremos

analizando las opciones de acceso a datos más atractivas que presenta la plataforma.

Éstos son los aspectos a abordar en este capítulo:

1) *Databinding* en Silverlight 2.
2) Implementación de *Databinding* de un solo sentido en una clase personalizada.
3) Acceso a datos desde un servicio WCF y presentación de datos en un control `DataGrid`.
4) Acceso a servicios SOAP tradicionales.
5) Interacción con servicios *RESTful* sobre HTTP: acceso a orígenes de datos *RSS*.
6) ADO NET Data Services.
7) Vinculación de objetos de datos desde Expression Blend 2.0.

DataBinding en Silverlight 2

La separación entre los datos mostrados y su origen se asemeja a la separación entre diseño y funcionalidad que preconiza el modelo de trabajo de Silverlight. Dicha separación no supone un problema en el código XAML gracias al mecanismo de *DataBinding*. Ya hemos visto algunas de las formas de utilización del enlace a datos en los capítulos anteriores, especialmente, a la hora de usar los recursos definidos a nivel de aplicación. Vamos a ahondar algo más en el tema (recordando que los mecanismos de enlace a datos en Silverlight 2, son un subconjunto de los existentes en Windows Presentation Foundation).

Se trata de vincular un dato existente en una propiedad del `CoreCLR` con una propiedad de un `UIElement` (elemento de

la interfaz de usuario) como `Text`, `Background`, o cualquier otra. Hacer corresponder un origen y un destino, de forma que los mecanismos del `CoreCLR` mantengan ambos en sincronía (al menos, inicialmente).

Sintaxis de DataBinding en XAML

El patrón sintáctico de *Databinding* en Silverlight, es casi idéntico al utilizado en WPF. Por ejemplo, para enlazar el texto de un botón (propiedad `Content`) al contenido de una variable accesible de nombre `vDatos`, utilizaríamos:

```
<Button Content = "{ Binding vDatos }" />
```

El sistema se comporta como una "maquina conectora", que enlaza los orígenes de datos con sus destinos. Ahora bien, puede funcionar en 3 modos (`BindingMode`):

- **OneTime**: *binding* único o inicial: que actualiza los datos una sola vez con la información del origen, cuando se crea el *binding*.
- **OneWay**: *binding* de un solo sentido: Actualizan el destino con los datos de origen cuando se crea el *binding* y cada vez que cambia el origen.
- **TwoWay**: *binding* en los dos sentidos: actualiza ambos, origen y destino, cuando se modifica el valor de cualquiera de los dos (sincronía).

Téngase en cuenta que hay situaciones en las que un mismo proceso puede necesitar de los 3 tipos de enlace simultáneamente, dependiendo de los campos. Por

ejemplo, supuestos 3 datos como "*Hora de conexión*", "*Datos_WebMaster*" y "*Stock_Disponible*", podría darse que cada uno de ellos perteneciese a uno de los tipos citados (en ese orden).

DataContext e ItemSource

Muchos elementos de la interfaz de usuario disponen de dos propiedades especialmente pensadas para el enlace a datos: `DataContext` e `ItemSource`. La primera, se utiliza para identificar el origen de datos de un enlace. Puede consistir en datos planos, pero suele asociársele un objeto `Binding`, que dispone de medios para obtener los orígenes de datos. Los bloques de texto (en el ejemplo siguiente) y otros controles, presentan esa propiedad. Una ventaja adicional en el uso de **DataContext**, es que puede asignarse a un contenedor de otros objetos, tanto en lenguaje XAML, como por código. Típicamente, el enlace suele ser a una clase del `CoreCLR`.

`ItemSource`, sin embargo, se diferencia de `DataContext` en que se usa para especificar una colección de objetos origen que debe usarse para la lectura del contenido. En realidad, es válido cualquier objeto que implemente la interfaz `IEnumerable`.

Ejemplo

Vamos a crear un nuevo proyecto Silverlight 2 en Visual Studio, llamado `LibrosNetalia`, y añadiremos una clase de usuario llamada `Libro`, que representará de forma sencilla una capa de negocio. Ahora bien, queremos que los cambios en la capa de negocio sean reconocidos por lFa interfaz de usuario. Es decir, que cuando se cambie

el valor de la propiedad en la clase que contiene los datos de origen, la interfaz lo refleje automáticamente.

La interfaz INotifyPropertyChanged

Para ello, basta con hacer que nuestro objeto de negocio implemente la interfaz `INotifyPropertyChanged` (expuesta por el espacio de nombres `System.Component- Model`). Su contrato especifica que debe incluirse un evento del tipo `PropertyChanged-EventHandler`, al que —por convención— llamaremos `PropertyChanged`. Cada vez que se cambia el valor de la propiedad, el método encargado del cambio notificará a la interfaz de usuario esta circunstancia. En la práctica, de esa tarea se encarga el método `Set` de la propiedad que lanza el evento `Property-Changed`. Implementamos el caso de un único campo (el `Título` del libro), que sería el que vemos en el listado 7-1.

Listado 7-1

```
namespace LibrosNetalia
{
 public class Libro : INotifyPropertyChanged
 {
   private string Titulo_Libro;
   public event PropertyChangedEventHandler PropertyChanged;

   public string Titulo
   {
     get { return Titulo_Libro; }
     set
     {
```

```
      Titulo_Libro = value;
      if ( PropertyChanged != null )
        {
           PropertyChanged(this, new Property ChangedEventA rgs("Titulo"));
        }
     }
   }
 }
}
```

El modelo seguido para la propiedad **Titulo**, sería similar para el resto de propiedades de las que la interfaz de usuario deba estar al tanto.

Supongamos que —en esta primera fase— vamos a añadir 2 campos más para mostrar al usuario: El autor (tipo **string**) y sus existencias en almacén (tipo **int**). No lo complicamos más para mantener el código legible y, por la misma razón, los datos a mostrar estarán empotrados dentro de la clase **Page**.

El siguiente paso a dar es diseñar una interfaz de usuario sencilla capaz de mostrar los 3 datos de cada libro disponible en almacén. Nos basta con un **Grid** de un par de columnas y 4 filas que contenga los rótulos descriptivos de cada campo y el valor del campo. Y añadimos al final un botón para permitir el paso de un libro al siguiente. La parte más interesante del código XAML es la del mecanismo de *binding*.

En el código adjunto, las propiedades variables (los datos a mostrar de cada libro), van enlazadas mediante la sintaxis:

```
<Elemento Propiedad_a_enlazar= {Binding Nombre_del_Campo} />
```

Donde **Nombre_del_Campo** hace referencia a la propiedad pública de la entidad de negocio que crearemos después.

Listado 7-2

```
<UserControl x:Class="Cap7_Datos1.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="450" Height="230">
  <Border Background="Navy" CornerRadius="20" BorderBrush="Navy"
        BorderThickness="5">
  <Grid x:Name="LayoutRoot" Background="AliceBlue" Margin="5 5 5 5">
      <Grid.RowDefinitions>
          <RowDefinition MaxHeight="45" />
          <RowDefinition MaxHeight="45" />
          <RowDefinition MaxHeight="20" />
          <RowDefinition MaxHeight="45" />
          <RowDefinition MaxHeight="65" />
      </Grid.RowDefinitions>

      <TextBlock Text="Cuadernos Técnicos de dotNetManía" FontSize="18"
          FontWeight="Bold" VerticalAlignment="Center"
          HorizontalAlignment="Center" Grid.Row="0" />
      <StackPanel Orientation="Horizontal" Margin="33,0,0,0" Grid.Row="1">
        <TextBlock x:Name="tbTitulo" Text="Título: "
              VerticalAlignment="Center"
              HorizontalA lignment="Left" Grid.Row="1" />
        <TextBlock x:Name="tbTituloLibro" Text="
                          {Binding Titulo, Mode=OneWay }"
```

```
            VerticalAlignment="Center" FontSize="13"
            FontWeight="Bold" TextWrapping="Wrap" Grid.Row="1" />
      </StackPanel>
      <StackPanel Orientation="Horizontal" Margin="33,0,0,0" Grid.Row="2">
        <TextBlock x:Name="tbA utor" Text="Autor: " VerticalAlignment="Center"
              HorizontalAlignment="Left" />
        <TextBlock x:Name="tbAutorLibro" Text="{Binding Autor, Mode=OneWay}"
              FontWeight="Bold" VerticalAlignment="Center" />
      </StackPanel>
      <TextBlock x:Name="tbExistencias" Text="Existencias:"
              VerticalAlign-ment="Center" HorizontalAlignment="Left"
              Grid.Row="3" Margin="33,0,0,0" />
      <TextBox x:Name="txtExistencias" Text="
                           {Binding Existencias, Mode=TwoWay}"
              Vertical Alignment="Center" HorizontalAlignment ="Center"
             Height="21" Width="190" Grid.Row="3" />
        <Button x:Name="Change" Content="Siguiente Libro" Height="30"
              Width= "120" HorizontalAlignment ="Center"
              VerticalAlignment="Center" Grid.Row="4" />
    </Grid>
  </Border>
</UserControl>
```

A continuación, en la clase de negocio (**Libro.cs**) añadimos dos propiedades: **Autor** y **Existencias** del producto, con sus llamadas a **PropertyChanged**, para notificar a la interfaz de usuario:

Listado 7-3

```
public class Libro : INotifyPropertyChanged
 {
   public event PropertyChangedEventHandler PropertyChanged;
   private string Titulo_Libro;
   public string Titulo
   {
     get { return Titulo_Libro; }
     set
     {
       Titulo_Libro = value;
       if (PropertyChanged != null)
       {
         PropertyChanged(this, new PropertyChanged EventArgs("Titulo"));
       }
     }
   }
   private string Autor_Libro;
   public string Autor
   {
       get { returnA utor_Libro; }
       set {
           Autor_Libro = value;
           if (PropertyChanged != null)
           {
           PropertyChanged(this, new PropertyChanged EventArgs("Autor"));
           }
       }
   }
   private int Existencias_Libro;
```

```
  public int Existencias
  {
    get { return Existencias_Libro; }
    set
    {
      Existencias_Libro = value;
      if (PropertyChanged != null)
      {
        PropertyChanged(this, new PropertyChanged EventA rgs("Existencias"));
      }
    }
  }
```

Así conseguimos que se actualicen los datos a visualizar. Finalmente, la clase **Page** pone en marcha todo y almacena una lista genérica de objetos **Libro** (3, para simplificar). Cuando pulsamos el botón "Siguiente libro", hacemos que la variable **LibroActivo** tome los datos que se quieren mostrar y en ese momento actúa el mecanismo de *binding*. Éste es el código completo de la clase **Page**:

Listado 7-4

```
public partial class Page : UserControl
 {
   List<Libro> ListaLibros;
   Libro LibroEnPantalla = new Libro();
   public int LibroActivo = 0;

   public Page()
   {
```

```
  InitializeComponent();
  Loaded += new RoutedEventHandler(Page_Loaded);
}

void Page_Loaded(object sender, RoutedEventArgs e)
{
  // Evento para el boton
  Change.Click += new RoutedEventHandler(Change_Click);
  // Valores iniciales del Libro
  ListaLibros = InicializaLista();
  LibroEnPantalla = ListaLibros[0];
  AsignarOrigenDeDatos();
}
private List<Libro> InicializaLista()
{
  ListaLibros = new List<Libro>(2);
  // Inicializa manualmente los valores de 3 libros de la lista.
  ListaLibros.Add(new Libro());
  ListaLibros[0].Autor = "Pep Lluis Baño";
  ListaLibros[0].Titulo = "Robot dispensador para MSDN video";
  ListaLibros[0].Existencias = 10;
  LibroActivo++;

  ListaLibros.Add(new Libro());
  ListaLibros[1].Autor = "Luis Miguel Blanco";
  ListaLibros[1].Titulo = "Diseño de Informes con SQL
                        Reporting Services";
  ListaLibros[1].Existencias = 30;
  LibroActivo++;

  ListaLibros.Add(new Libro());
```

```
    ListaLibros[2].Autor = "Marino Posadas";
    ListaLibros[2].Titulo = "Programación segura con .NET Framework";
    ListaLibros[2].Existencias = 20;
    LibroActivo = 0;
    return ListaLibros;
  }

  void Change_Click(object sender, RoutedEventArgs e)
  {
    // Avanzamos al siguiente y, al final, volvemos al primero
    LibroActivo++; if (LibroActivo > 2) LibroActivo = 0;
    LibroEnPantalla = ListaLibros[LibroActivo];
    AsignarOrigenDeDatos();
  }
  void AsignarOrigenDeDatos()
  {
    LayoutRoot.DataContext = LibroEnPantalla;
  }
}
```

De esta forma, se enlaza directamente el *grid* (**LayoutRoot**) con el libro activo. Cada vez que se pulsa el botón "Siguiente libro", el contador avanza una posición, se *cachean* los datos en la variable **LibroActivo** y se asigna el origen de datos. No se necesita nada más. El resultado es el de la figura 7-1.



Figura 7-1. Salida en pantalla del ejemplo anterior.

Acceso a datos mediante servicios Web

Con todo, la mayor parte de las veces, el origen de datos no coincidirá con la máquina del navegador. Como ya se ha apuntado anteriormente, la solución pasa por los servicios Web y éstas son las posibilidades más interesantes con Silverlight 2.0:

- Servicios Web basados en Windows Communication Foundation (**WCF**).
- Servicios Web tradicionales **ASMX** (basados en el protocolo SOAP).
- Servicios Web basados en **REst** (Repre-sentational State Transformation)
- **ADO.NET Data Services**.

Acceso a datos mediante servicios WCF

Hasta el momento de la aparición de los últimos *Service Pack* de Visual Studio 2008 y .NET Framework 3.5, era necesario retocar manualmente la configuración

predeterminada de los servicios Web generados por el IDE para acceder a los datos expuestos por un servicio WCF. Pero ahora disponemos de una nueva plantilla llamada "`Siverlight-enabled WCF Service`", que dispone respecto a los anteriores de las siguientes ventajas:

- Basta con que definamos los métodos en el servicio, sin necesitar la predefinición del contrato a través de una interfaz asociada.
- No hay que cambiar la configuración del *binding* de `Web.Config` a `basicHttpBinding`, ya que lo asume de forma predeterminada.
- No es necesario especificar la compatibilidad con ASP.NET para ciertos casos.

NOTA. Si el usuario tuviera la necesidad de operar con Silverlight mediante la creación de un servicio WCF tradicional, tenga en cuenta que estos tres aspectos necesitarán de un retoque, si bien funcionan perfectamente, una vez hecho esto.

Ejemplo

Supongamos un escenario de acceso: en un servidor de bases de datos SQL Server, tenemos instalada una base de datos de ejemplo, llamada `Pedidos08` (disponible en formato Access junto a los ejemplos descargables, en el sitio Web de **dotNetMania**), con 5 tablas muy simples. Al crear un nuevo proyecto Silverlight, nos aseguraremos de seleccionar la opción de una solución que contiene una aplicación Web. A continuación, resolvemos la arquitectura de la aplicación de servidor, que requerirá de dos nuevos ítems: la conexión con el servidor de datos y la creación del servicio WCF usando la plantilla antes citada.

El gráfico adjunto muestra someramente la arquitectura de la figura 7-2.

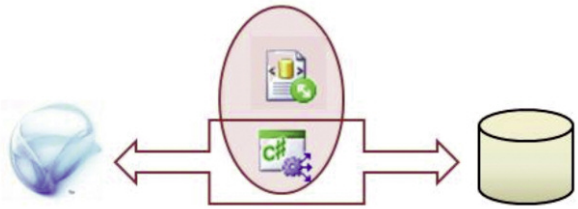


Figura 7-2. Arquitectura de la aplicación de ejemplo de uso de WCF[1].

Tendremos, pues, dos proyectos: el propio de Silverlight (con una referencia al servicio WCF que expone los datos), y el proyecto Web, que incluye el servicio WCF, la clase que accederá a los datos (usaremos LINQ-to-SQL) y la configuración necesaria para el funcionamiento.

Tenga presente el lector que una de las recomendaciones en la arquitectura de aplicaciones Silverlight es que la lógica de acceso a datos esté siempre en el servidor. De ahí que la creación de las clases para acceso a datos y el servicio mismo, estarán ubicados en la aplicación Web de pruebas.

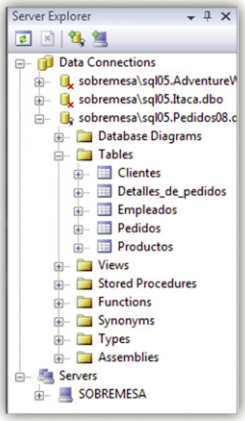
Los pasos a dar en la creación del proyecto son los siguientes:

[1]. Gráfico tomado del blog de Luis Miguel Blanco: **http://geeks.ms/ blogs/lmblanco/default.aspx.**

1) Creación del mecanismo de acceso a datos con una clase de negocio. En este proceso utilizamos la plantilla `LINQ to SQL Classes`.
2) Creación del servicio WCF que nos dará acceso a los datos: la plantilla `Silverlight-Enabled WCF Service`.
3) Modificación del servicio para acceder a los datos suministrados por la clase de negocio.
4) Creación de una interfaz de usuario XAML que muestre los datos obtenidos.

Comenzamos añadiendo un nuevo ítem al sitio Web del tipo `LINQ to SQL Classes` y seleccionamos un nombre para la clase de acceso a datos (`DatosPedidos.dbml`, en el ejemplo). El IDE de Visual Studio nos creará una estructura de 3 ficheros relacionados, uno de los cuales es una clase que sirve de *proxy* para la comunicación con el servidor. En concreto, este proceso generará la clase `DatosPedidosDataContext` (clase que hereda de `DataContext`), ya preparada para su uso, pero todavía sin conexión con datos reales.

Vinculado con el fichero `.dbml`, el IDE abre el *Diseñador de Objetos Relacionales* (ver figura 7-3a), donde deberemos de establecer el modelo de datos a usar en el aplicación. Éste es el momento de escoger los orígenes de datos y las tablas a manejar en nuestra página (puede utilizar cualquiera de las existentes en SQL Server). Una vez seleccionado el servidor de datos (visible en el Explorador de servidores), podremos arrastrar las tablas que queramos a la ventana del editor y continuar el proceso. Aquí, elegimos solamente la tabla `Clientes`, de la que vamos a pedir un listado simple.



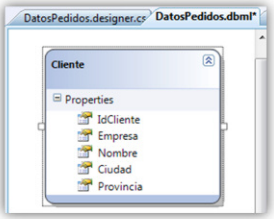


Figura 7-3a y 7-3b. Explorador de servidores mostrando orígenes de datos SQL Server y diseño de la clase Cliente en el Diseñador de Objetos Relacionales.

Dependiendo de lo seleccionado, el diseñador nos mostrará una ventana como la de la figura 7-3b, y el generador de clases habrá actualizado la clase de negocio con la información relativa a la tabla `Clientes`. En una aplicación real, tendríamos que diseñar aquí todo el modelo de datos, incluyendo relaciones entre tablas, etc.

Hecho esto, se regenera la clase de negocio, que ahora contendrá una sub-clase `Cliente`, con referencias a cada uno de los campos definidos en ella (ver fichero `DatosPedidos.Designer.cs`).

El siguiente paso es la creación del servicio. En la aplicación Web, añadimos un ítem de la clase `Silverlight-enabled WCF Service`. Llamaremos a nuestro servicio `Servicio_ Pedidos`. A continuación, modificamos el servicio para que se adapte a nuestra aplicación, creando tantas funciones como operaciones queramos que exponga el

servicio, y marcando cada una de ellas con el atributo `[OperationContract]`. Si empezamos con un listado de clientes, podemos utilizar la siguiente sintaxis LINQ para devolver ese listado:

Listado 7-5

```
[OperationContract]
public List<Cliente> ListadoClientesMadrid()
{
   DatosPedidosDataContext dp = new DatosPedidosDataContext();
   var ListaClientes = from c in dp.Clientes
                       where c.Ciudad == "Madrid"
                       select c;
   return ListaClientes.ToList();
}
```

Compilamos el proyecto Web para que Visual Studio actualice todas las referencias. Con esto, tendríamos montada la maquinaria necesaria para suministrar datos a la aplicación Silverlight, o sea, toda la parte de servidor.

Ahora tenemos que vincular el servicio desde el cliente y preparar una sencilla interfaz de usuario para mostrar los datos (usaremos un control `DataGrid`).

Lo primero, lo resolvemos con la ayuda del *Asistente de descubrimiento de servicios*, seleccionando las referencias de servicios sobre la aplicación Silverlight y pulsando la opción "Añadir Referencia de Servicio". Obtendremos una caja de diálogo como la de la figura 7-4, donde tenemos un espacio para indicar la URL del servicio

manualmente, el botón para navegar a esa URL, y la opción de "Descubrir Servicios en la propia solución" (botón "Discover" en la imagen), que es por lo que optamos aquí.

El resultado es la creación de un grupo de ficheros XML de configuración, además de uno que hace de *proxy* entre nuestro control Silverlight y el servicio propiamente dicho: `Reference.cs`. Esta clase se corresponde con los objetos de datos de la clase LINQ-to-SQLgenerada en la aplicación Web, de forma que tenemos acceso a toda la información que se encuentra en aquella como si se tratara de una clase local más.

Y una cosa más: hemos hecho que el método **ListadoClientesMadrid** devuelva la lista de clientes utilizando la sintaxis `return ListaClientes.ToList();`. Con eso conseguimos que, en la parte XAML —que vamos a construir a continuación—, el proceso de *binding* al `Datagrid` sea directo.

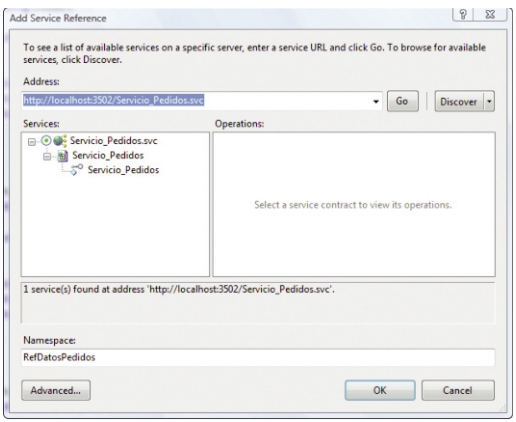


Figura 7-4. Ventana para la referencia de servicios Web.

La parte de código en C# estará prácticamente completada, a efectos del ejemplo. Solo resta el código de la llamada al servicio desde el control Silverlight cuando la página Web de prueba se cargue. Para ello, en el código XAML, definimos un manejador del evento **Loaded** del **DataGrid** (o del **UserControl**), donde después programaremos la llamada, (que será de tipo asíncrono). La interfaz de usuario XAML es muy simple:

- Un **TextBlock** con el título del listado
- Un **DataGrid** en el que indicamos que las columnas deben autogenerarse a partir de los datos suministrados en el proceso de *binding* (propiedad **AutoGenerate Columns**), y la capacidad de que el usuario cambie el tamaño de las columnas (propiedad **CanUserResizecolumns**). Todo esto, después de hacer referencia a la librería donde se encuentra el control **DataGrid** (**System.Windows.Controls**), o de arrastrarlo directamente a la ventana de código. Este es el XAML resultante:

Listado 7-6

```
<UserControl x:Class="Cap7_2WCF.Page"
  xmlns:data="clr-namespace:System.Windows.Controls;
                        assembly=System.Windows.Controls.Data"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="495" Height="155">
  <Grid x:Name="LayoutRoot" Background="Navy" Height="Auto" Width="Auto" >
   <Grid.RowDefinitions>
```

```
      <RowDefinition Height="20" />
      <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <TextBlock Text="Listado de Clientes de Madrid" Foreground="AliceBlue"
       HorizontalA lignment="Center" VerticalAlignment="Center"
       Grid.Row="0" />
  <data:DataGrid x:Name="RejillaClientes" A lternatingRowBackground="Beige"
       AutoGenerateColumns="True" Width="Auto" Height="Auto"
       Grid.Row="1" Grid.Column="1" CanUserResizeColumns="True"
       Loaded="RejillaClientes_Loaded" BorderBrush="Navy"
       BorderThickness="3" />
</Grid>
   </UserControl>
```

Y, por último, el código de soporte del control Silverlight que hace la llamada (ver listado 7-7).

Aquí vemos la llamada asíncrona al servicio WCF. Una vez que la carga del control ha finalizado, se instancia un objeto de la clase **ServicioPedidosClient** (que se encuentra en el espacio de nombres **RefDatosPedidos**, definido en la ventana de descubrimiento del servicio Web).

Listado 7-7

```
private void RejillaClientes_Loaded(object sender, RoutedEventArgs e)
{
  RefDatosPedidos.ServicioPedidosClient ListadoClientes = new
            Cap7_2WCF.RefDatosPedidos.ServicioPedidosClient();
  ListadoClientes.ListadoClientesMadridCompleted += new EventHandler
     <Cap7_2WCF.RefDatosPedidos.ListadoClientesMadridCompletedEventArgs>
          (ListadoClientes_ListadoClientesMadridCompleted);
```

```
  ListadoClientes.ListadoClientesMadridA sync();
}
void ListadoClientes_ListadoClientesMadridCompleted(object sender,
    Cap7_2WCF.RefDatosPedidos.ListadoClientesMadridCompletedEventArgs e)
{
  RejillaClientes.ItemsSource = e.Result;
}
```

A continuación, se declara el manejador de evento para el proceso de llamada asíncrona, y se llama al servicio. Cuando el proceso concluye, el código pasa por el método **`ListadoClientes_ListadoClientesMadridCompleted`**, donde el resultado es recibido a través del parámetro **`e`**, que contiene la lista de clientes en su propiedad **`Result`**.

Esta vez, aprovechamos las características de la propiedad **`ItemsSource`** que ya comentamos más arriba. Basta con asignarle los resultados, y tendremos un conjunto de datos similar al de la figura 7-5, que representa la salida de la aplicación en Internet Explorer:

Listado de Clientes de Madrid

Ciudad	Empresa	IdCliente	Nombre	Provincia
Madrid	Repuestos Industriales "La tuerca"	5	Aitor Nillos	Madrid
Madrid	Fontaneros de Urgencia	54	Iñigo Teras	Madrid
Madrid	Clínica Dental "La clavada"	66	Fermín Cisivo	Madrid
Madrid	Radio "El Mitin"	77	Peter Tulias	Madrid

Figura 7-5. Salida del programa anterior.

Nótese que el proceso genera automáticamente las columnas de datos, lo que significa que las va a ordenar alfabéticamente si no le indicamos lo contrario.

El proceso es básicamente ése. Todas las opciones adicionales en el tratamiento de datos o en la obtención de resultados serán programadas en el servicio, reinterpretadas en la clase `proxy` que el IDE crea por nosotros, y utilizadas como sea necesario dentro de la interfaz de usuario.

Además, el `DataGrid` dispone de muchas otras posibilidades de personalización. Podemos permitir que el usuario reubique las columnas (como se hace en este caso), reordenar las filas seleccionando el nombre de la columna, permitir o no la edición directa en las celdas, y controlar todo el proceso de edición. Aparte, claro está, de todas las opciones de presentación y visualización que ya hemos visto en Silverlight 2. También es posible editar, y modificar las celdas de forma que incluyan otros objetos embebidos que hagan más sencilla la edición, por ejemplo, mostrando un `ComboBox` de las provincias disponibles al seleccionar una celda de la columna `Provincia`.

Acceso a datos mediante servicios Web SOAP (ASMX)

Otra opción de acceso a servicios consiste en llamar a un servicio ASMX clásico, basado en el protocolo SOAP. El proceso casi no requiere ninguna manipulación añadida por parte del programador, más allá de las tradicionales de este tipo de llamadas. Vamos a replicar lo hecho en capítulos anteriores para calcular el número del DNI y exponer esa funcionalidad como un servicio.

Para ello, en nuestro escenario clásico de Silverlight con su sitio de Web de pruebas, añadiríamos un servicio Web ASMX. Tras modificarlo para dar cobertura a la funcionalidad que queremos (en lugar del clásico "Hola

Mundo"), tendríamos una clase de servicio como la del código del listado 7-8.

Listado 7-8

```
using System;
using System.Text.RegularExpressions;
using System.Web.Services;

namespace Cap7_3SOA P.Web
{
  /// <summary>
  /// Un solo método que devuelve la letra del DNI a partir de su número
  /// </summary>
  [WebService(Namespace = "http://tempuri.org/")]
  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
  [System.ComponentModel.ToolboxItem(false)]
  public class ServicioDNI : System.Web.Services.WebService
  {
        string Letras = "TRWA GMYFPDXBNJZSQVHLCKET";
        Regex reNum = new Regex(@"^\d+$");

        [WebMethod]
        public string LetraDNI(string DNI)
        {
            bool esNumero = reNum.Match(DNI).Success;
            if (DNI != "" && esNumero)
            {
                DNI += " - " + Letras[Int32.Parse(DNI) % 23];
            }
            else { DNI = "Error en el número del DNI"; }
```

```
            return DNI;
        }
    }
}
```

No es necesario nada más para que esté accesible, salvo que se encuentre en un dominio distinto, cosa que comentaremos más adelante. Simplemente, desde la aplicación Silverlight, añadir una referencia Web, que será fácilmente descubierta por el mecanismo automático anterior. Con eso, dispondremos de un proxy válido para comunicarnos con el servicio. Nos queda ver cómo codificar la clase que efectúa la llamada.

Básicamente, declararemos una variable de instancia del servicio, y preparamos el entorno para una llamada asíncrona con un botón (`btnDNI`) que pasará al servicio el valor de un `TextBox (txtDNI)`, donde se introduce el número. Se declara el manejador de evento de forma similar al caso anterior, y, se efectuada la llamada.

En el manejador (`sDNI_LetraDNICompleted`), recibiremos el resultado (como antes) en la propiedad `Result` del segundo argumento de evento (`e`). El código tiene este aspecto, y funciona como cabe esperarse (nada especial para ver en pantalla, por lo que obviamos la salida):

Listado 7-9

```
using System;
using System.Windows;
using System.Windows.Controls;
using Cap7_3SOA P.RefServicioDNI;
namespace Cap7_3SOA P
```

```
{
  public partial class Page : UserControl
  {
    RefServicioDNI.ServicioDNISoapClient sDNI = new ServicioDNISoapClient();

    public Page() { InitializeComponent(); }

    private void btnDNI_Click(object sender, RoutedEventArgs e)
    {
        sDNI.LetraDNICompleted += new EventHandler
        <LetraDNICompletedEventA rgs>(sDNI_LetraDNICompleted);
        sDNI.LetraDNIA sync(txtDNI.Text);
    }

    void sDNI_LetraDNICompleted(object sender, LetraDNICompletedEventArgs e)
    {
        txtDNI.Text = e.Result;
    }
  }
}
```

Utilización de servicios REst

Últimamente, ha ganado importancia el modelo de servicios Web basados en características ya presentes en el protocolo HTTP[2]. Este protocolo define 4 posibles *verbos* o comandos que admite cuando recibe solicitudes remotas. De hecho, podíamos utilizar dos de ellos en servicios Web ASMX (GET y POST). Los otros dos

[2]. Para una explicación más completa del modelo ver (en inglés): **http://en.wikipedia.org/wiki/Representational_State_Transfer**.

son PUT y DELETE. Se ha aprovechado esta característica para hacer corresponder dentro del modelo cada uno de esos verbos con las 4 acciones principales que podemos solicitar de un servidor de datos (`SELECT`, `UPDATE`, `INSERT` y `DELETE`).

Pero se trata de un modelo de servicios un tanto particular en tanto que no existe un contrato real entre el que llama y el servicio que expone la funcionalidad. Al no existir tal modelo, la recuperación de la información no se produce mediante una clase `proxy`, como en los casos anteriores. En su lugar, utilizamos una clase capaz de acceder al contenido del sitio de forma directa. Los dos candidatos que ofrece el `CoreCLR` de Silverlight para esa acción son `WebClient` y `HttpWebRequest`. El primero, soporta los verbos GET y POST, mientras que el segundo permite una aproximación más granular al trabajo con REst.

Políticas de seguridad

Pero, el uso de servicios reales, fuera del dominio de la aplicación y sin presencia de un contrato, supone la existencia de unas políticas de seguridad establecidas en el servidor al que se realiza la petición. La mejor solución a este problema, se basa en la presencia de un fichero de servidor llamado `ClientAccessPolicy.xml`[3]. Ahí se establecen, al menos, qué tipo de peticiones se admiten, y qué tipo de clientes tiene garantizado el acceso. Un fichero típico de esta clase ofrece el siguiente aspecto:

[3]. También se puede utilizar un fichero CrossDomain.xml, pero este ofrece más flexibilidad.

Listado 7-10

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" /> <!-- Aquí puede indicarse el filtro de
                                dominios -->
      </allow-from>
        <grant-to>
             <resource path="/" include-subpaths ="true" />
             <!-- Y aquí, las ubicaciones permitidas para el sitio -->
        </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

El código XML anterior es expuesto por la fuente RSS que vamos a utilizar en nuestro ejemplo (el conocido sitio de blogs *Geeks.ms*) y cumple con estos requisitos. Se indica que se admiten peticiones desde cualquier origen (`<allow-from>`), y se permite el acceso a cualquier subdirectorio del sitio (`<grant-to>`), a partir de la raíz. Cualquier sitio que incluya esa definición, permite solicitar sus servicios desde Silverlight, y para comprobarlo, accederemos a una fuente RSS/ATOM que devuelve datos en formato XML, relativos a las entradas de sus blogs.

Nos basta con instanciar una clase `WebClient`, para realizar la llamada. El resultado, lo trataremos mediante un

objeto **XMLReader**, y accederemos a sus nodos con la ayuda de la clase **SyndicationFeed**, presente en **System.ServiceModel.Syndication**, y especialmente preparada para organizar la información de estos conjuntos de datos.

El código fuente de la clase quedaría de esta forma:

Listado 7-11

```
using System;
using System.Net;
using System.ServiceModel.Syndication;
using System.Windows.Controls;
using System.Xml;

namespace Cap7_3SOA P
{
  public partial class RSS : UserControl
  {
    WebClient wcRSS = new WebClient();
    public RSS()
    {
      InitializeComponent();
      wcRSS.OpenReadCompleted += new
        OpenReadCompletedEventHandler(wcRSS_OpenReadCompleted);
      wcRSS.OpenReadA sync(new Uri("http://geeks.ms/ blogs/mainfeed.aspx"));
    }
    void wcRSS_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
    {
      try
      {
        XmlReader xrdr = XmlReader.Create (e.Result);
```

```
      SyndicationFeed feed = Syndication Feed.Load(xrdr);

      txtRSSConectado.Text = feed.Title.Text;

      foreach (SyndicationItem si in feed. Items)
      {
           txtRSS.Content += "Fecha: " + si.PublishDate + "\n";
           txtRSS.Content += "? " + si.Title.Text + "\n";
      }
      xrdr.Close();
    }
    catch { txtRSS.Content= "Error en acceso al origen RSS"; }
  }
 }
}
```

Cuando la llamada al método **OpenRead- Async** termina, lo que recibimos en **Open -ReadCompleted** es un *stream* en formato XML. Tomamos el dato del *feed* principal para usarlo como título, asignando el valor al **TextBlock txtRSSConectado** y situamos la salida correspondiente dentro de un **ScrollViewer** (**txtRSS**), donde, para cada entrada, vamos añadiendo la fecha y el título de la publicación. En diseño (trivial, por lo que obviamos el listado) ambos elementos de la interfaz de usuario estarán vacíos.

El lector puede comprobar mediante un punto de ruptura la estructura de datos devuelta, y diseñar sus propios lectores de información RSS con ayuda de las dos clases que hemos utilizado (**XmlReader** y **SyndicationFeed**).



Figura 7-6. Salida del código anterior.

ADO.NET Data Services

Anteriormente llamado `Proyecto Astoria`, **ADO.NET Data Services** (ADS) consiste en una combinación de patrones y librerías que permiten la fácil creación y consumo de servicios Web. Los usuarios encontrarán todo lo necesario para trabajar con ellos si tienen instalado el Service Pack 1 de Visual Studio 2008. El objetivo es permitir la creación de servicios de datos flexibles y que se integren en la Web de forma natural.

Como consecuencia de ello, ADS utiliza sintaxis URI para referirse a los datos, y formatos sencillos y bien conocidos, tales como *JSON* y *ATOM* para representarlos. Esto supone que la arquitectura final se ofrece en un modelo de tipo *REst*, con el que los gestores de acceso a datos pueden interactuar usando los verbos estándar

HTTP (GET, POST, PUT y DELETE), como apuntábamos antes.

Al objeto de modelar esos datos expuestos, ADS utiliza un derivado del modelo entidad/relación llamado **Entity Data Model** (**EDM**), que organiza los datos en entidades y establece relaciones entre ellos.

ADS y los datos relacionales

Para los datos de tipo relacional, (pues se puede acceder a diversos orígenes), ADS expone un modelo construido a partir del **ADO.NET Entity Framework**[4], que funciona como una capa complementaria del servicio, suministrando toda la maquinaria para el tratamiento de datos. Para los otros tipos, o, en caso de que queramos utilizar otras tecnologías de acceso, como *LINQ-to-SQL*, se suministra un mecanismo que permite modelar cualquier origen de datos en forma de entidades y asociaciones, para poder exponerlas como servicios de datos siguiendo el criterio anterior.

Esto supone que cualquier aplicación que pueda acceder a servicios Web puede acceder a los datos expuestos de esta forma. Vamos a ver su funcionamiento con un ejemplo basado en la base de datos que hemos venido utilizando hasta aquí.

Ejemplo de ADO.NET Data Services

La primera fase es muy similar a la que seguimos en la creación del servicio WCF.

---

[4] .Para una completa introducción al tema, ver `http://msdn.micro soft.com/en-us/library/aa697427(VS.80).aspx.`

- Creamos una aplicación Silverlight 2, con un sitio Web de pruebas.
- Montamos el modelo de base de datos sobre el sitio Web creando un nuevo ADO.NET Entity Data Model. Indicaremos que queremos crearlo a partir de una base de datos existente, y conectamos con el origen de datos siguiendo el Asistente de conexión y poniendo los nombres adecuados.
- Llegados al punto de selección de las entidades de datos, marcaremos todas las tablas, vistas y procedimientos almacenados necesarios para el proceso (figura 7-7).
- Terminamos esa operación, tras lo cual se genera una clase que representa a todas las entidades presentes en el modelo. Se abre la ventana del Diseñador del Modelo de entidades, para hacer el ajuste fino que fuera necesario (figura 7-8).

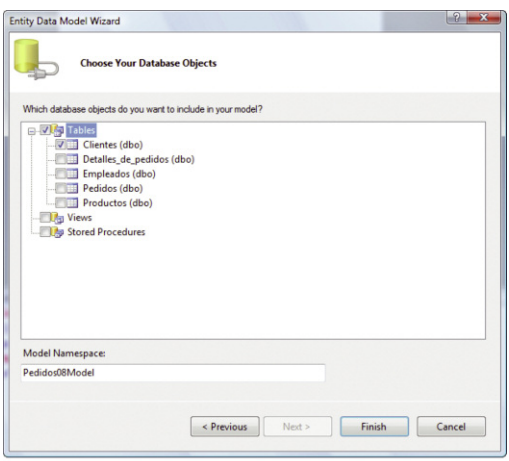


Figura 7-7. Selección de las entidades de trabajo para el modelo de datos.

La clase generada hereda de `ObjectContext`, y de forma similar a los antiguos `DataSets`, contendrá clases que "mapearán" todas las entidades presentes en el modelo, y ofrecerán los servicios necesarios para las operaciones CRUD sobre ese modelo.

Lo que resta en la parte del servidor, es generar el servicio que expondrá estos datos al mundo exterior. Ahí es donde interviene ADO.NET Data Services en lugar de los mecanismos anteriores. *Los datos expuestos así, serán accesibles de forma directa desde el navegador utilizando una sintaxis especial tipo URI propuesta por esta tecnología (sin necesidad de ejecutar la aplicación)*.

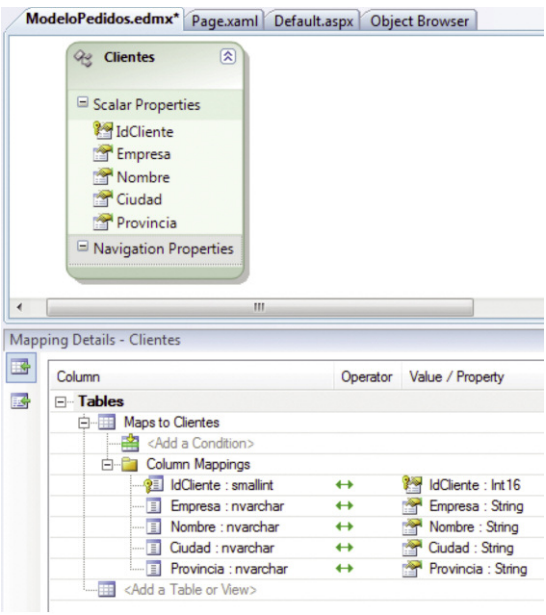


Figura 7-8. Ventanas del Diseñador de Modelos de Entidades.

Añadimos, por tanto, un nuevo servicio ADO.NET Data Service del apartado Web de las plantillas disponibles, al que llamamos **adsPedidos.svc**. Este servicio consta de una clase que es el único punto a retocar a mano en esta fase: la clase **adsPedidos**, que hereda de **DataService** y a la que indicamos cuál es la entidad con la que queremos operar. Además, en su método **InitializeService**, deberemos expresar cuáles de las entidades disponibles exponemos en este momento, y los permisos de acceso. El código resultante será el del listado 7-12.

Y eso es todo por parte del servidor. Lo novedoso de este modelo es que podemos comprobar si funciona en este mismo momento sin necesidad de la aplicación cliente, ya que es posible interrogar al servicio utilizando una sintaxis de este tipo desde el navegador:

```
http://localhost:puerto/Servicio/Entidad?Parámetros
```

Podemos abrir un navegador y probar algo como **http://localhost:1446/adsPedidos.svc/ Clientes**, obteniendo una respuesta del servicio en texto plano, que cada navegador formateará como crea conveniente, pero que contendrá los datos solicitados (podemos comprobarlo viendo el código fuente generado como respuesta).

Listado 7-12

```
using System.Data.Services;
namespace Cap7_A DS.Web
{
  public class adsPedidos : DataService<Pedidos08Entities>
```

```
 {
   // Este método se llama una sola vez para inicializar políticas
                                de servicio
   public static void InitializeService(IDataServiceConfiguration config)
   {
      config.SetEntitySetA ccessRule("Clientes", EntitySetRights.AllRead);
      config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);
   }
 }
}
```

Comprobado el funcionamiento, la parte de cliente resulta trivial, y es similar a las otras que hemos creado. Solo hay que recordar que deberemos seleccionar la llamada asíncrona al servicio. Lo primero es hacer la referencia al servicio. Al igual que antes, será descubierto fácilmente por el asistente. En este caso, lo denominamos **`ServicioClientes`**.

La interfaz de usuario también es sencilla, porque solo nos interesan los datos y su acceso. Un **`ListBox`** puede bastar. Ahora bien, debemos de declarar los elementos que van a aparecer en el **`ListBox`** dentro de su plantilla, creando un modelo para los **`ListBoxItem`** e indicando los enlaces de datos (*Binding*) adecuados. Una interfaz similar a esta es suficiente para el caso:

Listado 7-13

```
<UserControl xmlns:data=
"clr-namespace:System.Windows.Controls;assembly= System.Windows.Controls.Data"
   x:Class="Cap7_4A DODS.Page"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Width="400" Height="300">
<Grid x:Name="LayoutRoot" Background="White">

    <ListBox x:Name="ListaCtes" Background="Beige"
         HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    ItemsSource="{Binding Mode=OneWay}" Margin="8,8,8,10" >
    <ListBox.ItemTemplate>
        <DataTemplate>
        <StackPanel x:Name="Listado" Orientation="Horizontal"
        VerticalAlignment="Bottom" Margin="5" >
          <TextBlock x:Name="IdCliente" Text="{Binding Path=IdCliente}"
          Margin="5,0,0,0" VerticalAlignment="Bottom"
          HorizontalAlignment="Left" FontSize="12"/>

          <TextBlock x:Name="Empresa" Text="{Binding Path=Empresa}"
          Margin="5,0,0,0" VerticalAlignment="Bottom"
          HorizontalAlignment="Left" FontSize="12"/>

          <TextBlock x:Name="Nombre" Text="{Binding Path=Nombre}"
          Margin="5,0,0,0" VerticalAlignment="Bottom"
          HorizontalAlignment="Left" FontSize="12"/>

          <TextBlock x:Name="Ciudad" Text="{Binding Path=Ciudad}"
          Margin="5,0,0,0" VerticalAlignment="Bottom"
          HorizontalAlignment="Left" FontSize="12"/>
        </StackPanel>
      </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

```
  </Grid>
</UserControl>
```

Como vemos, cada ítem del **ListBox** tiene definida una plantilla de datos con 4 **TextBlocks**, para los 4 campos de la tabla **Clientes**. Se establece el *binding* con el nombre de cada columna y se nombra el **ListBox** (**ListaCtes**).

El último paso, es hacer la llamada al servicio desde la clase **Page**. Primero, hacemos referencia al **ServicioClientes**, generado antes. En el constructor, declaramos el *objeto de contexto* y le indicamos que su destino es una URI que apunta al servicio. A continuación, realizamos la llamada asíncrona, declarando el método que recogerá los datos cuando ésta haya concluido.

En el método receptor, recuperaremos el objeto de contexto, para que realice la consolidación de la llamada, y asignamos al **ListBox** los datos obtenidos. El código fuente completo de la clase **Page** es el siguiente:

Listado 7-14

```
using System;
using System.Data.Services.Client;
using System.Windows.Controls;
using Cap7_A DS.ServicioClientes;

namespace Cap7_A DS
{
  public partial class Page : UserControl
  {
    public Page()
```

```
  {
    InitializeComponent();
    DataServiceContext ctxCtes = new
      DataServiceContext( new Uri("adsPedidos.svc", UriKind.Relative));

    ctxCtes.BeginExecute<Clientes>(new Uri
                              ("/Clientes", UriKind.RelativeOrAbsolute),
                                  RellenarClientes, ctxCtes);
  }

  private void RellenarClientes(IAsyncResult Resultado)
  {
    DataServiceContext ctxLlamada = Resultado.AsyncState as
                        DataServiceContext;
    ListaCtes.DataContext = ctxLlamada.EndExecute<Clientes>(Resultado);
  }
 }
}
```

Y la salida, muy sencilla, se muestra en la figura 7-9.



Figura 7-9. Salida del ejemplo de ADO.NET Data Services.

DataBinding con Expression Blend 2.0

El proceso de construcción de enlaces de datos puede realizarse utilizando Expression Blend 2, si bien es preciso tener en cuenta algunos aspectos de su integración con Visual Studio. Dentro de Blend, podemos utilizar las ventanas de edición para establecer los vínculos necesarios para todos los casos. Vamos a analizar los procesos requeridos para el enlace a datos por esta vía.

Especialmente, nos interesa la forma de establecer el enlace a datos entre objetos complejos y controles de listas, y cómo usar Expression Blend para la creación de *binding*. Uno de esos casos típicos —que vamos a ilustrar aquí— es el de los listados con una lista de ítems en un control `ListBox` al estilo de lo visto en el ejemplo anterior.

Vinculación a objetos de negocio

Vamos a empezar con el enlace de elementos de la interfaz de usuario con objetos de negocio.

> **NOTA**. La posibilidad de leer información en formato XML no está implementada todavía, y —en esta versión, al menos— la solapa correspondiente se encuentra deshabilitada, desde aplicaciones Silverlight. Si se quiere acceder a datos en formato XML se debe utilizar la técnica usada para los servicios REst —un objeto WebRequest, junto a un XmlReader o StreamReader, y si se necesita cargar recursos de los paquetes XAP en ese formato, el CoreCLR suministra la clase XmlXapResolver, con ese propósito[5].

Supongamos que queremos realizar la vinculación mediante Blend 2.0 en el ejemplo del visor de datos de libros. Como se trata de un conjunto manejable de información

---

[5]. Puede ver un ejemplo de esta técnica en MSDN: `http://msdn.microsoft.com/en-us/library/cc645034(VS.95).aspx`

(9 ítems[6]), decidimos ubicar los datos en una clase de negocio separada, de forma que al instanciarla, suministre todos los métodos de acceso a datos.

Podemos reutilizar la clase **Libro** para que albergue la información correspondiente a esos 3 campos y permita el acceso con propiedades públicas. Esta clase, será manejada por otra que hará las veces de gestor (**GestorLibros**), que se encargará de devolver una lista de libros con toda la información. Resumiendo: una clase con 3 campos que representa la entidad de negocio y una clase manejadora con un método que devuelve la lista de objetos **Libro**. El aspecto del código C# de esa clase sería el siguiente:

Listado 7-15

```
public class GestorLibros : INotifyPropertyChanged
{
   public event PropertyChangedEventHandler PropertyChanged;
   ObservableCollection<Libro> _Lista;
   public ObservableCollection<Libro> ListaDeLibros //Expone los datos
   {
       get
       {
           if (_Lista.Count < 1) ObtenerLibros();
           return _Lista;
       }
       set
```

[6] Los publicados por .netalia, hasta el momento.

```
    { _Lista = value;
        if (PropertyChanged != null) PropertyChanged(this, new
                PropertyChangedEventArgs("ListaDeLibros"));
        }
    }
    // Constructor personalizado
public GestorLibros()
{
    _Lista = new ObservableCollection<Libro>();
    if (HtmlPage.IsEnabled == false)
    {
        ObtenerLibros();
    }
}

public void ObtenerLibros()
{
  _Lista = new ObservableCollection<Libro>();
  // Inicializa manualmente los valores de 3 libros de la _Lista.
  _Lista.Add(new Libro());
  _Lista[0].Autor = "Pep Lluis Baño";
  _Lista[0].Titulo = "Robot dispensador para MSDN video";
  _Lista[0].Existencias = 10;

  _Lista.Add(new Libro());
  _Lista[1].Autor = "Luis Miguel Blanco";
  _Lista[1].Titulo = "Diseño de Informes con SQL Reporting Services";
  _Lista[1].Existencias = 30;

  _Lista.Add(new Libro());
  _Lista[2].Autor = "Marino Posadas";
```

```
_Lista[2].Titulo = "Programación segura con .NET Framework";
_Lista[2].Existencias = 20;

_Lista.Add(new Libro());
_Lista[3].Autor = "Pepe Hevia";
_Lista[3].Titulo = "Microsoft SharePoint Products & Technologies";
_Lista[3].Existencias = 200;

_Lista.Add(new Libro());
_Lista[4].Autor = "Braulio Díaz y Reyes García";
_Lista[4].Titulo = "Optimización de A SP.NET 2.0";
_Lista[4].Existencias = 50;

_Lista.Add(new Libro());
_Lista[5].Autor = "Gustavo Vélez";
_Lista[5].Titulo = "Programación con SharePoint 2007";
_Lista[5].Existencias = 0;

_Lista.Add(new Libro());
_Lista[6].Autor = "Miguel Katrib, et. al.";
_Lista[6].Titulo = "Windows Presentation Foundation";
_Lista[6].Existencias = 70;

_Lista.Add(new Libro());
_Lista[7].Autor = "Grupo Weboo";
_Lista[7].Titulo = "Visual Studio 2008";
_Lista[7].Existencias = 120;

_Lista.Add(new Libro());
_Lista[8].Autor = "Marino Posadas";
_Lista[8].Titulo = "Programación con Silverlight 2.0";
```

```
        _Lista[8].Existencias = 5;


        }
}
public class Libro : INotifyPropertyChanged
// (Ver implementación en el apartado "DataBinding en Silverlight 2"
// al principio de este capítulo)
```

Una vez codificada la clase de negocio, podemos volver a Blend para concluir la programación y realizar en enlace a datos adecuado. En el código anterior se ve cómo **GestorLibros** da acceso general a los mecanismos de construcción de datos, y **Libro**, representa cada libro individual. El enlace a datos desde Blend puede realizarse para ambas clases y la herramienta reconoce cualquier objeto de este tipo (o del *runtime*).

Enlazando a objetos de negocio

Teniendo seleccionada la ventana de gestión del proyecto, Fijémonos en la sub-ventana inferior con el título general **Data**. Una de las solapas indica **CLR Object**, (un objeto de negocio programado como una clase estándar de .NET). Si marcamos esta opción accedemos a la ventana de la figura 7-10, en la que hemos seleccionado (aunque no se necesitaba) la opción "*Show System Assemblies*" (mostrar objetos del sistema), para tener una idea más completa de su oferta.

Una vez ahí, seleccionamos nuestra clase **GestorLibros**, disponible en el árbol de la aplicación, y aceptamos. Veremos que el objeto pasa a formar parte de la lista de elementos accesibles desde nuestra interfaz de usuario.

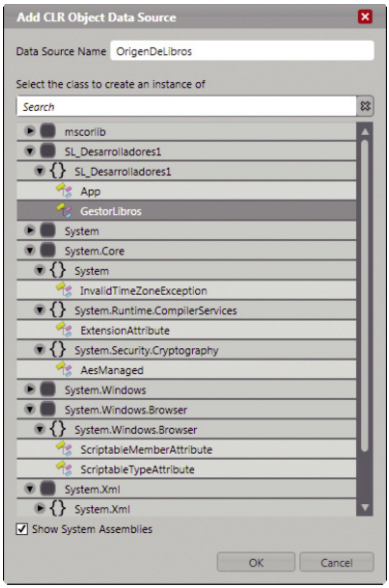


Figura 7-10. Ventana de selección de objetos de negocio.

Lo que sucede es que se han generado automáticamente varias referencias en el código XAML: la declaración del control de usuario incluye ahora nuevos espacios de nombres y se ha añadido al control la referencia a nuestra clase de negocio. Los cambios aparecen en el fragmento de código siguiente:

Listado 7-16

```
<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"
   xmlns:SL_Desarrolladores1="clr-namespace:SL_Desarrolladores1"
   x:Class="SL_Desarrolladores1.Page"
   Width="440" Height="480" mc:Ignorable="d" >
<UserControl.Resources>
   <SL_Desarrolladores1:GestorLibros x:Key="GestorLibrosDS"
                  d:IsDataSource="True"/>
</UserControl.Resources>
```

Los elementos expuestos por la clase ya están disponibles para procesos de *binding* desde los objetos de la interfaz de usuario. De hecho, si desplegamos el objeto **`GestorLibros`** en la Ventana de datos, veremos que ofrece un array de **`Libros`** que se expone como una **`ObservableCollection`**.

NOTA. Una colección de este tipo es una novedad aportada por Silverlight para que los objetos que se utilicen en el enlace a datos sean capaces de notificar a la interfaz de usuario cuando ha sucedido un cambio en su contenido.

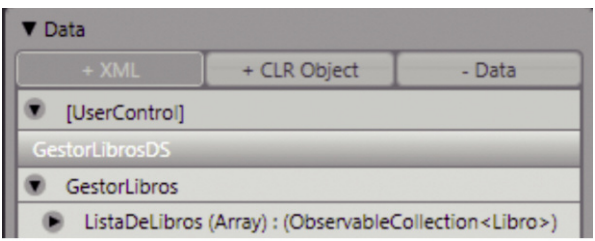


Figura 7-11. Referencia a la librería de GestorLibros desde la ventana de datos.

Ahora, podemos optar por una opción bien sencilla: arrastrar la colección de elementos que nos interesa mostrar sobre un control de la interfaz de usuario, al estilo

de lo que hacíamos en algunos escenarios de Windows Forms. Si lo hacemos así, veremos que se nos ofrece un menú contextual de enlace con dos categorías (eso depende siempre del tipo de datos a enlazar y del control que va a servir de destino del enlace):

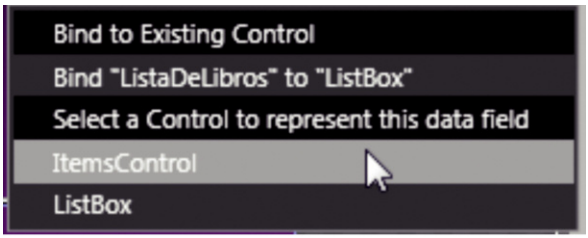


Figura 7-12. Menú contextual de enlace a datos en Blend.

Podemos escoger la opción predeterminada ("ListaDeLibros to ListBox") o seleccionar otro control adecuado. El menú contextual tiene dos espacios: el predeterminado ("Bind to Existing Control") y otras posibles opciones que pueden tener sentido en este proceso ("Select a control to…").

Debido a que no hemos indicado qué elemento de los expuestos por la clase `Libro` nos interesa mostrar, en la Ventana de diseño solo aparece el nombre del objeto dentro del control que lo contiene, pero ha detectado el número de objetos disponibles en la colección (los 9 ítems), con lo que muestra una interfaz de usuario como la de la figura 13 (si ejecutamos la aplicación la salida por pantalla sería idéntica).

Lo que tenemos que hacer a continuación es crear una plantilla para que el control `ListBox` muestre correc-

tamente la información que nos interesa de cada libro. Para ello, en la opción de menú "Object -> Edit Other Templates -> Edit Item Template -> Create Empty", creamos una nueva plantilla para estos datos.

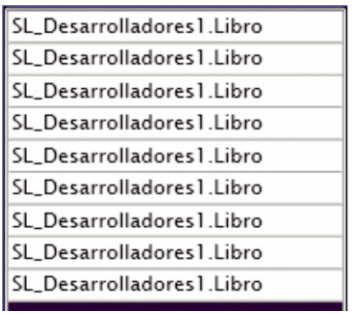


Figura 7-13. ListBox mostrando los objetos libro enlazados sin especificar más información.

Nos aparecerá una caja de diálogo titulada "Crear plantilla de recurso de datos" ("Create DataTemplate Resource"), donde podemos dar un nombre a la plantilla y especificar si queremos que ese recurso sea parte de toda la aplicación (para ser utilizado por otros controles) o preferimos mantenerlo como un recurso restringido al control que estamos construyendo. Optamos por mantenerlo restringido a este control, como se ve en la figura 7-14:

En ese momento, la Ventana "Objects & Timeline" cambia, y se nos presenta como si estuviéramos editando un ítem completamente nuevo. Nos genera un elemento `ItemTemplate` que contiene un `Grid` vacio y a partir de ahí podemos empezar a definir los contenidos.

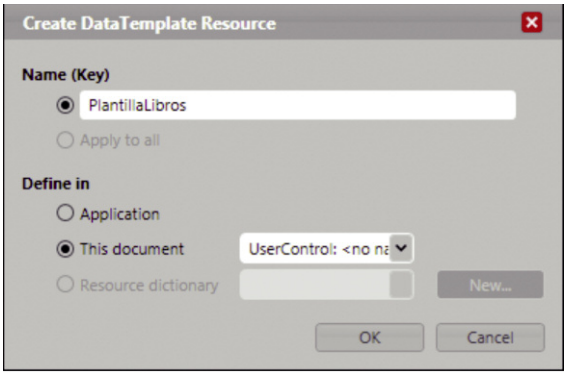


Figura 7-14. Ventana de creación de recursos de plantillas de datos.

Con ese `ItemTemplate` seleccionado, es momento de construir la interfaz de usuario de cada elemento enlazado al `ListBox`. Por ejemplo, podemos usar un `StackPanel` con dirección horizontal, e incluir en cada uno de ellos un `TextBlock` que represente a los 3 campos disponibles en la clase de negocio. Una vez codificada esta parte, se supone que el código XAML completo de la aplicación (eliminadas las imágenes), será similar al siguiente:

Listado 7-17

```
<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"
  xmlns:SL_Desarrolladores1="clr-namespace:SL_Desarrolladores1"
     x:Class="SL_Desarrolladores1.Page"
     Width="440" Height="480" mc:Ignorable="d" >
```

```
<UserControl.Resources>
  <SL_Desarrolladores1:GestorLibros x:Key="GestorLibrosDS"
                        d:IsDataSource="True"/>
    <DataTemplate x:Key="Plant_Libros">
      <Grid>
        <StackPanel x:Name="StackContenedor"
                      Orientation="Horizontal" Width="400">
          <TextBlock Height="Auto" Width="220" TextWrapping="Wrap"
             x:Name="Titulo" HorizontalAlignment="Left"
             VerticalAlignment="Bottom" Text="{Binding Mode=OneWay,
             Path=Titulo, Source={StaticResource LibroDS}}"
             FontFamily="Comic Sans MS" FontWeight="Bold" FontSize="14"/>
          <TextBlock Height="Auto" Width="129" TextWrapping="Wrap"
             Text="{Binding Mode=OneWay, Path=Autor,
             Source={StaticResource LibroDS}}" x:Name="Autor"/>
          <TextBlock Height="Auto" Width="30" TextWrapping="NoWrap"
             Text="{Binding Mode=OneWay, Path=Existencias,
             Source={StaticResource LibroDS}}"
             HorizontalAlignment="Right" TextAlignment="Right"
             FontWeight="Normal" x:Name="Existencias"
             Foreground="#FF1A 46D3"/>
          </StackPanel>
          </Grid>
      </DataTemplate>
      <SL_Desarrolladores1:Libro x:Key="LibroDS" d:IsDataSource="True"/>
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot" >
      <Grid.ColumnDefinitions>
         <ColumnDefinition Width="496"/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
```

```
        <RowDefinition Height="83.179"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="0*"/>
      </Grid.RowDefinitions>
    <Border HorizontalAlignment="Stretch" Margin="32,4,104,4" x:Name="Borde"
      Grid.ColumnSpan="1" Background="#FF4C2F88" BorderBrush="#FF6A B6B9"
      BorderThickness="2,2,2,2" CornerRadius="20,20,20,20" ‘
      VerticalAlignment="Stretch"/>
    <TextBlock HorizontalAlignment="Stretch"
      Margin="48,24,104,22.2999992370605" x:Name="Titulo"
      VerticalAlignment="Stretch" Grid.ColumnSpan="1" Grid.RowSpan="1"
      FontFamily="Lucida Sans Unicode" FontWeight="Bold"
      Foreground="#FFF0E2E2" Text="Visor de datos de libros"
      TextWrapping="Wrap" d:LayoutOverrides="Height, GridBox"
      TextAlignment="Center" FontSize="24" />
    <ListBox Margin="16,15,72,23.8570003509521" VerticalAlignment="Stretch"
      Grid.Row="1" HorizontalAlignment="Stretch"
      ItemsSource="{Binding Mode=OneWay, Path=ListaDeLibros,
                    Source={StaticResource GestorLibrosDS}}"
      ItemTemplate="{StaticResource Plant_Libros}"/>
  </Grid>
</UserControl>
```

O sea, dentro de los recursos del control de usuario se hace una referencia al objeto de negocio y se crea una plantilla de datos (**DataTemplate**) que enlaza directamente con la clase que expone la información (**GestorDatos**). Los bloques de texto destinados a mostrar datos vinculan su propiedad **Text** con el recurso concreto (que aquí se llama **LibroDS**), y dentro de él, seleccionan (atributo **Path**), el campo a mostrar.

En ese momento, podemos ejecutar la aplicación o refrescar los datos. El editor visual de Blend nos ofrecerá el aspecto final de la aplicación, que será similar al de la figura 7-15.



Figura 7-15.Ventana de diseño de Blend, mostrando los datos enlazados de la clase GestorLibros.

Por tanto, Blend 2.0 permite a los diseñadores trabajar con clases de negocio y tener una aproximación real al aspecto que ofrecerán los datos en pantalla, favoreciendo el trabajo conjunto con los desarrolladores al aportar su visión de la aplicación con un punto de vista más real que en solo diseño. Y en el caso de confluir ambas tareas en una persona, el manejo de Blend con clases de acceso a datos, puede darnos una idea de qué información merece ser resaltada o tratada en forma especial.

NOTA. Para más opciones de manipulación del DataGrid en Silverlight no cubiertas en esta obra véase el blog de Scott Morrison (co-autor del control) en **`http://blogs.msdn. com/scmorris`** y en castellano, el blog "El aprendiz de brujo" (**`http://geeks.ms/blogs/lmblanco`**) de Luis Miguel Blanco.

8

Arquitectura y distribución de aplicaciones

Arquitectura de aplicaciones Silverlight: recomendaciones

Los departamentos de divulgación tecnológica de Microsoft están realizando un gran esfuerzo para asesorar a empresas que quieren adoptar esta tecnología, de forma que puedan seguir patrones de buenas prácticas desde un principio. De acuerdo con estas recomendaciones, hay una serie de consejos que debieran tenerse en cuenta desde un inicio para la óptima conclusión del ciclo de vida completo en una aplicación Silverlight.
Y todo ello, considerando la arquitectura de las aplicaciones RIA como un subconjunto de las aplicaciones de N-Niveles (*N-Tier*) y/o N-Capas (*N-Layer*). En general, al hablar de *N-Tier* nos referimos al nivel físico, mientras que por *N-Layer* entendemos el nivel lógico, por lo que las primeras deberían estar diseñadas a nivel lógico mediante las segundas. Eso nos lleva al marco tecnológico en el que queremos desarrollar nuestra aplicación: esto es, qué tecnologías estarán implicadas en la construcción.

Marco tecnológico

Podemos pensar en 3 aproximaciones distintas para la utilización de Silverlight en sitios Web:

1) Inclusión de "islas" Silverlight. Normal-mente, reduce la utilización de Silverlight a elementos puntuales, como anuncios publicitarios o complementos visuales de páginas para enriquecer la experiencia de usuario. Se trabaja con elementos Silverlight individuales vinculados a etiquetas `<div>` en la(s) página(s) correspondientes.

   Se pueden utilizar técnicas AJAX para que la carga se produzca de forma asíncrona y dinámica. En general, no es la mejor aproximación para la construcción de aplicaciones RIA.
2) La página entera es un control de usuario único. El usuario crea su propio sistema de navegación, dependiendo de los patrones utilizados, y se va cargando y descargando el control principal con las páginas individuales. Ideal para aplicaciones RIA estándar y/o de medio volumen.
3) Si se trata de aplicaciones complejas, se recomienda la utilización de marcos de desarrollo, como PRISM V.2 (conjunto de patrones para aplicaciones Silverlight 2, disponible en "*Patterns & Practices*"). PRISM se ha creado desde cero para WPF y Silverlight 2, no es una migración. El modelo aparece representado en la figura 8-1:

Para más información sobre el estado de PRISM v2.0, descargas y documentación, visitar el sitio de CodePlex

http://www.code plex.com/CompositeWPF, así como el blog de **Blaine Wastell** (http://blogs. msdn.com/blaine/archive/2008/09/02/scope-for-prism-2-0.aspx).

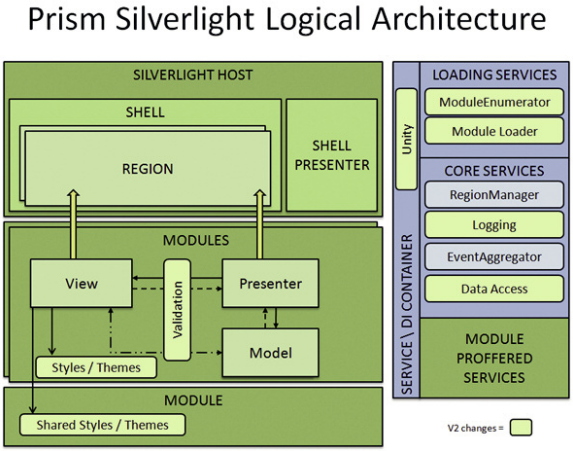


Figura 8-1.Modelo arquitectónico propuesto por PRISM v.2 para aplicaciones Silverlight 2.0.

No obstante, e independientemente del marco elegido, hay un montón de buenas prácticas aplicables en la construcción de una aplicación Silverlight. Algunas de ellas las comentamos a continuación.

Buenas prácticas en la construcción de aplicaciones Silverlight 2

Son muchas las áreas donde aplicar este tipo de consejos para el ciclo de vida de una aplicación RIA. Comentaremos alguna de las más importantes[1]:

[1].Muchas de estas ideas están tomadas de los blogs de César de la Torre (http://blogs.msdn.com/cesardelatorre), Josh Holmes (http://joshholmes.com), Scott Guthrie (http://weblogs.asp.net /scottgu) y Michael Sync (http://michaelsync.net).

- Capas de presentación
- Controles
- Media y gráficos
- Navegación
- Gestión de estados
- Validación de datos
- Cachés
- Accesibilidad
- Comunicaciones entre capas
- Seguridad
- Acceso a datos

Capas de presentación

Se recomienda tener presentes los modelos de presentación actuales: Modelo-Vista-Controlador (**M-V-C**), Modelo-Vista-Presentación (**M-V-P**), y Modelo-Vista-View-Model (**M-V-VM**). El patrón MVP se presta mejora para las aplicaciones RIA, donde, si la salida es muy compleja, se pueden implementar múltiples presentadores.

Controles

Ya hemos visto los controles que incorpora Silverlight "de fábrica" y también comentábamos las últimas extensiones de controles y herramientas disponibles a través de CodePlex. Todos ellos son modificables y adaptables por el usuario y pueden servir igualmente de base para la construcción de controles más complejos. No obstante, no debemos olvidar la ya abundante oferta de terceros: ComponentOne, Infragistics, Xceed, Telerik, DevExpress y otros anuncian soluciones profesionales y hay muchos otros gratuitos

como proyectos de código comunitario tanto en CodePlex, como en sitios similares.

Media y gráficos

Hay varias recomendaciones respecto a esto, y será una decisión individual dependiendo de las necesidades de cada aplicación. Pero el uso del *streaming adaptativo*, para la descarga de vídeos en el navegador es una de las prácticas recomendadas para solucionar problemas de ancho de banda. De igual forma, se fomenta la carga de imágenes y vídeos de forma asíncrona desde fuera del fichero XAP.

La parte gráfica que no pueda ser resuelta de forma vectorial (o sea, si se requiere de imágenes que tienen que ser incluidas en el XAP por razones de presentación inicial), la sugerencia es utilizar el formato PNG, dado que, además de ser un estándar oficial W3C, su relación entre tamaño y calidad es de las mejores (y recuerde que el formato GIF no está soportado)[2].

Navegación

El consejo en este punto es claro: las aplicaciones RIA ofrecen una mejor experiencia de usuario cuando hay una sola interfaz central, basada íntegramente en Silverlight. Esto presenta potenciales problemas de control del flujo (interacciones del usuario navegando hacia atrás-adelante, etc.). Por ello se recomienda capturar y anular estos botones del navegador para limitar la navegación a nuestro sistema Silverlight.

---

[2].No obstante, Joe Stegman, dispone en su blog (`http://blogs .msdn.com/jstegman`) de interesantes aportaciones para la conversión automática a otros formatos.

Una opción adicional es interceptar la URI y gestionar manualmente el histórico del navegador (por ejemplo, usando el objeto **History** de AJAX). El código siguiente, basado en unas ideas de **Josh Holmes**, ilustra este punto.

De igual forma, el proceso inicial siempre es importante de cara a la experiencia de usuario. Se debiera de mostrar una metáfora de progreso según se está produciendo la carga del control, y que esta sea lo suficientemente visible y fiable (evitemos el mal gusto de pasarse la mitad del tiempo en el 99%, haciendo pruebas de ancho de banda).

Listado 8-1

```
private void Application_Startup(object sender, StartupEventArgs e)
{
   string startPageParameter = "/PaginaInicial";
   if (!e.InitParams.ContainsKey(paramPagInicial))
   {
       this.RootVisual = new PaginaInicialPredeterminada();
   }
   Else
   {
       switch (e.InitParams[paramPagInicial])
       {
         case "PaginaInicialPredeterminada":
           this.RootVisual = new PaginaInicialPredeterminada(); break;
         case "PaginaInicialNoPredeterminada":
           this.RootVisual = new PaginaInicialNoPredeterminada(); break;
         default:
           throw new Exception("/PaginaInicial debe ser " +
```

```
            "'PaginaInicialPredeterminada' o " +
            "'PaginaInicialNoPredeterminada'.");
        }
    }
}
```

Gestión de estados

Si, por la razón que sea, se necesita guardar el estado de la aplicación, distinga primero con claridad qué estado: el del cliente o el del servidor. Si es el del cliente, lo recomendable es el uso de Almacenamiento aislado (*Isolated Storage*), para mantener el estado entre sesiones, comprobando siempre la cuota de disco antes de grabar. Se puede sincronizar con el servidor cada cierto tiempo o al final de la sesión.

Si se trata de guardarlo en el servidor, mejor que suceda solo cuando sea imprescindible, y usando un mecanismo seguro, como una base de datos.

Validación de datos de entrada

Silverlight 2.0 no presenta un modelo interno de validación, por lo que ésta debe hacerse en el cliente o vía servicios Web. Las recomendaciones son:

- Identificar claramente las *fronteras* entre capas y validar solamente aquellos datos que las crucen.
- Si la validación se hace en cliente, el código, mejor separado (incluso de las clases de negocio).
- Recuerde los consejos generales de cargas parciales cuando el tratamiento es voluminoso, no muestre por pantalla las entradas no fiables y mantenga la validación en servidor solamente para los datos críticos.

Cachés

El cache del navegador será idóneo para los datos que no cambian durante la sesión. El almacenamiento aislado no debe usarse para objetos BLOB (objetos binarios grandes, como gráficos pesados o vídeos), y debemos comprobar la cuota antes de escribir, además de llevar un control de errores.

Accesibilidad

Mientras que en la versión 1.0 apenas estaba presente, aquí la situación ha cambiado de forma radical. Por un lado, cuanta más presencia de información vectorial, mejor, ya que puede escalarse cuanto sea necesario sin pérdida de calidad. Por otro lado, la versión 2.0 presenta las siguientes novedades:

- Soporte completo de teclado (foco y entradas, además de control de *Tabs*).
- Inclusión de información de accesibilidad en los elementos XAML, a través de las `AutomationProperties` (propiedades `Name` y `Help`) y del llamado `AutomationPeer`.
- Soporte de accesibilidad a controles personalizados.
- Soporte de lectores de pantalla y exposición del árbol de automatización completo.
- Notificaciones para requisitos de alto contraste.
- Soporte de formatos de audio comprimido de calidad (ver capítulo 1).

Si necesitamos ver el árbol de accesibilidad para nuestras aplicaciones, recomendamos desde aquí la

herramienta *UI Spy*[3] (similar al *XML-Spy*, pero especial para Silverlight).

Comunicaciones entre capas

En lo tocante a llamadas a servicios Web de distintas clases, recuerde las consideraciones respecto a sincronía, hechas en el capítulo anterior. Si —aún así— necesita que algún proceso sea síncrono, utilice otros hilos de ejecución, preferiblemente basados en la clase `BackgroundWorker` (sino, el navegador se bloquea durante el proceso).

Si se necesitan comunicaciones en ambos sentidos en tiempo real con el servidor, considere la utilización de servicios *WCF FullDuplex*, o el uso de *sockets*, que es mucho más escalable.

Seguridad

Hablando en general, la seguridad en Silverlight es mayor que en otros tipos de aplicaciones debido a lo que comentamos al comienzo de esta obra sobre la ejecución en *SandBox*. Recuerde también la explicación del modelo de seguridad para la ejecución de código del capítulo 1, dentro del apartado dedicado al *CoreCLR*.

La Autenticación y la Autorización pueden seguir los patrones estándar de las aplicaciones Web tradicionales, según lo comentado en el capítulo 7 sobre los ficheros de configuración para acceso entre dominios. Por lo demás, en un escenario de Internet, lo mejor es usar autenticación *Forms*, y autorización basada en roles, como en ASP.NET.

---

[3] `http://msdn2.microsoft.com/en-us/library/ms727247.aspx`

Si estamos en una Intranet, la autenticación Windows y los roles de Windows, pueden ser adecuados, aunque debe tenerse presente que no funciona para usuarios de otros sistemas operativos. Por último, respecto al cifrado de las comunicaciones, considere como forma genérica el uso de SSL sobre HTTPS y recuerde que para otros propósitos el *CoreCLR* soporta un subconjunto de la oferta criptográfica de .NET en los espacios de nombres `System.Security.Cryptography` y `System.Messaging`, especialmente si se tienen que guardar datos sensibles en la zona de almacenamiento aislado.

Respecto al uso de *cookies*, las restricciones son por la vinculación a un mismo nivel: de dominio, de protocolo y de puerto.

Acceso a datos

En el capítulo anterior hemos visto ya algunas recomendaciones en función del escenario de acceso. No obstante, recordemos tres puntos en este apartado tan crítico:

- **Reglas de negocio**: como norma general, ya se ha indicado que la lógica de negocio debe residir en el servidor. Solo interesa mover al cliente aquellos aspectos imprescindibles para mejorar el rendimiento de casos especiales de uso, o por razones de reutilización. Y nunca dividir demasiado la lógica entre servidor y cliente. Lo que tenga que ir en el cliente, mejor situarlo en ensamblados separados para la carga dinámica y que nunca sean ensamblados críticos.
- El uso del caché del cliente puede mejorar la situación en algunas circunstancias.

- Los filtros de datos, siempre en el servidor, para reducir el tráfico.

Excepciones y depuración

Debemos diseñar una estrategia de tratamiento de excepciones: las de tipo síncrono, tratarlas con bloques `try/catch`, y las de tipo asíncrono, mediante el evento `OnError` de Silverlight. Además, recuerde no utilizar la lógica de excepciones para controlar el flujo del programa y determinar claramente cuándo se notifica al servidor las excepciones producidas en el cliente.

En cuanto a la depuración, tenga presentes las capacidades de Visual Studio 2008, incluyendo la de depurar distintos hilos de ejecución. Esto resulta especialmente útil cuando se están utilizando objetos del tipo `BackgroundWorker` y similares.

Además, Silverlight permite el uso y almacenamiento de trazas en ficheros, pero esto funciona por usuario. Podemos —tras una planificación— transferir esos ficheros al servidor con determinada frecuencia para un tratamiento global, pero sin olvidar las restricciones que tiene el uso del almacenamiento aislado.

En cuanto a la monitorización de uso de la aplicación, recomendamos un vistazo a un *Building Block* para Silverlight, disponible en CodeProject, llamado CLOG: `http://www.codeproject.com/KB/silverlight/SilverlightLogging.aspx`.

Distribución de las aplicaciones

Lo primero es idear una estrategia adecuada para los casos en los que el complemento de Silverlight no esté instalado. Recuerde que ya hemos comentado la presencia del elemento ASP.NET `<PluginNotInstalled Template>`, donde

podemos diseñar una pequeña IU para implementar una experiencia de usuario distinta en estos casos.

Optimación de la experiencia inicial

En el escenario típico, el usuario sin el complemento de Silverlight instalado, se encuentra con una zona vacía, y un logo de enlace a la descarga de la versión de Silverlight disponible en ese momento. Sería deseable disponer de una mayor indicación, para que el visitante tenga un anticipo de lo que ofrece la página.

Para minimizar el problema, Microsoft, publicó —a primeros de 2008— un documento titulado "Optimizing the Silverlight Installation Experience", que resumo aquí:

La experiencia de instalación

La idea es evitar el comportamiento predeterminado cuando Silverlight no se encuentra en la máquina, sustituyendo el espacio vacío del control por una imagen representativa de lo que se puede esperar tras su descarga y puesta en funcionamiento. Se distinguen, pues, dos tipos de experiencias de usuario: la predeterminada (cuando la versión del complemento instalado es suficiente para ver la página) y la de instalación (cuando no es así).

Para conseguir una experiencia de instalación adecuada, debemos:

- Diseñar la experiencia predeterminada.
- En base a ella, diseñar la experiencia de instalación.
- Enlazar ambas experiencias.

Para el diseño de la experiencia predeterminada, seleccionamos una imagen que sea descriptiva de aquello que podrá ver el usuario una vez instalado el complemento (puede servir con una captura de pantalla) y tener claramente separado lo que es el control Silverlight, de lo que le rodea. Puede verse en la siguiente imagen, que he incorporado a mi sitio (`www.elavefenix.net`) en el apartado de fotos:



Figura 8-2. Imagen alternativa del visor de fotografías.

Conviene igualmente, planificar redistribuciones cuando hay una instancia en ejecución y llevar un control de las versiones de los componentes. El atributo `MinimumVersion` del elemento `<asp:Silverlight>` nos permite un control más fino de las versiones.

Por último, no olvide comprobar la ejecución en los navegadores más populares. Esto incluye, al menos, tres: IE, Firefox y Safari, aunque puede que —según lo anunciado— dentro de poco también haya soporte por parte de Operan y Google Chrome. No debiera de haber grandes diferencias, pero —en este tipo de aplicaciones—

algún pequeño cambio puede estropear la interfaz de usuario. Por otro lado, es de esperar la aparición de actualizaciones de los navegadores para solucionar la compatibilidad con el complemento. En principio, la versión final de Silverlight 2 es totalmente compatible con la 1.0.

Algunas técnicas recomendadas

Dado lo vasto del modelo de aplicaciones que propone Silverlight 2.0, es natural que haya procedimientos que se repitan en situaciones diversas, y no queremos pasar por alto un comentario sobre algunas de ellas.

Acceso a recursos de la aplicación

La carga de recursos de la aplicación debería de hacerse siempre que sea posible en forma dinámica y asíncrona (utilizando un objeto `WebClient` y una URI). Si éste es el caso, resulta sencillo programarlo utilizando un mecanismo similar al siguiente:

Listado 8-2

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
   WebClient wc = new WebClient();
   wc.OpenReadCompleted += new OpenReadCompletedEventHandler(
                           wc_OpenReadCompleted1);
   wc.OpenReadAsync(new Uri("Foto.jpg", UriKind.Relative));
}

void wc_OpenReadCompleted1(object sender, OpenReadCompletedEventArgs e)
{
```

```
    if (e.Error == null)
    {
        StreamResourceInfo sri = new StreamResourceInfo(
                                        e.Result as Stream, null);
        BitmapImage imagen = new BitmapImage();
        imagen.SetSource(sri.Stream);
        ControlImagen.Source = imagen; // ControlImagen es cualquier
                                  // control capaz de albergar una imagen…
    }
}
```

Ahora bien, dependiendo del escenario, puede interesarnos empotrar los recursos más imprescindibles o aquellos que pensemos que pueden resultar de difícil acceso para el usuario final, dentro de la propia aplicación (ya sea en la DLL o en el fichero XAP que vamos a distribuir), y no hay limitación al tipo de fichero que se puede incluir en el paquete XAP: imágenes, vídeo, iconos, ficheros de texto, HTML, etc.

Tratamiento de recursos en tiempo de compilación

En Visual Studio tendremos que indicar previamente que cierto elemento queremos que sea embebido en la aplicación en el proceso de compilación, lo que haremos desde la ventana de propiedades del elemento:

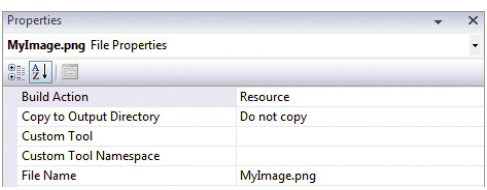


Figura 8-3.La propiedad Build Action permite asignar cualquier elemento como recurso.

Una vez hecho esto, desde el código, utilizaremos la clase **StreamResourceInfo**, para el acceso al recurso. En este caso, al tratarse de una imagen, crearemos un **Bitmap** previo a la asignación de éste a un control:

Listado 8-3

```
using System.Windows.Resources; // para StreamResourceInfo
using System.Windows.Media.Imaging; // para BitmapImage

//y en el evento adecuado…

StreamResourceInfo sr = Application.GetResourceStream(
new Uri("SilverlightApplication1;component/Foto.png", UriKind.Relative));
BitmapImage bmp = new BitmapImage();
bmp.SetSource(sr.Stream);
ControlImagen.Source = bmp
```

Si la imagen se encontrase guardada en una subcarpeta le indicaríamos la situación al objeto **StreamResourceInfo** con una sintaxis del tipo **component/Imágenes/Foto.png**). En el caso de que prefiramos embeberlo en el propio fichero XAP de distribución, sustituiremos la opción de compilación anterior por **Content** (en lugar de **Resource**, en la Ventana de propiedades). El código fuente sería prácticamente el mismo, salvo en la cadena de creación de la URI:

Listado 8-4

```
StreamResourceInfo sr = A pplication.GetResourceStream(
                            new Uri("Foto.png", UriKind.Relative));
```

```
BitmapImage bmp = new BitmapImage();
bmp.SetSource(sr.Stream);
```

E igual que antes, si estuviese en un directorio se lo indicaríamos, pero, ahora sin la barra separadora inicial: **Imágenes/Foto.png**.

Manejo de fuentes adicionales

Otro de los casos habituales de carga de ficheros desde el paquete distribuible es el de las fuentes especiales. El proceso puede hacerse de la misma forma, pero resulta especialmente sencillo desde Blend 2.0, ya que basta con añadir las fuentes como ítem existente y Blend hará una copia en un directorio local, a partir de la cual podemos referirnos a esa fuente en el propio diseño. También podemos seleccionar una fuente del sistema con idéntico resultado.

En el código XAML, sólo tendremos que indicar qué versión de la fuente deseamos o dejar que Blend realice esa tarea por nosotros. El código resultante XAML será algo así (fragmento):

Listado 8-5

```
<TextBlock FontFamily="./Fonts/timheuer.ttf#Tim Heuer Normal"
    FontSize="24" Text="Esto es una prueba de fuentes embebidas.
                    (Tim Heuer Normal)"
    TextWrapping="Wrap" HorizontalAlignment="Center" />
<TextBlock FontFamily="./Fonts/FUJI2N.TTF#Fujiyama2"
    HorizontalAlignment="Center" FontSize="24" TextWrapping="Wrap">
   <Run Text="Y esto, otra prueba "/>
```

```
    <Run Text="(Fujiyama2)"/>
</TextBlock>
```

Esto da una gran libertad, ya que muchas fuentes no son voluminosas (en este caso, ocupan unos 40 Kb entre ambas) y se puede conseguir una gran personalización de la página (figura 8-4).

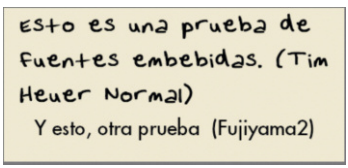


Figura 8-4. Fuentes embebidas.

Hay que constatar que la versión final de Silverlight 2 ha supuesto una considerable mejora en la calidad visual de la interpretación de las fuentes.

Configurando un servidor Web para el alojamiento de Silverlight

Silverlight puede ser distribuido desde diferentes plataformas Web, y no solo Internet Information Server. Por ejemplo, el popular servidor Apache es perfectamente utilizable para este propósito, igual que Sun Java System Web Server.

Sin embargo, el mecanismo utilizado por los servidores para determinar el tipo de contenido asociado a una petición (*Web request*), se gestiona mediante tablas de asignación del servidor a los llamados tipos MIME. Por defecto, los servidores se encuentran configurados

para atender peticiones de unos pocos tipos bien definidos, y esto es correcto, pero, como Silverlight introduce dos tipos de extensiones de fichero (`.xaml` para los ficheros de presentación y `.xap` para el formato binario de empaquetado basado en `.zip`), es necesario añadir esas extensiones al servidor.

Esto se consigue mediante la vinculación de dichas extensiones a sus *tipos* MIME (*Multipurpose Internet Mail Extensions*)[4]. La tabla siguiente muestra las correspondencias necesarias para ellas.

Extension	MIME Type
.xaml	application/xaml+xml
.xap	application/x-silverlight-app

Tabla 1. Tipos MIME introducidos por Silverlight.

La forma de añadir las extensiones a un servidor, varía de servidor en servidor, pero existen enlaces de Internet explicativos de esta mecánica para los servidores más populares (ver notas al pie [5],[6],[7] y [8]).

En caso de que se desee conseguir un soporte completo de los nuevos tipos propuestos por las aplicaciones ClickOnce y WPF en un servidor, también sería necesario añadir los siguientes:

4. Ver definición en Wikipedia: `http://en.wikipedia.org/wiki/MIME`.

5 Para Apache 2.0: **`http://httpd.apache.org/docs/2.0/mod/mod_mime.html#addtype`**

6 Para Sun Java System Web Server: **`http://docs.sun.com/app/docs/doc/819-2630/abumi?a=view`**

7 Para Internet Information Server 7: **`http://technet2.microsoft.com/windowsserver2008/en/library/2761b1cfcb53-40b2-80a1-f0c97030d7d61033.mspx?mfr=true`**

8 Para Internet Information Server 6: **`http://support.microsoft.com/kb/326965`**

Extension	MIME Type
.manifest	application/manifest
.applicatio	application/x-ms-application
.xbap	application/x-ms-xbap
.deploy	application/octet-stream
.xps	application/vnd.ms-xpsdocument

Tabla 2. Tipos MIME adicionales para aplicaciones ClickOnce y WPF

Con esto, tendríamos configurado el servidor para las peticiones originadas en estas aplicaciones. No obstante, hay ocasiones en las que no disponemos de permisos de modificación del servidor donde tenemos alojada la aplicación. La ventaja, es que la extensión problemática (en este caso, `.xap`), está basada en ZIP (como ocurre con los documentos Office 2007). Por tanto, basta con renombrar la extensión `.xap` a `.zip` y hacer que el parámetro de la etiqueta HTML que carga el control Silverlight (`<object>` o `<asp:Silvelright>`) apunte a la nueva ubicación del fichero. Por ejemplo, si tenemos un fichero Silverlight llamado originalmente `ContenidoSL.xap`, basta con cambiar la extensión a `ContenidoSL.zip`, y podríamos añadir en el código fuente una secuencia equivalente a esta (listado 8-6).

Listado 8-6

```
<div id="silverlightControlHost">
<object data="data:application/x-silverlight," type="
                  application/x-silverlight-2"
```

```
            width="100%" height="100%">
               <param name="source" value="ContenidoSL.zip"/>
</object>
</div>
```

Donde, el `mime-type` que define la etiqueta `<object>`, es, en este caso, `application/x-silverlight-2`, o sea, la última versión sobre la que estamos trabajando.

Valores instalados de forma predeterminada en IIS 7.0 sobre Windows

Si utilizamos IIS 7.0 sobre plataformas Windows, hay dos excepciones en las que todos los tipos MIME necesarios ya se encuentran instalados:

- Windows Server 2008.
- Windows Vista Service Pack 1 (instalación limpia).

Para Vista, si hemos actualizado desde Vista RTM a Vista SP1, esto no es así, debiéndose desinstalar y reinstalar IIS 7.0 en el equipo.

Conclusión

El material cubierto en esta obra, no ha sido más que un viaje inicial por la geografía del producto. Hemos tratado de cubrir todos los aspectos importantes de Silverlight 2.0, con la fortuna de haber podido revisarlo todo y hacer las correcciones y añadidos necesarios que introducía la versión final, respecto a las *betas*

anteriores. Ateniéndonos al formato reducido de esta serie de Cuadernos Técnicos de **dotNetManía** no había espacio para mucho más, pero espero que el lector encuentre en este texto un fundamento suficiente que le permita comenzar a practicar y —después— profundizar en Silverlight 2.

apéndice 1

Bibliografía e información on-line

Libros

- *Shared Source CLI*, *David Stutz*, **Ted Neward y Geoff Shilling**. O'Reilly Editorial, 2003
- *Silverlight 1.0 Unleashed*, **Adam Nathan**, SAMS, 2007

Artículos

- *Programming Silverlight with the CoreCLR*, **Andrew Pardoe**, MSDN Magazine, Agosto 2008
- *Create Animations with XAML and Expression Blend*, **Lawrence Moroney**, MSDN Magazine, Agosto 2008
- *Wicked Code: Craft Custom Controls for Silverlight 2*, **Jeff Prosise**, MSDN Magazine, Agosto 2008
- *Service Driven Apps with Silverlight 2 and WCF*, **John Papa**, MSDN Magazine, Septiembre 2008

Información on-line en inglés

- Sitio oficial: `http://Silverlight.net`.
- Blog de **Scott Guthrie**: `http://weblogs.asp.net/Scottgu`.
- Blog de **Joe Stegman**: `http://blogs.msdn.com/jstegman`.
- Blog de **Tim Heuer**: (Method~of~failed): `http://timheuer.com/blog`.

- **Project Astoria** Team Blog:
  `http://blogs.msdn.com/astoriateam`.
- Blog de **César de la Torre**:
  `http://blogs.msdn.com/cesardelatorre`.
- Blog de **Jessy Liberty**:
  `http://silverlight.net/blogs/jesseliberty`.
- 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`.

Información on-line en castellano

- Blog de **Luis Miguel Blanco**:
  `http://geeks.ms/blogs/lmblanco`.
- Blog de **David Salgado**: `http://blogs.msdn.com/davidsalgado`.