Microsoft® Visual Basic™

Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight

Aplicación WPF
•
Introducción a WPF
•
Menús y barras de herramientas
•
Controles y cajas de diálogo
•
Enlaces de datos en WPF
•
Acceso a una base de datos



LINQ
•
Navegación de tipo Web
•
Silverlight
•
Servicios WCF
•
Autenticación y autorización
•
Acceso a datos utilizando WCF RIA Services

Fco. Javier Ceballos

WWW

Puede descargarse el CD-ROM con las URL para obtener el software de desarrollo y las aplicaciones contenidas en el libro

Ra-Ma®

Visual Basic™

Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight

Visual Basic™

Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight

Fco. Javier Ceballos Sierra
Profesor titular de la
Escuela Politécnica Superior
Universidad de Alcalá

Ra-Ma®



Visual Basic: Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight.
© Fco. Javier Ceballos Sierra
© De la edición: RA-MA 2012

MARCAS COMERCIALES: las marcas de los productos citados en el contenido de este libro (sean o no marcas registradas) pertenecen a sus respectivos propietarios. RA-MA no está asociada a ningún producto o fabricante mencionado en la obra, los datos y los ejemplos utilizados son ficticios salvo que se indique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa ni de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA Editorial
C/ Jarama, 3A, polígono industrial Igarsa
28860 PARACUELLOS DEL JARAMA, Madrid
Teléfono: 91 658 42 80
Telefax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: **ebooks.ra-ma.com**, **www.ra-ma.es** **y** **www.ra-ma.com**
ISBN: 978-84-9964-422-6
Depósito legal: M-24114-2012
Autoedición: Fco. Javier Ceballos
Filmación e impresión: Closas-Orcoyen, S.L.
Impreso en España
Primera impresión: julio 2012

Si algo puede salir mal, saldrá mal.
Ley de Murphy.

Dedico esta obra, con mucho cariño,
a mi nieta Isabella.

CONTENIDO

PRÓLOGO..... XXI

Para quién es este libro..... XXIII
Cómo está organizado el libro..... XXIV
Qué se necesita para utilizar este libro..... XXV
Sobre los ejemplos del libro..... XXV
Agradecimientos..... XXV

CAPÍTULO 1. APLICACIÓN WPF..... 1

PROGRAMANDO EN WINDOWS..... 3
BIBLIOTECA WPF..... 5
ESTRUCTURA DE UNA APLICACIÓN..... 6
XAML..... 8
¿Por qué XAML?..... 10
Código subyacente..... 11
INICIO DE LA APLICACIÓN..... 12
COMPILAR Y EJECUTAR LA APLICACIÓN..... 15
DISEÑO DE LA INTERFAZ GRÁFICA..... 17
Información básica sobre XAML..... 17
Espacios de nombres XML..... 18
Propiedades como atributos..... 19
Propiedades como elementos..... 20
Propiedades de contenido..... 20
Extensiones de marcado..... 21
Propiedades asociadas..... 23
Propiedades de dependencia..... 24

Crear un elemento 26
Controles más comunes 26
Añadir una etiqueta y editar sus propiedades 27
Añadir un botón de pulsación y editar sus propiedades 28
Añadir una descripción abreviada a un elemento 29
Paneles de diseño 29
Canvas 29
StackPanel 30
WrapPanel 31
DockPanel 32
Grid 33
MANEJO DE EVENTOS 36
Asignar manejadores de eventos a un objeto 37
EVENTOS ADJUNTOS 38
INYECTAR CÓDIGO XAML DURANTE LA EJECUCIÓN 38
CICLO DE VIDA DE UNA VENTANA 41
PROPIEDADES BÁSICAS DE LA VENTANA 42
Administración de la duración 43
Administración de ventanas 43
Apariencia y comportamiento 43
CONFIGURACIÓN DE UNA APLICACIÓN 45
RECURSOS DE UNA APLICACIÓN 47
ATRIBUTOS GLOBALES DE UNA APLICACIÓN 48
CICLO DE VIDA DE UNA APLICACIÓN 49
Permitir una sola instancia de la aplicación 51
Cómo se genera un evento 52
Especificar cuándo se cerrará la aplicación 52
Pantalla de presentación 53
Argumentos en la línea de órdenes 54
Acceso a la aplicación actual 55
RESUMEN 56
EJERCICIOS PROPUESTOS 56

CAPÍTULO 2. INTRODUCCIÓN A WPF 57

CLASES WPF 57
ETIQUETAS, CAJAS DE TEXTO Y BOTONES 61
Desarrollo de la aplicación 63
Objetos 63
Eventos 63
Pasos a seguir durante el desarrollo 64
El formulario, los controles y sus propiedades 64

Tecla de acceso 67
Botón predeterminado 67
Propiedades comunes 67
EVENTOS ENRUTADOS 69
¿Cómo se definen? 71
Responder a los eventos 72
Eventos relacionados con el teclado 74
Eventos relacionados con el foco 77
Seleccionar el texto de una caja de texto 79
Eventos relacionados con el ratón 80
INTERCEPTAR LA TECLA PULSADA 84
Estado del teclado 86
VALIDACIÓN DE UN CAMPO DE TEXTO 87
ENLACE DE DATOS 89
Enlace de datos sin el motor de WPF 89
Notificar cuándo cambia una propiedad 93
Enlace de datos con el motor de WPF 96
La clase Binding 98
Contexto de datos 99
Crear un enlace 99
Origen de datos implícito 100
Origen de datos explícito 103
Enlaces con otros controles 103
Conversores 104
Validación de datos 107
Regla ExceptionValidationRule 108
Regla DataErrorValidationRule 112
Información del enlace 114
Regla de validación personalizada 116
Fuentes relativas 118
ESTILOS Y PLANTILLAS 118
Estilos 119
Vincular controladores de eventos 120
Desencadenadores 121
Plantillas 122
Plantillas de control 123
Plantillas de datos 125
RECURSOS 126
Recursos creados mediante código 128
Recursos del sistema 129
TEMAS Y MÁSCARAS 130
RESUMEN 134
EJERCICIOS PROPUESTOS 135

CAPÍTULO 3. MENÚS Y BARRAS DE HERRAMIENTAS 137

ARQUITECTURA DE UNA BARRA DE MENÚS 137
DISEÑO DE UNA BARRA DE MENÚS 138
Crear una barra de menús 139
Controlador de un elemento de un menú 142
Aceleradores y nemónicos 144
ÓRDENES ENRUTADAS 145
Vincular una orden enrutada con un control 146
Modelo de una orden enrutada 148
Cómo se ejecuta una orden enrutada 150
Órdenes enrutadas personalizadas 153
Aceleradores de teclado 155
Información adicional en las órdenes enrutadas 159
¿Dónde se aplica la orden? 161
Utilizar parámetros 162
ICommand versus RoutedCommand 165
DETALLES DE UN ELEMENTO DE UN MENÚ 174
MENÚS CONTEXTUALES 175
BARRA DE HERRAMIENTAS 177
Diseño de una barra de herramientas 177
Contenedor de barras de herramientas 180
BARRA DE ESTADO 181
Diseño de una barra de estado 182
DESARROLLO DE UN EDITOR DE TEXTOS 183
Caja de texto multilínea 184
Diseño del editor 185
El portapapeles 186
Clase Clipboard 187
Manipular la selección del texto 187
Diseño de la barra de menús 189
Diseño de la barra de herramientas 191
Asignar a un elemento de la interfaz la tarea a realizar 193
Archivo - Salir 193
Editar - Cortar 193
Editar - Copiar 194
Editar - Pegar 195
Opciones - Fuente 195
Opciones - Tamaño 197
Ayuda - Acerca de 198
Eventos comunes a todos los elementos WPF 199
Habilitar o inhabilitar los elementos de un menú 199
Marcar el elemento seleccionado de un menú 201

Deshacer y rehacer 202
Recordar las ediciones reversibles 202
Añadir a la interfaz la orden Deshacer 203
Añadir a la interfaz la orden Rehacer 204
Menú contextual 204
Asociar un icono a la aplicación 204
MENÚS DINÁMICOS 204
RESUMEN 208

CAPÍTULO 4. CONTROLES Y CAJAS DE DIÁLOGO 211

CAJAS DE DIÁLOGO MODALES Y NO MODALES 212
CAJAS DE MENSAJE 212
CAJAS DE DIÁLOGO PERSONALIZADAS 215
Crear una caja de diálogo 217
Mostrar una caja de diálogo 219
Gestionar los botones Aceptar y Cancelar 220
Introducción de datos y recuperación de los mismos 221
DIÁLOGO ACERCA DE 223
VENTANA PROPIETARIA 225
OTROS CONTROLES WPF 226
Casillas de verificación 227
Botones de opción 232
GroupBox y Expander 237
Listas simples 238
Diseñar la lista 241
Iniciar la lista 242
Acceder a los elementos seleccionados 242
Colección de elementos de una lista 243
Lista de elementos de tipo CheckBox 245
Listas desplegables 246
Diseñar la lista 247
Iniciar la lista 249
Acceder al elemento seleccionado 249
Colección de elementos de una lista desplegable 249
Controles de rango definido 251
ScrollBar 251
Slider 255
ProgressBar 256
Visor con barras de desplazamiento 258
Control con pestañas 259
Gestión de fechas 260

ListView ... 262
TreeView ... 264
Guardar el documento XML ... 269
Recargar el documento XML ... 269
Expandir o contraer los nodos ... 270
DataGrid ... 271
Columnas del DataGrid ... 271
Inmovilizar columnas ... 274
Filas del DataGrid ... 274
Detalles de las filas ... 276
CAJAS DE DIÁLOGO ESTÁNDAR ... 278
Cajas de diálogo Abrir y Guardar ... 278
Cajas de diálogo Windows Forms estándar ... 281
Caja de diálogo Imprimir ... 282
CONTROLES DE DOCUMENTOS WPF ... 283
Documentos dinámicos ... 284
Elementos Block ... 285
Elementos Inline ... 289
Paragraph y Run ... 291
Interactuando con los elementos mediante programación ... 291
Acceso a documentos en un fichero ... 297
Editar un documento ... 297
Imprimir un documento ... 300
TEMPORIZADORES Y MODELO DE SUBPROCESOS ... 303
Timer ... 304
Resolución del temporizador ... 306
DispatcherTimer ... 308
RESUMEN ... 309
EJERCICIOS PROPUESTOS ... 309

CAPÍTULO 5. ENLACE DE DATOS EN WPF ... 315

ASPECTOS BÁSICOS ... 315
ENLACE A COLECCIONES DE OBJETOS ... 317
Cómo implementar colecciones ... 317
Vistas de colección ... 318
PLANTILLAS DE DATOS ... 320
Definir una plantilla de datos ... 322
Mejorar la presentación ... 324
Utilizar desencadenadores para aplicar valores de propiedad ... 325
XML COMO FUENTE DE DATOS ... 327
Datos jerárquicos ... 328

Islas de datos ... 331
Soporte .Net para trabajar con XML ... 331
Obtener la vista ... 333
Elemento actual ... 333
Navegar ... 334
Ordenar ... 335
Filtrar ... 336
Agrupar ... 337
Fuente de datos XML sin el proveedor ... 337
Vinculación maestro-detalle ... 339
OBJETOS COMO FUENTE DE DATOS ... 343
Enlace a una colección de objetos ... 345
Vistas de colección de objetos ... 347
Obtener la vista ... 350
Elemento actual ... 350
Navegar ... 351
Ordenar ... 352
Filtrar ... 352
Agrupar ... 353
Insertar y borrar elementos de la colección ... 353
Vinculación maestro-detalle ... 354
Proveedor de datos de objetos ... 357
Virtualización ... 359
Datos introducidos por el usuario ... 360
Solicitar datos al usuario ... 361
Validación ... 366
Visualización de los errores de validación ... 368
Regla de validación personalizada ... 370
Permanecer en la caja de diálogo si hay errores ... 373
Grupos de enlaces ... 374
DataGrid ... 376
Columnas del DataGrid ... 378
Inmovilizar columnas ... 379
Filas del DataGrid ... 379
Selección de celdas ... 380
Detalles de las filas ... 381
Filtrado, agrupación y ordenación ... 383
Validación ... 383
RESUMEN ... 383

CAPÍTULO 6. ACCESO A UNA BASE DE DATOS 385

SQL 386
Crear una base de datos 386
Crear una tabla 386
Escribir datos en la tabla 388
Modificar datos de una tabla 388
Borrar registros de una tabla 389
Seleccionar datos de una tabla 389
Crear una base de datos 391
Base de datos Microsoft Access 391
Base de datos Microsoft SQL Server 393
ADO.NET 394
Componentes de ADO.NET 395
Conjunto de datos 396
Proveedor de datos 398
Objeto conexión 399
Objeto orden 401
Objeto lector de datos 401
Adaptador de datos 402
Modos de conexión 404
Probando una conexión 405
Servicio de conexiones 407
ACCESO CONECTADO A BASE DE DATOS 408
ATAQUES DE INYECCIÓN DE CÓDIGO SQL 411
Órdenes parametrizadas 414
Procedimientos almacenados 415
TRANSACCIONES 417
Transacción implícita TransactionScope 417
Transacciones explícitas 421
CONSTRUIR COMPONENTES DE ACCESO A DATOS 424
Capa de presentación 425
Operaciones contra la base de datos 427
Objetos de negocio 428
Capa de acceso a datos 431
Capa de lógica de negocio 435
Lógica de interacción con la capa de presentación 436
Desacoplar la IU del resto de la aplicación 439
Adaptar la colección de objetos 440
Capa de lógica de negocio 443
Lógica de interacción con la capa de presentación 446
Validación 447
ACCESO DESCONECTADO A BASE DE DATOS 451

Crear la base de datos ... 455
Crear un proyecto WPF ... 456
Conectarse a la base de datos Sql Server ... 457
Crear la capa de acceso a datos ... 458
Capa de lógica de negocio ... 461
Lógica de interacción con la capa de presentación ... 463
Actualizaciones ... 465
Clase DataView ... 467
RESUMEN ... 470

CAPÍTULO 7. LINQ ... 471

RECURSOS DEL LENGUAJE COMPATIBLES CON LINQ ... 471
Declaración implícita de variables locales ... 472
Matrices de tipos definidos de forma implícita ... 472
Tipos anónimos ... 472
Propiedades auto-implementadas ... 473
Iniciadores de objetos y colecciones ... 473
Métodos extensores ... 474
Expresiones lambda ... 475
El delegado Func(Of *T*, *TResu*) ... 477
Operadores de consulta ... 478
Árboles de expresiones lambda ... 481
EXPRESIONES DE CONSULTA ... 484
Compilación de una expresión de consulta ... 487
Sintaxis de las expresiones de consulta ... 489
Cláusula Group ... 489
Productos cartesianos ... 490
Cláusula Join ... 490
Cláusula Into ... 491
Cláusula Let ... 492
PROVEEDORES DE LINQ ... 493
ENTITY FRAMEWORK ... 494
MARCO DE ENTIDADES DE ADO.NET ... 495
Consultar un modelo de objetos ... 499
ACCESO A UNA BASE DE DATOS ... 501
Conectarse a la base de datos ... 502
Generar el modelo de entidades ... 502
Las clases de entidad y el contexto de objetos ... 509
Propiedades de navegación ... 511
Mostrar datos en una interfaz gráfica ... 513
Una aplicación con interfaz gráfica ... 514

Vincular controles con el origen de datos 516
Filtros 521
Contextos de corta duración 522
REALIZAR CAMBIOS EN LOS DATOS 522
Modificar filas en la base de datos 525
Insertar filas en la base de datos 526
Borrar filas en la base de datos 529
Problemas de concurrencia 532
El seguimiento de cambios 535
EJERCICIOS RESUELTOS 538
RESUMEN 543
EJERCICIOS PROPUESTOS 544

CAPÍTULO 8. NAVEGACIÓN DE TIPO WEB 545

WPF, XBAP y Silverlight 545
NAVEGACIÓN 546
Crear la base de datos 547
Crear el proyecto 548
NavigationWindow 549
Page 551
Añadir páginas a la aplicación 552
Diseño de la interfaz gráfica 554
Lógica de negocio 561
Pasar datos entre páginas 563
Duración y diario de las páginas 564
Hyperlinks 565
Frame 567
Funciones de página 569
Diseño 571
Lógica de negocio 574
APLICACIÓN XBAP 576
Publicar la aplicación 577
Seguridad 581
ACCESO A UNA BASE DE DATOS DESDE UNA XBAP 583
Crear la base de datos 583
Conectarse a la base de datos 584
Generar el modelo de entidades 584
Interfaz gráfica 585
Vincular controles con el origen de datos 586
Controles de usuario 590
Modificar registros 593

Guardar los cambios realizados 595
Añadir un nuevo registro 596
Borrar un registro 602
EL CONTROL WEBBROWSER 602
RESUMEN 603

CAPÍTULO 9. SILVERLIGHT 605

ARQUITECTURA 606
CREAR UNA APLICACIÓN SILVERLIGHT 608
Arquitectura de la aplicación Silverlight 609
Compilación de la aplicación Silverlight 612
Página de entrada 613
DISEÑAR UNA PÁGINA SILVERLIGHT 615
Controles Silverlight 615
Redistribuir el espacio de los elementos de un Grid 615
Texto estático 618
Imágenes 619
Controles de contenido 620
Atributos de anotación de datos 621
Diseño de la interfaz 622
Contexto de datos 624
TextBox 624
DescriptionViewer 625
ValidationSummary 625
Label 625
Validación de los datos 625
Origen de los datos 627
Controles de elementos 629
Controles de texto y elementos de texto 631
Controles de rango definido 633
Controles para gestionar fechas 633
Degradados 634
Ventanas y cajas de diálogo 636
Popup 636
ChildWindow 637
GRÁFICOS, ANIMACIÓN Y MULTIMEDIA 642
Gráficos 642
Transformaciones 643
Animaciones 647
Audio y vídeo 654
NAVEGACIÓN 664

Navegación personalizada ... 665
Navegación de Silverlight ... 666
Frame ... 667
Administrador de identificadores de recursos ... 671
Navegación externa ... 674
Extender el sistema de navegación ... 675
Compatibilidad de ejecución fuera del explorador ... 675
Plantilla aplicación de navegación de Silverlight ... 676
ACCESO A DATOS ... 676
Acceso a los datos de una colección ... 677
Crear la base de datos ... 681
Crear una aplicación Silverlight ... 683
Vincular controles con el origen de datos ... 684
Paginación controlada ... 687
Paginación personalizada ... 688
Filtrar los registros de la colección ... 693
Trabajar con imágenes ... 694
Cargar una nueva imagen ... 699
Guardar los cambios realizados ... 701
Añadir un nuevo registro ... 701
Borrar un registro ... 705
PUBLICAR LA APLICACIÓN ... 706
RESUMEN ... 708

CAPÍTULO 10. SERVICIOS WCF ... 711

MODELO DE PROGRAMACIÓN DE WCF ... 712
Implementar un servicio WCF ... 712
Definir un contrato ... 714
Implementar un cliente WCF ... 720
Configuración del cliente ... 724
Obtener acceso al servicio WCF ... 725
Comunicación entre dominios ... 728
Publicar la aplicación ... 730
SERVICIOS WCF HABILITADOS PARA SILVERLIGHT ... 734
Crear un servicio WCF habilitado para Silverlight ... 734
Implementar un cliente WCF ... 738
Añadir una referencia al servicio ... 738
Publicar la aplicación ... 740
SERVICIOS WEB Y LINQ ... 743
Arquitectura de N capas lógicas y N niveles físicos ... 744
Crear la base de datos ... 745

Obtener acceso a la base de datos 746
Crear el servicio WCF 748
Cliente Silverlight 756
Llenar la lista 761
Mensajes para el usuario 762
Ordenar la lista 763
Mostrar datos 764
Actualizar datos 766
Actualizar la foto 767
Agregar datos 767
Borrar datos 768
Publicar el servicio WCF y la aplicación Silverlight 769
RESUMEN 775

CAPÍTULO 11. AUTENTICACIÓN Y AUTORIZACIÓN 777

SERVICIOS DE AUTENTICACIÓN 778
Autenticación de Windows 779
Autenticación mediante formularios 779
Clase FormsAuthentication 781
Autenticación mediante formularios en Silverlight 782
SERVICIOS DE APLICACIÓN DE ASP.NET 790
Crear la estructura de la aplicación 791
Asignar y configurar servicios de aplicación 793
Crear usuarios 796
Autenticación 800
Funciones (roles) 804
Perfiles 808
Autorización de ASP.NET 808
SIMPLIFICAR EL DESARROLLO DE APLICACIONES 813
Plantilla aplicación de negocios Silverlight 815
Autenticación, funciones y perfiles 819
RESUMEN 823

CAPÍTULO 12. ACCESO A DATOS UTILIZANDO RIA SERVICES 825

ACCESO A DATOS 826
Crear y configurar la solución 827
Mostrar datos utilizando la clase LoadOperation 828
Generar el modelo de entidades 828
Agregar un servicio de dominio 829
LoadOperation 832

DomainDataSource 839
Parámetros de consulta 841
Ordenar, filtrar y agrupar 842
Paginación 842
Actualizar la base de datos 843
Añadir nuevos registros 848
Borrar registros 851
RESUMEN 852

APÉNDICE A. ENTORNO DE DESARROLLO INTEGRADO 855

MICROSOFT VISUAL STUDIO 855
Crear un nuevo proyecto 857
El formulario 861
Dibujar los controles 862
Borrar un control 866
Propiedades de los objetos 867
Icono de la aplicación 869
Escribir los controladores de eventos 869
Guardar la aplicación 872
Verificar la aplicación 872
Propiedades del proyecto 874
Crear soluciones de varios proyectos 875
Opciones del EDI 876
Personalizar el EDI 877
SQL SERVER EXPRESS 877
SQL SERVER MANAGEMENT STUDIO EXPRESS 880
EXPLORADOR DE BASES DE DATOS 881
AÑADIR UN DATASET AL PROYECTO 883
Esquemas XSD 886
Base de datos XML 887
VISUAL WEB DEVELOPER 891
INSTALACIÓN DE ASP.NET EN WINDOWS 891
Registro manual de ASP.NET en IIS 891

APÉNDICE B. CD 893

ÍNDICE 895

PRÓLOGO

Visual Basic es hoy uno de los lenguajes de programación más populares del mundo. Desde que Microsoft liberó Visual Basic 1.0 en 1991 han tenido lugar muchos cambios. Visual Basic 1.0 revolucionó la forma de desarrollar software para Windows; desmitificó el proceso de desarrollo de aplicaciones con interfaz gráfica de usuario y abrió este tipo de programación a las masas. En sus posteriores versiones, Visual Basic ha continuado proporcionando nuevas características que facilitaron la creación de aplicaciones para Windows cada vez más potentes; por ejemplo, la versión 3.0 introdujo el control de datos para facilitar el acceso a bases de datos, y la versión 4.0 mejoró y potenció este acceso con los objetos DAO. Con la aparición de Windows 95, Microsoft liberó Visual Basic 4.0, que abrió la puerta al desarrollo de aplicaciones de 32 bits y a la creación de DLL. La versión 5.0 mejoró la productividad con la incorporación de la ayuda inteligente y la introducción de los controles ActiveX. Posteriormente la versión 6.0 nos introdujo en la programación de Internet con las aplicaciones DHTML y el objeto *WebClass*. Después dispusimos de Visual Basic .NET, que vino a revolucionar el mundo de las comunicaciones permitiendo escribir aplicaciones escalables para Internet. Siguieron Visual Basic 2005, Visual Basic 2008 y ahora Visual Basic 2010, una evolución del lenguaje Visual Basic, que se diseñó para generar aplicaciones con seguridad de tipos y orientadas a objetos de manera productiva. Esta generación de Visual Basic continúa la tradición de ofrecer una manera rápida y fácil de crear aplicaciones basadas en .NET Framework.

Visual Basic .NET, después Visual Basic 2005, Visual Basic 2008 y ahora Visual Basic 2010, cambia la idea de programar de las versiones iniciales. Ahora se requiere una programación orientada a objetos, lo que obligará al desarrollador a programar de forma ordenada, con unas reglas metodológicas de programación análogas a las de otros lenguajes de programación orientados a objetos como C++, C# o Java por citar algunos de los más utilizados.

La palabra "Visual" hace referencia, desde el lado del diseño, al método que se utiliza para crear la interfaz gráfica de usuario si se dispone de la herramienta adecuada (con *Microsoft Visual Studio* se utiliza el ratón para arrastrar y colocar los objetos prefabricados en el lugar deseado dentro de un formulario) y desde el lado de la ejecución, al aspecto gráfico que toman los objetos cuando se ejecuta el código que los crea, objetos que formarán la interfaz gráfica que el usuario de la aplicación utiliza para acceder a los servicios que ésta ofrece. La palabra "Basic" hace referencia al lenguaje BASIC (*Beginners All-Purpose Symbolic Instruction Code*), un lenguaje utilizado por más programadores que ningún otro lenguaje en la historia de la informática. Visual Basic ha evolucionado a partir del lenguaje BASIC original y ahora está soportado por una biblioteca orientada a objetos directamente relacionada con la interfaz gráfica de Windows. Y "NET" hace referencia al ámbito donde operarán nuestras aplicaciones webs (*Network* - red).

De forma resumida, Visual Basic es un lenguaje orientado a objetos seguro y elegante que permite a los desarrolladores construir un amplio rango de aplicaciones seguras y robustas que se ejecutan sobre .NET Framework. Podemos utilizar Visual Basic para crear aplicaciones cliente Windows tradicionales, servicios webs XML, servicios WCF, componentes distribuidos, aplicaciones cliente servidor, aplicaciones para acceso a bases de datos, y muchas otras. *Microsoft Visual Basic 2010* y superiores proporcionan un editor de código avanzado, diseñadores de interfaces de usuario apropiados, depurador integrado, y muchas otras utilidades para facilitar el desarrollo rápido de aplicaciones basadas en el lenguaje Visual Basic y en .NET Framework.

Por otra parte, .NET es una plataforma de desarrollo compuesta básicamente por el CLR (la máquina virtual), la BCL (la biblioteca básica) y un conjunto de lenguajes de programación, entre los que se encuentra Visual Basic. .NET partió con la versión 1.0 y pasó por las 1.1, 2.0, 3.0, 3.5 y 4.0, que es la versión en el momento de escribir esta obra, la cual se incluye con Visual Studio 2010 y con Windows 7 e incluye, entre otras cosas, LINQ, WPF, WCF, WF, ASP.NET 4.0, AJAX, VSTO y Silverlight. ¿Y todo esto para qué? Pues para crear aplicaciones de Internet más ricas. Se trata realmente de nuevas bibliotecas que nos permiten hacer lo que ya hacíamos con .NET 2.0, pero de una forma más vistosa, más rápida y más eficiente. Si miramos hacia atrás (Win32), teníamos la biblioteca MFC. Después vino .NET y ahora vienen otras bibliotecas que tratan de facilitar al desarrollador la realización de aplicaciones más ricas en menos tiempo.

El contenido de este libro se va a centrar básicamente en las tecnologías WPF, WCF y Silverlight.

WPF (*Windows Presentation Foundation*) es una biblioteca para el desarrollo de interfaces gráficas de usuario vectoriales avanzadas. Esta biblioteca de clases no ha sido creada para sustituir a *Windows Forms*, sino que es otra biblioteca que

facilita el desarrollo de aplicaciones de escritorio en las que estén implicados diversos tipos de medios: vídeo, documentos, contenido 3D, secuencias de imágenes animadas, etc. WPF también es idóneo si lo que se necesita es crear una interfaz de usuario con un aspecto personalizado, si hay que establecer vínculos con datos, o si desea crear una aplicación de escritorio con un estilo de navegación similar a una aplicación web. A diferencia de *Windows Forms*, utiliza el lenguaje de marcado XAML para implementar su interfaz gráfica y los lenguajes de programación administrados, como Visual Basic, para escribir el código subyacente que implemente su comportamiento. Esta separación entre la apariencia y el comportamiento permite a los diseñadores implementar la apariencia de una aplicación al mismo tiempo que los programadores implementan su comportamiento.

WCF (*Windows Communication Foundation*) es un marco de trabajo unificado para hacer fácil la comunicación entre aplicaciones diversas en cualquier plataforma (.NET, J2EE, etc.) combinando en una sola tecnología lo que en versiones anteriores de Visual Studio existía en varias tecnologías: servicios webs ASMX, .NET Remoting, Enterprise Services y Message Queue Server. WPF está orientado al servicio. Los servicios son autónomos y comparten esquemas (datos) y contratos (funcionalidad), no clases ni tipos en general y al intercambiar mensajes no tienen que asumir nada acerca de "qué es lo que hay al otro lado del extremo". Los clientes consumen servicios y los servicios ofrecen soluciones a los clientes intercambiando mensajes, con lo que un servicio puede, a su vez, ser cliente de otro servicio.

Silverlight es una tecnología multiplataforma que se ejecuta en varios exploradores. Al igual que Flash, Silverlight permite crear contenido interactivo que se ejecuta en el cliente, con soporte para gráficos dinámicos, contenido multimedia y animación, que va mucho más allá del HTML ordinario y también, al igual que Flash, Silverlight se implementa en los navegadores mediante un plug-in. Silverlight incluye una versión reducida de .NET Framework y de WPF, lo que permite a los desarrolladores escribir código de cliente utilizando Visual Basic o C#.

Para quién es este libro

Este libro está pensado para aquellas personas que quieran aprender a desarrollar aplicaciones que muestren una interfaz gráfica al usuario, aplicaciones para acceso a bases de datos y para Internet, utilizando básicamente las bibliotecas WPF, WCF, Silverlight y el entorno de desarrollo Visual Studio 2010 o superior. Para ello, ¿qué debe hacer? Pues simplemente leer ordenadamente los capítulos del libro, resolviendo cada uno de los ejemplos que en ellos se detallan.

Evidentemente, el autor asume que el lector conoce el lenguaje *Visual Basic*, ya que este libro no describe este lenguaje debido a que este tema fue expuesto

pormenorizadamente en sus otros libros *Microsoft Visual Basic .NET - Lenguaje y aplicaciones* o *Microsoft Visual Basic .NET - Curso de programación*, ambos editados también por RA-MA, y que tiene conocimientos orientados al desarrollo de aplicaciones utilizando la biblioteca *Windows Forms* y el entorno de desarrollo ASP.NET, temática que tampoco se expone porque fue expuesta en su otro libro *Enciclopedia de Microsoft Visual Basic*.

Cómo está organizado el libro

El libro se ha estructurado en 12 capítulos más algunos apéndices que a continuación se relacionan. El capítulo 1 estudia los conceptos básicos de WPF y nos introduce en el desarrollo de una aplicación WPF. En el capítulo 2 se hace una introducción a la jerarquía de clases de WPF, al uso de los controles y eventos más frecuentes, a la validación de datos y a la personalización de la apariencia de una aplicación. El capítulo 3 nos enseña cómo añadir una barra de menús, de herramientas o de estado a una ventana WPF, cómo añadir un menú contextual y a utilizar las órdenes enrutadas. El capítulo 4 explica cómo utilizar multitud de controles WPF en el diseño de interfaces gráficas y cómo apoyar estos diseños con cajas de diálogo. En el capítulo 5 se estudia el enlace a datos, uno de los pilares de WPF, y las colecciones de objetos, ya que éstas serán los orígenes de los datos que serán proporcionados por los enlaces a los controles de la interfaz gráfica del usuario y viceversa. El capítulo 6 cubre el acceso a bases de datos utilizando ADO.NET y el desarrollo de aplicaciones basado en capas. El capítulo 7 continúa con el acceso a bases de datos, pero utilizando el lenguaje de consultas integrado LINQ, para después centrarnos en el proveedor LINQ to Entities que permite consultar las entidades que definen el modelo conceptual de Entity Framework. El capítulo 8 nos enseña cómo utilizar el modelo de navegación de WPF, un modelo basado en páginas. En el capítulo 9 se estudia la tecnología Silverlight y cómo desarrollar aplicaciones para la web o no, utilizando esta tecnología. En el capítulo se 10 estudian los servicios WCF y se expone cómo desarrollar una aplicación Silverlight de N capas y N niveles, que tiene que acceder a una base de datos a través de servicios WCF. El capítulo 11 estudia cómo implementar la autenticación y la autorización en una aplicación Silverlight. Y el capítulo 12 estudia el acceso a datos utilizando WCF RIA Services.

CAPÍTULO 1. APLICACIÓN WPF
CAPÍTULO 2. INTRODUCCIÓN A WPF
CAPÍTULO 3. MENÚS Y BARRAS DE HERRAMIENTAS
CAPÍTULO 4. CONTROLES Y CAJAS DE DIÁLOGO
CAPÍTULO 5. ENLACES DE DATOS EN WPF
CAPÍTULO 6. ACCESO A UNA BASE DE DATOS
CAPÍTULO 7. LINQ
CAPÍTULO 8. NAVEGACIÓN DE TIPO WEB
CAPÍTULO 9. SILVERLIGHT
CAPÍTULO 10. SERVICIOS WCF

CAPÍTULO 11. AUTENTICACIÓN Y AUTORIZACIÓN
CAPÍTULO 12. ACCESO A DATOS UTILIZANDO WCF RIA SERVICES
APÉNDICE A. ENTORNO DE DESARROLLO INTEGRADO

Qué se necesita para utilizar este libro

Este libro ha sido escrito utilizando el paquete *Microsoft .NET Framework Software Development Kit* (SDK) versión 4.0 incluido en el entorno de desarrollo *Microsoft Visual Studio 2010* (o en su defecto *Visual Basic 2010 Express*, *Visual Web Developer 2010 Express* y *SQL Server 2010 Express*) que incluye todo lo necesario para escribir, construir, verificar y ejecutar aplicaciones .NET. Por lo tanto, basta con que instale en su máquina el software mencionado. Las versiones *Express*, que puede descargar desde *http://www.microsoft.com/express/*, son gratuitas.

Nota: para probar las aplicaciones webs se recomienda instalar el servidor de aplicaciones IIS (*Internet Information Server*) incluido con Windows (*Inicio > Panel de control > Agregar y quitar programas > Windows*). Esto tiene que hacerlo antes de instalar *Microsoft Visual Studio 2010* o *Visual Web Developer 2010 Express*.

Sobre los ejemplos del libro

La imagen del CD de este libro, con las aplicaciones desarrolladas y el software para reproducirlas, puede descargarla desde:
https://www.tecno-libro.es/ficheros/descargas/9788499644226.zip
La descarga consiste en un fichero ZIP con una contraseña `ddd-dd-dddd-ddd-d` que se corresponde con el ISBN de este libro (teclee los dígitos y los guiones).

Agradecimientos

He recibido ayuda de algunas personas durante la preparación de este libro, y por ello les estoy francamente agradecido. También, deseo expresar mi agradecimiento a *Microsoft Ibérica* por poner a mi disposición, en particular, y de todos los lectores, en general, el software que el estudio de esta obra requiere.

Francisco Javier Ceballos Sierra
`http://www.fjceballos.es/`

CAPÍTULO 1

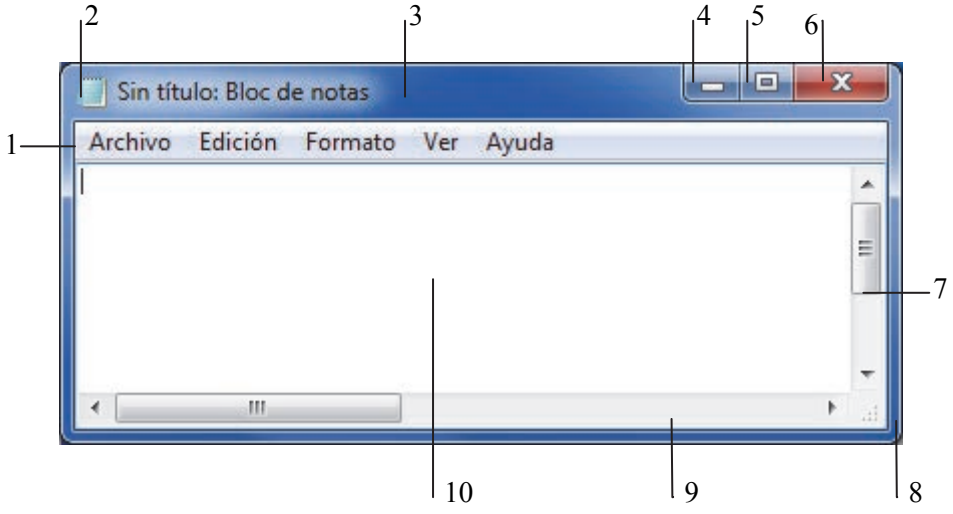
© F.J.Ceballos/RA-MA

APLICACIÓN WPF

Una de las grandes ventajas de trabajar con Windows es que todas las ventanas se comportan de la misma forma y todas las aplicaciones utilizan los mismos métodos básicos (menús desplegables, botones) para introducir datos o emitir órdenes.

El punto de interacción entre el usuario y una aplicación independiente es una ventana. Una ventana típica de Windows tiene las siguientes partes:

1. *Barra de menús*. Visualiza el conjunto de los menús disponibles para esa aplicación. Cuando se activa alguno de los menús haciendo clic con el ratón sobre su título, se visualiza el conjunto de órdenes que lo forman.



2. *Icono de la aplicación y menú de control*. El menú de control proporciona órdenes para: restaurar su tamaño, mover, tamaño, minimizar, maximizar y cerrar la ventana.

3. *Barra de título*. Contiene el nombre de la ventana y del documento. Para mover la ventana a otro lugar, apunte con el ratón a esta barra, haga clic utilizando el botón izquierdo del ratón y, manteniendo pulsado el botón, arrastre en la dirección deseada. Un doble clic maximiza o retorna a tamaño normal la ventana, dependiendo de su estado actual.

4. *Botón para minimizar la ventana*. Cuando se pulsa este botón, la ventana se reduce a su forma mínima. Ésta es la mejor forma de mantener las aplicaciones cuando tenemos varias de ellas activadas y no se están utilizando en ese instante.

5. *Botón para maximizar la ventana*. Cuando se pulsa este botón, la ventana se amplía al máximo y el botón se transforma en  . Si éste se pulsa, la ventana se reduce al tamaño anterior.

6. *Botón para cerrar la ventana*. Cuando se pulsa este botón, se cierra la ventana y la aplicación si la ventana es la principal.

7. *Barra de desplazamiento vertical*. Cuando la información no entra verticalmente en una ventana, Windows añade una barra de desplazamiento vertical a la derecha de la ventana.

8. *Marco de la ventana*. Permite modificar el tamaño de la ventana. Para cambiar el tamaño, apunte con el ratón a la esquina o a un lado del marco, y cuando el puntero cambie a una flecha doble, con el botón izquierdo del ratón pulsado arrastre en el sentido adecuado para conseguir el tamaño deseado.

9. *Barra de desplazamiento horizontal*. Cuando la información no entra horizontalmente en una ventana, Windows añade una barra de desplazamiento horizontal en el fondo de la ventana.

   Cada barra de desplazamiento tiene un *cuadrado de desplazamiento* que se mueve a lo largo de la barra para indicar en qué posición nos encontramos con respecto al principio y al final de la información tratada, y dos *flechas de desplazamiento*.

   Para desplazarse:

   - Una línea verticalmente o un carácter horizontalmente, utilice las flechas de desplazamiento de las barras.

- Varias líneas verticalmente o varios caracteres horizontalmente, apunte con el ratón a una flecha de desplazamiento, haga clic con el botón izquierdo y mantenga el botón pulsado.

- Aproximadamente una pantalla completa, haga clic sobre la barra de desplazamiento. Para subir, haga clic por encima del cuadrado de desplazamiento de la barra vertical, y para bajar, haga clic por debajo del cuadrado. Para moverse a la izquierda, haga clic a la izquierda del cuadrado de desplazamiento de la barra horizontal, y para moverse a la derecha, haga clic a la derecha del cuadrado.

- A un lugar específico, haga clic sobre el cuadrado de desplazamiento y, manteniendo el botón del ratón pulsado, arrastre el cuadrado.

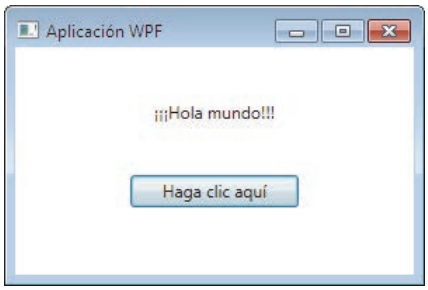
10. *Área de trabajo*. Es la parte de la ventana en la que el usuario coloca el texto y los gráficos.

Un objeto en general puede ser movido a otro lugar haciendo clic sobre él y arrastrándolo manteniendo pulsado el botón izquierdo del ratón.

PROGRAMANDO EN WINDOWS

Una aplicación para Windows diseñada para interaccionar con el usuario presentará una interfaz gráfica que mostrará todas las opciones que el usuario puede realizar. Dicha interfaz se basa fundamentalmente en dos tipos de objetos: ventanas (también llamadas formularios) y controles (botones, cajas de texto, menús, listas, etc.); esto es, utilizando estos objetos podemos diseñar dicha interfaz, pero para que proporcione la funcionalidad para la que ha sido diseñada, es necesario añadir el código adecuado. En resumen, para realizar una aplicación que muestre una interfaz gráfica, se crean objetos que den lugar a ventanas y sobre esas ventanas se dibujan otros objetos llamados controles; finalmente se escribe el código fuente relacionado con la función que tiene que realizar cada objeto de la interfaz. Esto es, cada objeto estará ligado a un código que permanecerá inactivo hasta que se produzca el evento que lo active. Por ejemplo, podemos programar un botón (objeto que se puede pulsar) para que al hacer clic sobre él con el ratón muestre un formulario solicitando unos determinados datos.

Según lo expuesto, una aplicación en Windows presenta todas las opciones posibles en uno o más formularios para que el usuario elija una de ellas. Por ejemplo, en la figura siguiente, cuando el usuario haga clic sobre el botón *Haga clic aquí*, en la caja de texto aparecerá el mensaje *¡¡¡Hola mundo!!!* Se dice entonces que la programación es *conducida por eventos* y *orientada a objetos*.



Cuando desarrollamos una aplicación de este estilo, la secuencia en la que se ejecutarán las sentencias no puede ser prevista por el programador. Por ejemplo, si en lugar de un botón, *btSaludo*,

```
<Button Content="Haga clic aquí"
        Name="btSaludo"
        Click="btSaludo_Click" />
```

hubiera dos o más botones, claramente se ve que el programador no puede escribir el programa pensando que el usuario va a pulsarlos en una determinada secuencia.

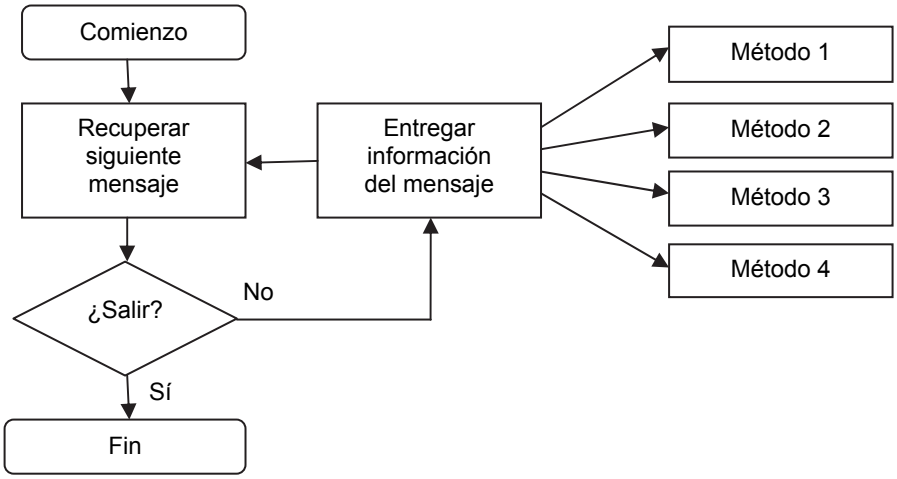
Por lo tanto, para programar una aplicación Windows hay que escribir código separado para cada objeto en general, quedando la aplicación dividida en pequeños procedimientos o métodos conducidos por eventos. Por ejemplo:

```
Private Sub btSaludo_Click(sender As Object, e As RoutedEventArgs)
  etSaludo.Content = "¡¡¡Hola mundo!!!"
End Sub
```

El método *btSaludo_Click* será puesto en ejecución en respuesta al evento **Click** del objeto identificado por *btSaludo* (botón titulado "Haga clic aquí"). Quiere esto decir que cuando el usuario haga clic en el objeto *btSaludo* se ejecutará el método *btSaludo_Click*. Esto es justamente lo que está indicando el código XAML `Click="btSaludo_Click"` que se muestra un poco antes: el método *btSaludo_Click* manejará el evento **Click** de *btSaludo*. Por ello, esta forma de programar se denomina programación conducida por eventos y orientada a objetos.

Los *eventos* son mecanismos mediante los cuales los objetos (ventanas o controles) pueden notificar de la ocurrencia de sucesos. Un evento puede ser causado por una acción del usuario (por ejemplo, cuando pulsa una tecla), por el sistema (transcurrido un determinado tiempo) o indirectamente por el código (al cargar una ventana). En Windows, cada ventana y cada control pueden responder a un conjunto de eventos predefinidos. Cuando ocurre uno de estos eventos, Windows lo transforma en un mensaje que coloca en la cola de mensajes de la aplicación implicada. Un método **Run**, denominado *bucle de mensajes*, es el encargado de

extraer los mensajes de la cola y despacharlos para que sean procesados. Evidentemente, cada mensaje almacenará la información suficiente para identificar el objeto y ejecutar el método que tiene para responder a ese evento. En la figura siguiente puede ver de forma gráfica cómo actúa el bucle de mensajes mientras la aplicación está en ejecución:



Como ejemplo, un poco más adelante construiremos la aplicación que acabamos de comentar, la que da lugar al mensaje "¡¡¡Hola mundo!!!".

BIBLIOTECA WPF

Para diseñar aplicaciones que utilicen interfaces gráficas (ventanas con controles, como etiquetas, cajas de texto, botones, barras de desplazamiento, etc.), .NET proporciona, además de la biblioteca *Windows Forms*, otra biblioteca de clases denominada *WPF* (***W**indows **P**resentation **F**oundation* - clases base de .NET para el desarrollo de interfaces gráficas de usuario vectoriales avanzadas definidas en su mayoría en el espacio de nombres **System.Windows**).

WPF no ha sido creado para sustituir a *Windows Forms*, sino que la biblioteca *Windows Forms* seguirá siendo mejorada y mantenida por Microsoft. *WPF* es simplemente otra biblioteca con otras posibilidades para el desarrollo de aplicaciones de escritorio. Por ejemplo, facilita el desarrollo de aplicaciones en el que estén implicados diversos tipos de medios: vídeo, documentos, contenido 3D, secuencias de imágenes animadas, o una combinación de cualquiera de los anteriores. *WPF* también es idóneo si lo que se necesita es crear una interfaz de usuario con un aspecto personalizado (*skins*), si hay que establecer vínculos con datos XML, o si hay que cargar dinámicamente porciones de una interfaz de usuario desde un servicio web, o se desea crear una aplicación de escritorio con un estilo

de navegación similar a una aplicación web. Además, y a diferencia de *Windows Forms*, proporciona la capacidad para programar una aplicación utilizando el lenguaje de marcado XAML para implementar su interfaz gráfica y los lenguajes de programación administrados, como Visual Basic, para escribir el código subyacente que implemente su comportamiento. Esta separación entre la apariencia y el comportamiento permite a los diseñadores implementar la apariencia de una aplicación al mismo tiempo que los programadores implementan su comportamiento.

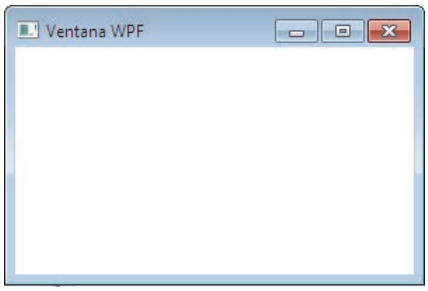
Con *WPF* se puede crear aplicaciones *Windows* de los siguientes tipos:

- *Aplicación WPF*. Se trata de una aplicación de escritorio, denominada también aplicación cliente Windows.
- *Aplicación de explorador WPF*. Se trata de aplicaciones cuya interfaz será mostrada a través del explorador o navegador. Combinan las características de las aplicaciones webs y de las aplicaciones cliente. Estas aplicaciones, denominadas también aplicaciones XBAP, se pueden publicar en un servidor web e iniciarse desde el explorador (igual que las aplicaciones webs). Su desarrollo es similar al de las aplicaciones cliente.
- *Bibliotecas de clases*. Ensamblados no ejecutables que contienen clases reutilizables.
- *Biblioteca de controles de usuario/personalizados de WPF*. Este tipo de aplicación se refiere a un control de usuario o personalizado (desarrollo avanzado de controles).

¿*WPF* o *Windows Forms*? *Windows Forms* todavía tiene un papel importante que desempeñar, a pesar de que *WPF* haya entrado en escena. Si estamos construyendo aplicaciones que no necesitan de la amplia y moderna funcionalidad de *WPF*, entonces no hay razón de peso para cambiar y dejar atrás toda una experiencia adquirida. Además, actualmente, *Windows Forms* acumula mucha más experiencia en Visual Studio que *WPF*, razón de peso para no olvidarnos de esta biblioteca, la cual fue expuesta en mi otro libro *Enciclopedia de Microsoft Visual Basic*.

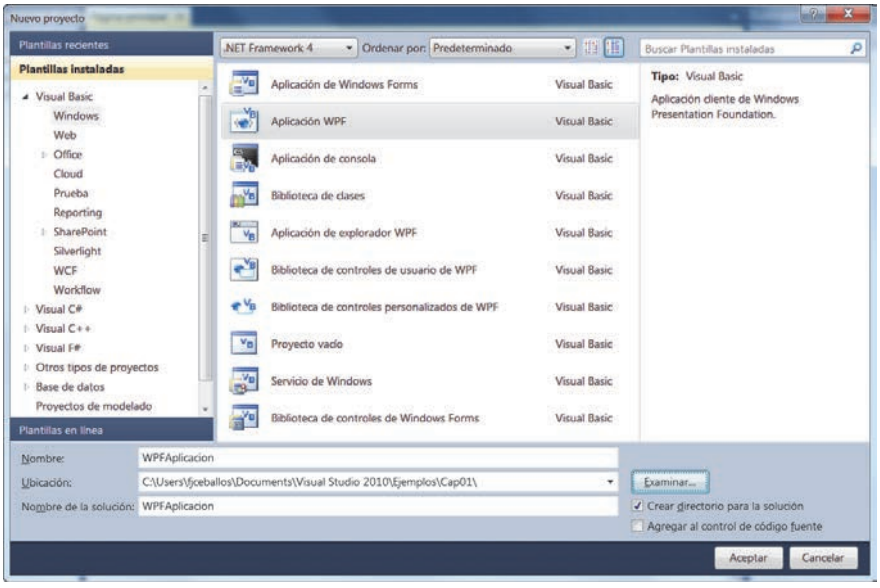
ESTRUCTURA DE UNA APLICACIÓN

Este apartado explica cómo construir una aplicación mínima que presente una interfaz gráfica. De esta forma podemos estudiar desde un punto de vista práctico cuáles son y cómo interaccionan entre sí los elementos que configuran una aplicación WPF. Dicha aplicación construirá y visualizará una ventana como la de la figura siguiente:



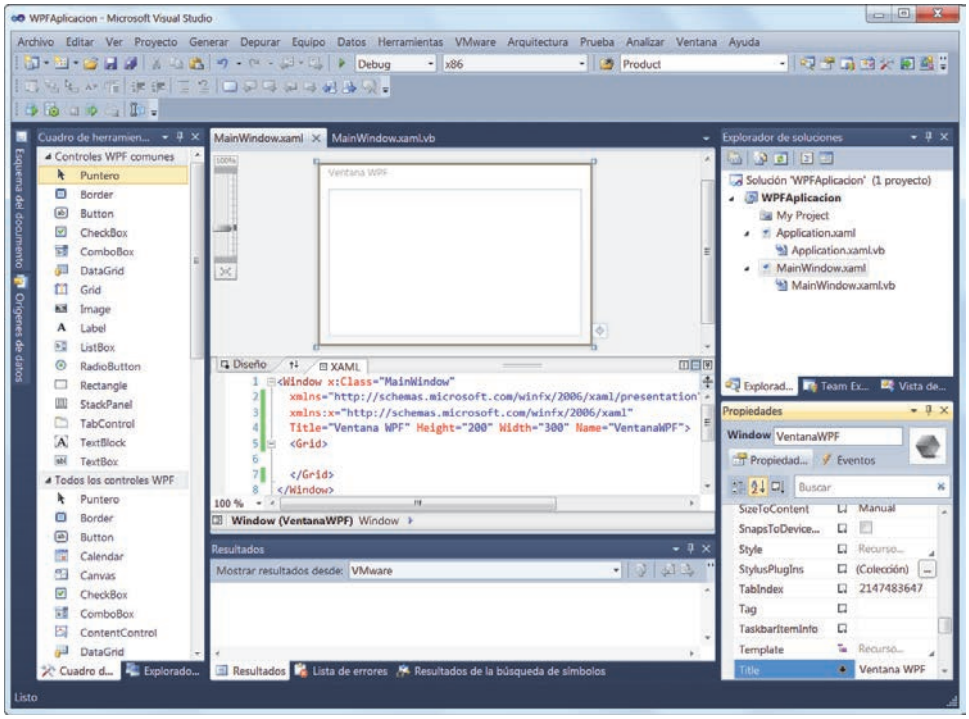
Esta ventana tiene un menú de control, un título y los botones de maximizar, minimizar y cerrar. Cuando el usuario pulse el botón la ventana se reducirá a un icono, cuando pulse el botón la ventana se agrandará para ocupar toda la pantalla y cuando lo haga en la ventana se cerrará. Así mismo, cuando haga clic encima del icono de la aplicación situado a la izquierda de la barra de título se abrirá el menú de control. Este menú incluye las órdenes: *Restaurar*, *Mover*, *Tamaño*, *Minimizar*, *Maximizar* y *Cerrar*. Las tres últimas realizan la misma función que los botones descritos.

Empecemos entonces por abrir Visual Studio y crear un nuevo proyecto de tipo *Aplicación WPF* (*WPF Application*) denominado *WPFAplicacion*. Obsérvese en la parte superior de la ventana que se puede seleccionar de una lista la versión de .NET Framework que vamos a utilizar.



Una vez hayamos seleccionado el tipo de proyecto e introducido su nombre y localización, hacemos clic en el botón *Aceptar*. Esta acción generará una aplica-

ción WPF mínima, que mostrará un objeto ventana, cuyo código podemos explorar en la figura siguiente correspondiente al EDI Visual Studio:



En este EDI observamos la caja de herramientas, la ventana de diseño, la ventana de código XAML, el explorador de soluciones, la ventana de propiedades, que añade una caja para poner el nombre del objeto seleccionado y una caja de búsqueda de propiedades, y la ventana de resultados.

Observe también en el explorador de soluciones los ficheros que se han creado: *Application.xaml*, *Application.xaml.vb*, *MainWindow.xaml* y *MainWindow.xaml.vb* que explicaremos a continuación.

XAML

XAML (*e**X**tensible **A**pplication **M**arkup **L**anguage*: lenguaje de marcado extensible para aplicaciones o XML para Aplicaciones) se utiliza para definir la estructura jerárquica de la interfaz de usuario, la cual se puede ver como un árbol de elementos (en nuestra aplicación, la raíz de este árbol es un objeto **Window** que tendrá como hijos los controles que añadamos), árbol que normalmente guardaremos en un fichero en XAML, proceso que recibe el nombre de seriación; esto es, el fichero XAML es simplemente una seriación de objetos. XAML es un len-

guaje declarativo. Como ejemplo, en la ventana de código XAML de la figura anterior podemos ver el código XAML que define la ventana y que está guardado en el fichero *MainWindow.xaml*:

```
<Window x:Class="WPFAplicacion.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="200" Width="300" Name="VentanaWPF">
  <Grid>

  </Grid>
</Window>
```

El código anterior declara una ventana (elemento **Window**), denominada *VentanaWPF*, con unos atributos determinados (se corresponden con propiedades en la clase); algunos, como **Title**, **Height**, **Width** y **Name**, han sido definidos de forma explícita. El atributo **xmlns** (***xml name space***) define un espacio de nombres XML; hay dos declaraciones de este tipo: la primera indica específicamente el espacio de nombres XAML predeterminado que está asignado al espacio de nombres *http://schemas.microsoft.com/winfx/2006/xaml/presentation* de WPF necesario para hacer referencia a los elementos de código XAML que proceden de los ensamblados de WPF (en este espacio de nombres los elementos en el marcado se pueden especificar sin un prefijo), y la segunda asigna al prefijo **x** el espacio de nombres XAML para el lenguaje XAML (en este espacio de nombres los elementos en el marcado se especifican con el prefijo **x**); se trata de un espacio de nombres adicional que contiene varias construcciones de programación que usaremos con mucha frecuencia en el código XAML como *x:Class*, *x:Name*, *x:Key*, *x:Static* o *x:Type*.

Por ejemplo, volviendo al ejemplo anterior, observamos que el atributo **x:Class** define la clase de código subyacente vinculada con este código XAML; esto es, este atributo le indica al procesador de XAML que deberá generar una clase *MainWindow* derivada del elemento raíz, **Window**. Ahora, fijémonos en el atributo **Title** de **Window**, no tiene prefijo de espacio de nombres, lo que indica, normalmente, que se corresponde con una propiedad del objeto .NET al que hace referencia el elemento XAML. Esto último sería equivalente al código siguiente:

```
VentanaWPF.Title = "Ventana WPF"
```

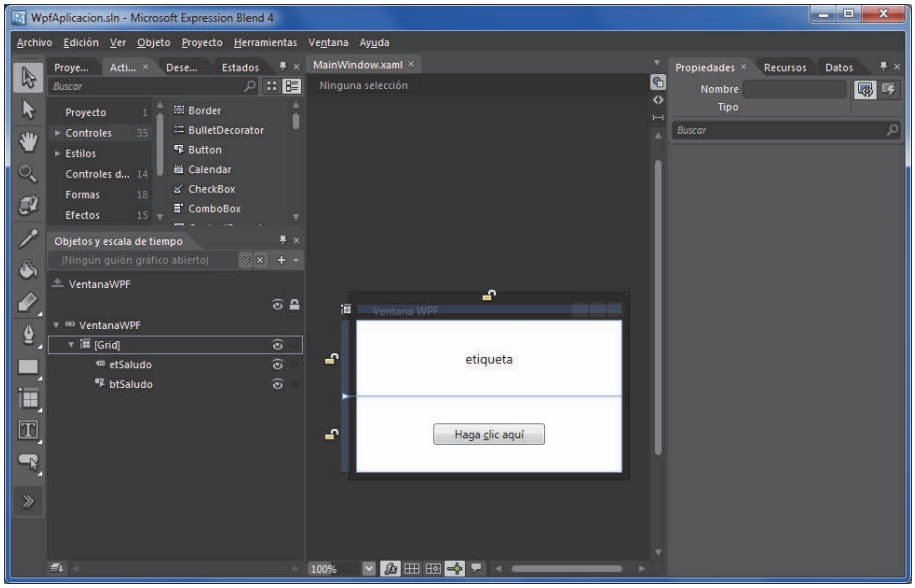
Resumiendo, detrás de cada elemento XAML hay siempre una clase del CLR de .NET y todo atributo XAML se corresponde con el nombre de una propiedad o de un evento de dicha clase; por ejemplo, el elemento *<Window>* especifica a través de su atributo **x:Class** que se utilizará la clase *MainWindow* del espacio de nombres *WPFAplicacion* derivada de la clase **Window** de la biblioteca WPF. Esto quiere decir que, para el ejemplo que nos ocupa, por omisión la clase sería **Win-**

dow; para un elemento <*Grid*> se utilizará la clase **Grid** y así sucesivamente. Deducimos entonces, que no hay porqué utilizar XAML para escribir una aplicación WPF; esto es, podríamos escribir código Visual Basic para crear los objetos que intervienen en la aplicación igual que lo hacíamos con la biblioteca *Windows Forms*.

¿Por qué XAML?

Si ha creado previamente aplicaciones con .NET Framework mediante tecnologías como ASP.NET y *Windows Forms*, el desarrollo de estas aplicaciones en WPF le resultará familiar: creará la interfaz gráfica con el lenguaje XAML (en vez de con XHTML) y controlará los eventos con el lenguaje .NET que prefiera, por ejemplo con Visual Basic. De esta forma, al no estar la apariencia de la aplicación estrechamente relacionada con su comportamiento, se reducen los costos de programación y mantenimiento.

Por otra parte, cuando diseñamos una interfaz gráfica con *Windows Forms*, lo que hace Visual Studio es generar código en el lenguaje de programación que estemos utilizando, por ejemplo Visual Basic. Esto limita el desarrollo a aplicaciones de diseño capaces de procesar este código, como Visual Studio. En cambio, si utilizamos XAML, cualquier herramienta de diseño capaz de procesar XML podrá deseriar el código XAML generado por Visual Studio, modificarlo y volverlo a seriar nuevamente, de modo que pueda volverse a cargar por Visual Studio. Así, mientras los programadores emplean Visual Studio para escribir el código subyacente, los diseñadores pueden utilizar otras herramientas, como Microsoft *Expression Blend*.



Código subyacente

Para responder a los eventos y manipular los objetos declarados en XAML, como etiquetas, cajas de texto, botones, se usa un fichero de código subyacente independiente. En nuestro caso, este fichero es *MainWindow.xaml.vb*:

```
Class MainWindow

End Class
```

Esta declaración parcial de la clase *MainWindow*, vinculada con la etiqueta <*Window*> anterior, no deja claro que una ventana, como la anterior, es un objeto de una clase derivada de **Window**. Lo que en realidad ocurre es que el procesador de marcado de WPF creará una clase parcial para el fichero XAML compilado, derivada del tipo de elemento raíz (en nuestro ejemplo de **Window**), que se combinará con la definición también parcial de la misma clase y espacio de nombres, expuesta anteriormente, que define el código subyacente. Esto lo podemos verificar si observamos el fichero *MainWindow.g.vb* que se localiza en la carpeta *obj/.../Release|Debug* del proyecto.

Resumiendo, podemos decir que el fichero *MainWindow.g.vb* se genera a partir del código XAML del fichero *MainWindow.xaml*, lo que corrobora lo dicho anteriormente: no hay porqué utilizar XAML para escribir una aplicación WPF, podríamos escribir código Visual Basic para crear los objetos que intervienen en la aplicación, pero de esta forma no separaríamos la apariencia de la aplicación de su comportamiento.

La deseriación/seriación se lleva a cabo a través de la funcionalidad proporcionada por las clases del espacio de nombres **System.Windows.Markup** como **XamlReader** y **XamlWriter**, por ejemplo. Podríamos ver esto de una forma práctica si, como estudiaremos a continuación, añadiéramos al formulario un botón vinculado con el método siguiente:

```
Private Sub button1_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs) Handles button1.Click
  Using destino As New FileStream("resultado.xaml",
                                  FileMode.Create)
    XamlWriter.Save(Me, destino)
  End Using
End Sub
```

El fichero *resultado.xaml* que se obtendría una vez ejecutado el método anterior, almacenaría la seriación del árbol de elementos que componen la interfaz gráfica de nuestra aplicación (objeto referenciado por **Me**):

```
<MainWindow Title="Ventana WPF" Name="VentanaWPF"
            Width="300" Height="200" Visibility="Visible"
  xmlns="clr-namespace:WpfAplicacion;assembly=WpfAplicacion"
  xmlns:av="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <av:Grid>
    <av:Button Name="button1" Width="75" Height="23" Margin="100,67,0,0"
    HorizontalAlignment="Left" VerticalAlignment="Top">Button</av:Button>
  </av:Grid>
</MainWindow>
```

El código Visual Basic que combina con XAML puede insertarse en el propio fichero XAML como se muestra en el ejemplo siguiente. No obstante, esta posibilidad se debe evitar o limitar ya que mantener una separación entre el código de marcado y el subyacente permite diferenciar mucho mejor las funciones del programador y del diseñador, lo que redundará en un mantenimiento más fácil de la aplicación.

```
<DockPanel>
  <x:Code>
    <![CDATA[
      Private Sub button1_Click(sender As System.Object,
      e As System.Windows.RoutedEventArgs)
        ...
      End Sub
    ]]>
  </x:Code>
  <Button Content="Button" Height="23" Name="button1" Width="75"
          Click="button1_Click" />
</DockPanel>
```

INICIO DE LA APLICACIÓN

Cuando ejecutamos la aplicación anterior obtenemos como resultado una ventana con toda su funcionalidad; pero, ¿cuál es el punto de entrada cuando la aplicación se ejecuta? Anteriormente, cuando construimos la aplicación, observamos que se crearon, además de los dos ficheros anteriores, otros dos, *Application.xaml* y *Application.xaml.vb*, para proveer a la aplicación de un punto de inicio y de otros servicios, como propiedades y recursos compartidos, que pueda necesitar:

```
<Application x:Class="Application"
    xmlns="http://schemas.microsoft.com/.../xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

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

En el código anterior, correspondiente al fichero *Application.xaml*, podemos observar que detrás del elemento XAML <*Application*> hay una clase *Application* (definida en *Application.xaml.vb* según se muestra a continuación) perteneciente al espacio de nombres *WPFAplicacion* que está derivada de la clase **Application** de la biblioteca WPF, que es la encargada de proporcionar la administración del inicio y todos esos servicios a los que nos hemos referido anteriormente:

```
Class Application
  ' Los eventos de nivel de aplicación,
  ' como Startup, Exit y DispatcherUnhandledException
  ' se pueden controlar en este archivo.
End Class
```

Si echamos un vistazo a la definición de esta clase formada por esta definición parcial más la que mostramos a continuación, podemos observar que contiene un método **Main** (utilizando *Windows Forms* este método era definido por el EDI en la clase *Program*) por donde se iniciará la ejecución de la aplicación:

```
Partial Public Class Application
  Inherits System.Windows.Application

  Public Sub InitializeComponent()
    #ExternalSource("..\..\..\Application.xaml",4)
    Me.StartupUri =
       New System.Uri("MainWindow.xaml", System.UriKind.Relative)
    #End ExternalSource
  End Sub

  Public Shared Sub Main()
    Dim app As Application = New Application()
    app.InitializeComponent
    app.Run
  End Sub
End Class
```

Igual que ocurría con la clase *MainWindow*, esta definición parcial se localiza en el fichero *Application.g.vb* de la carpeta *obj/.../Release|Debug* del proyecto.

Veamos qué ocurre cuando se ejecuta el método **Main**:

- Primero se invoca al constructor de la clase **Application** para construir un objeto *app*.

- Después se invoca al método *InitializeComponent* para establecer la interfaz de usuario que se mostrará automáticamente; eso es lo que indica la propiedad

StartupUri del objeto **Application**. Esto hará que se cargue el recurso de tipo *MainWindow* (la ventana).

- Finalmente, se visualiza la ventana y se inicia el bucle de mensajes de la aplicación invocando al método **Run** del objeto **Application**.

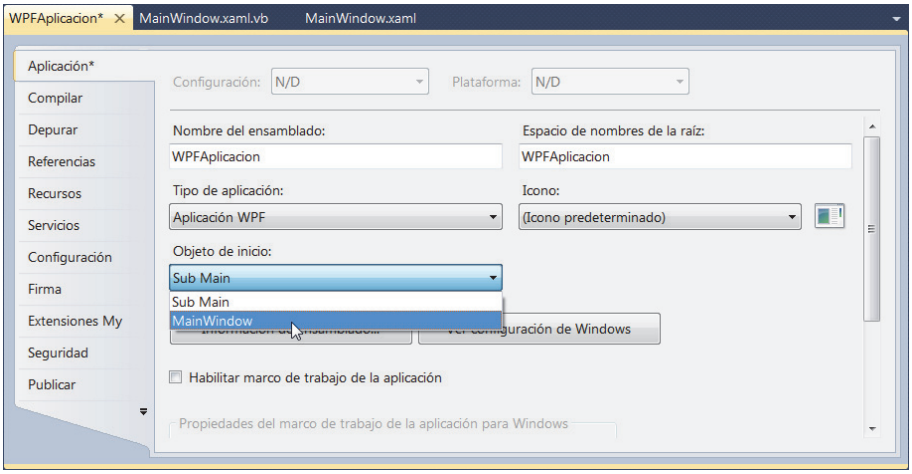
Una vez iniciada la ejecución de la aplicación, ésta queda a la espera de las acciones que pueda emprender el usuario de la misma. En el ejemplo tan simple que acabamos de presentar, una de las acciones que puede tomar el usuario es cerrar la ventana, lo que origina el evento correspondiente. ¿Cómo responde la aplicación a este evento? Finalizando el bucle de mensajes, para lo cual **Run** invoca al método **Application.Shutdown**. El método **Shutdown** cierra la aplicación y fuerza a **Run** a retornar. Lógicamente, se pueden producir otros eventos; por ejemplo: la ventana se abre, se minimiza, vuelve a su estado normal, etc. Lo que de momento tiene que saber es que la aplicación siempre responderá a cada evento invocando a su método asociado.

Normalmente, la propiedad **StartupUri** se establece mediante declaración en XAML según podemos observar en el fichero *Application.xaml* (línea 4).

Suponiendo que no necesitáramos inicialmente proporcionar recursos en el ámbito de la aplicación, podríamos prescindir de los ficheros *Application.xaml* y *Application.xaml.vb* e incluir el método **Main** (por ejemplo, el especificado a continuación) en la clase *MainWindow* o en cualquier otra clase:

```
Public Shared Sub Main()
  Dim app As Application = New Application()
  app.Run(New MainWindow())
End Sub
```

Esto quiere decir que cuando creamos una aplicación Visual Basic con Visual Studio, el código generado aporta explícitamente el método **Main** en una clase *Application* localizada parcialmente en el fichero *Application.g.vb*. No obstante, cuando lo requiera, puede añadir uno personalizado, según muestra el código anterior, indicándolo, si es preciso, en las propiedades de la aplicación:

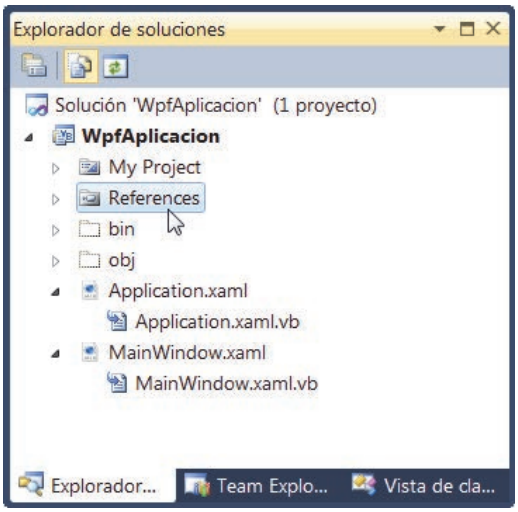


COMPILAR Y EJECUTAR LA APLICACIÓN

Un detalle de estilo a la hora de escribir el código. Observe que *Visual Studio* antepone a los nombres de las clases y otras estructuras de datos el nombre del espacio de nombres al que pertenecen (por ejemplo, cuando anteriormente añadimos el controlador del evento **Click** de un **Button**, el generador de código utilizó **System.Object** en lugar de escribir solamente **Object**), aunque no sería necesario, porque las referencias añadidas en el explorador de soluciones, equivalentes a las sentencias **Imports** que se muestran a continuación, ya los especifican.

```
Imports System
Imports System.Xaml
Imports System.Windows.Controls
```

Análogamente a como las carpetas o directorios ayudan a organizar los ficheros en un disco duro, la biblioteca de clases .NET está organizada en espacios de nombres que agrupan esas clases dispuestas según una estructura jerárquica. Pues bien, estos espacios de nombres se corresponden con bibliotecas dinámicas del mismo nombre, a las que tendremos que hacer referencia cuando se compile una aplicación, en la medida en la que sean necesarias. Esto se hace añadiendo dichas referencias al nodo *References* del proyecto si aún no están añadidas. Para ello, haga clic con el botón secundario del ratón sobre este nodo y seleccione *Agregar referencia*.



Para compilar la aplicación seleccione la orden adecuada del menú *Generar* y para ejecutarla podemos pulsar las teclas *Ctrl+F5*, o bien elegir la orden correspondiente del menú *Depurar*.

Una vez ejecutada la aplicación aparecerá sobre la pantalla la ventana, con el tamaño asignado, y podremos actuar sobre cualquiera de sus controles, o bien sobre las órdenes del menú de control, para minimizarla, maximizarla, moverla, ajustar su tamaño, etc. Ésta es la parte que el diseñador de Visual Studio realiza por nosotros y para nosotros; pruébelo. Finalmente, para cerrar la ejecución de la aplicación disponemos de varias posibilidades:

1. Hacer clic en el botón que cierra la ventana.
2. Hacer un doble clic en el icono situado a la izquierda en la barra de título de la ventana.
3. Activar el menú de control de la ventana y ejecutar *Cerrar*.
4. Pulsar las teclas *Alt+F4*.

Finalmente decir que aunque los formatos basados en XML, como XAML, son flexibles y fácilmente portables a otras herramientas y plataformas, no siempre son la opción más eficiente. XAML fue diseñado para ser lógico, legible y sencillo, pero durante su proceso no resulta todo lo rápido que sería deseable. WPF aborda esta deficiencia con *Binary Application Markup Language* (BAML), que no es nada más que una representación binaria de XAML.

Al compilar en Visual Studio una aplicación WPF, todos los ficheros XAML se convierten en BAML para, a continuación, ser incrustados como recursos en el ensamblado. El código BAML generado es un código significativamente menor y

optimizado, de tal manera que su proceso es más rápido durante la ejecución (en el ejemplo que nos ocupa, puede observar el fichero *MainWindow.baml* en la carpeta *obj/.../Release|Debug* del proyecto). Esta fase de compilación es transparente al desarrollador, pero es bueno saber que también es posible analizar XAML sin compilar, lo cual puede tener sentido en situaciones en las que se requiera que algo de la interfaz de usuario sea suministrado durante la ejecución (por ejemplo, podríamos obtener un bloque de código XAML desde una base de datos o desde un fichero, como veremos al final de este capítulo).

DISEÑO DE LA INTERFAZ GRÁFICA

Nuestro siguiente paso consistirá en añadir a la ventana los elementos que tienen que formar parte de la interfaz gráfica. Este proceso requiere colocar en la misma los controles que nosotros creamos adecuados y añadir a la aplicación otras ventanas si fuera necesario. ¿Cómo? Por medio de WPF.

Por lo expuesto hasta ahora sabemos que WPF es un sistema de presentación para crear aplicaciones Windows cliente, bien independientes o bien hospedadas en el explorador. Así mismo, WPF aporta una serie de características de programación de aplicaciones, entre las que se incluyen el lenguaje XAML, controles, enlace de datos, diseño, gráficos 2D y 3D, animación, estilos, plantillas, documentos, multimedia, texto y tipografía. Además, como se incluye en Microsoft .NET Framework, permite crear aplicaciones que incorporen otros elementos de la biblioteca de clases de .NET Framework.

Información básica sobre XAML

XAML tiene un conjunto de reglas que asignan elementos (como **Window**) a clases o estructuras, atributos a propiedades o eventos, y espacios de nombres de XML a espacios de nombres del CLR, según vimos anteriormente. En el ejemplo siguiente se han especificado dos elementos: **<Grid>**, con una etiqueta de cierre, y **<Button />**, éste con dos atributos: **Content** y **Name**. Cada uno de ellos se asigna a una clase definida en la biblioteca WPF, **Grid** y **Button** respectivamente, que aporta un constructor público sin parámetros que será invocado al cargar la página XAML para crear un objeto de esa clase.

```
<Grid>
  <Button Content="Haga clic aquí" Name="btSaludo" />
</Grid>
```

Un detalle a considerar: en la clase de código subyacente, a menudo necesitamos manipular los controles para, por ejemplo, leer o cambiar sobre la marcha las propiedades o los controladores de eventos. Para hacer esto posible, el control

debe incluir un atributo XAML que especifique el nombre. En el ejemplo anterior, el control **Button** incluye un atributo **Name** que permitirá utilizar el identificador *btSaludo* para referirse a ese botón en el fichero de código subyacente. Esto no sería posible si no hubiéramos incluido el atributo **Name**, como ocurre con el elemento **Grid**.

XAML distingue entre mayúsculas y minúsculas y los valores de los atributos, con independencia de su tipo, deben escribirse entre comillas dobles. XAML se basa en la clase **TypeConverter** de .NET para realizar la conversión entre el tipo **String** y el tipo nativo de las propiedades.

Veamos a continuación algunas reglas básicas del estándar XAML para facilitar su entendimiento.

Espacios de nombres XML

Anteriormente vimos que un fichero XAML declara un espacio de nombres XAML predeterminado en su elemento raíz que define qué elementos se pueden declarar sin calificarlos mediante un prefijo. Por ejemplo, si se declara un elemento *<ConversorDoubleBrush ... />*, se espera que *ConversorDoubleBrush* exista y sea un elemento válido en el espacio de nombres XAML predeterminado. Si no es así, por ejemplo porque está declarado en el código subyacente en el espacio de nombres *ConverTemps.Clases*, se debe calificar esa referencia con un prefijo; por ejemplo, *<**local**:ConversorDoubleBrush ... />*. El prefijo indica que el elemento existe en un espacio de nombres XAML diferente del espacio de nombres predeterminado y, en concreto en el siguiente ejemplo, donde se ha asignado un espacio de nombres del CLR al prefijo *local* para su uso o bien, en el caso de **x**, un espacio de nombres XAML, según muestra el ejemplo siguiente:

```
<Window x:Class="ConverTemps.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  xmlns:local="clr-namespace:ConverTemps.Clases">
  <Window.Resources>
    <local:ConversorDoubleBrush x:Key="converGradosCaBrush" />
  </Window.Resources>
  <Grid>

  </Grid>
</Window>
```

La función del atributo **xmlns** es declarar espacios de nombres XML y la del atributo **clr-namespace**, hacer referencia a espacios de nombres del CLR declara-

dos dentro del ensamblado. El fragmento de código XAML anterior declara dos espacios de nombres que encontraremos en cada aplicación WPF que creemos:

- *http://schemas.microsoft.com/winfx/2006/xaml/presentation* es el espacio de nombres del núcleo de WPF. Abarca todas las clases de WPF, incluyendo las de los controles que utilizamos para construir las interfaces de usuario. Este espacio de nombres se declara sin prefijo, por lo que se convierte en el espacio de nombres predeterminado para todo el documento. En otras palabras, cada elemento se coloca automáticamente en este espacio de nombres a menos que se especifique lo contrario.

- *http://schemas.microsoft.com/winfx/2006/xaml* es el espacio de nombres de XAML. Incluye diversos elementos de XAML que permiten interpretar el documento de una u otra forma. Este espacio de nombres se asigna al prefijo **x**, lo cual significa que se puede hacer referencia al mismo colocando el prefijo antes del nombre del elemento (por ejemplo <*x:id_elemento*>).

Los espacios de nombres permiten al analizador de XAML encontrar la clase correcta. Por ejemplo, cuando el analizador procesa los elementos **Window** y **Grid** busca en el espacio de nombres XAML predeterminado los correspondientes espacios de nombres .NET que definan las clases **System.Windows.Window** y **System.Windows.Controls.Grid**.

Por convenio, los espacios de nombres XML son a menudo URI (parece que apuntan a una localización web, pero no es así). Se utiliza este formato porque favorece que sea poco probable que las diferentes organizaciones sin darse cuenta creen diferentes lenguajes basados en XML con el mismo espacio de nombres. Por ejemplo, el dominio *schemas.microsoft.com* es propiedad de Microsoft y sólo esta firma puede utilizarlo. Otra razón de su uso es que, para facilitar la escritura del documento XAML, se optó por combinar varios espacios de nombres .NET bajo un único espacio de nombres XML, lo cual funciona porque dentro de los diferentes espacios de nombres .NET no hay clases con el mismo nombre.

Propiedades como atributos

En XAML se pueden utilizar atributos para especificar las propiedades de un objeto de una clase. Por ejemplo, la siguiente línea XAML especifica cómo se asignan valores a las propiedades **Content** y **Name** de un objeto de la clase **Button**:

```
<Button Content="Haga clic aquí" Name="btSaludo" />
```

Propiedades como elementos

Cuando el valor de la propiedad no se puede expresar como una cadena simple, en lugar de utilizar atributos, las propiedades se especifican como elementos hijo empleando la sintaxis: *ElementoPadre.NombrePropiedad*. Por ejemplo:

```
<Button Content="Haga clic aquí" Name="btSaludo" >
  <Button.Background>
    <SolidColorBrush Color="Blue"/>
  </Button.Background>
</Button>
```

La verdad es que este ejemplo tiene el mismo efecto que tendría un sencillo atributo *Background="#FF0000FF"*, por lo tanto no es razón suficiente para utilizar la sintaxis de propiedad como elemento. Sí sería necesario cuando necesitamos anidar definiciones más complejas dentro de la propiedad como, por ejemplo:

```
<Button Content="Haga clic aquí" Name="btSaludo" >
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0" Color="#FF080000" />
        <GradientStop Offset="0.60" Color="Blue" />
        <GradientStop Offset="1" Color="#FF050000" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Para especificar el color se ha utilizado bien un color predefinido, como **Blue**, o bien el formato *AARRGGBB*: número hexadecimal de ocho dígitos; los dos primeros dígitos especifican el nivel de transparencia (***A**lfa*; 0 indica que el color, no el objeto, es completamente transparente), los dos siguientes especifican el nivel de rojo (***R**ed*), los dos siguientes especifican el nivel de verde (***G**reen*) y los dos últimos especifican el nivel de azul (***B**lue*). Se pueden utilizar otros formatos con menos dígitos; por ejemplo: *RRGGBB*, *ARGB* o *RGB*.

Propiedades de contenido

Cualquier clase que se pueda utilizar como un elemento XAML puede designar a una de sus propiedades como propiedad de contenido. Esto, en la sintaxis de XAML, permite omitir la especificación explícita de esta propiedad, tomándose como valor para la misma el contenido de cualquier elemento secundario especificado. Por ejemplo:

```
<Button>
  Haga clic aquí
  <Button.Name>btSaludo</Button.Name>
</Button>
```

La propiedad **Content** que anteriormente se especificó explícitamente, obtiene ahora su valor del elemento secundario de texto especificado dentro del elemento **Button**. Esta sintaxis es correcta porque **Content** es la propiedad de contenido de la clase **Button**.

Extensiones de marcado

XAML a través de las extensiones de marcado ofrece una vía alternativa al control habitual de atributos o elementos del procesador de XAML de WPF, cediendo el procesamiento a una clase que decide durante la ejecución cómo establecer el valor de una propiedad.

Las extensiones de marcado más comunes son las que admiten referencias de recursos (**StaticResource** y **DynamicResource**) y las que admiten el enlace de datos (**Binding**). A continuación se muestra un ejemplo del uso de una extensión de marcado:

```
<Grid>
  <Grid.Resources>
    <SolidColorBrush x:Key="ColorFondoBt" Color="Azure" />
  </Grid.Resources>
  <Button Content="Haga clic aquí" Name="btSaludo"
          Background="{DynamicResource ColorFondoBt}" />
</Grid>
```

En este ejemplo el valor de la propiedad **Background** del objeto **Button** se obtiene a partir del recurso *ColorFondoBt*.

El carácter que identifica una extensión de marcado ante un procesador de XAML de WPF es la llave de apertura (**{**), que es seguida por una cadena (en el ejemplo **DynamicResource**) que debe hacer referencia a la clase que proporciona el comportamiento de la extensión en particular (**DynamicResourceExtension**), un espacio, la implementación de la extensión (en el ejemplo *ColorFondoBt*) y una llave de cierre (**}**).

Las extensiones de marcado en WPF están diseñadas principalmente para proporcionar un medio de hacer referencia a otros objetos ya existentes, o referencias diferidas a los objetos que se evaluarán durante la ejecución. Por ejemplo, un enlace de datos simple se consigue especificando la extensión de marcado **{Binding}** en lugar del valor que tomaría normalmente una propiedad determinada.

```
<Button Content="Haga clic aquí" Name="btSaludo"
        Background="{Binding Background, ElementName=btAceptar}" />
```

Aquí, **Binding** utiliza la propiedad **Background** del elemento *btAceptar* para asignar su valor a la propiedad **Background** del elemento *btSaludo*.

Las extensiones de marcado permiten utilizar una sintaxis de "propiedades como atributo" que, de no existir, no sería posible aplicar. Por ejemplo, un objeto **Style** es un tipo de referencia relativamente complejo que contiene varias propiedades, que se crea normalmente como un recurso y, a continuación, se hace referencia a él mediante una extensión de marcado que solicita el recurso:

```
<Grid>
  <Grid.Resources>
    <Style x:Key="BtFondoAzulClaro">
      <Setter Property="Control.Background" Value="LightBlue"/>
    </Style>
  </Grid.Resources>
  <Button Content="Haga clic aquí" Name="btSaludo"
          Style="{StaticResource BtFondoAzulClaro}" />
</Grid>
```

Aquí, **StaticResource** identifica la clase **StaticResourceExtension** que proporciona la implementación de la extensión de marcado. La cadena siguiente, *BtFondoAzulClaro*, se utiliza como la entrada para el constructor **StaticResourceExtension**. Podemos decir que cuando el procesador de XAML no encuentra una clase de extensión de marcado (en el ejemplo no encuentra **StaticResource**) añade al nombre la palabra **Extension** y lo intenta de nuevo.

La extensión de marcado **StaticResource** proporciona un valor para cualquier propiedad como atributo XAML buscando, durante la carga de la aplicación, una referencia a un recurso ya definido, y la extensión **DynamicResource** lo proporciona difiriendo la consulta de recursos hasta que el recurso solicitado se necesite realmente para construir un objeto durante la ejecución. Evidentemente, esta última extensión será la única forma de proporcionar un recurso que sólo esté disponible durante la ejecución. Finalmente indicar que las clases de extensión de marcado derivan de **MarkupExtension**.

Otras extensiones de marcado utilizadas comúnmente son:

- **x:Null**. Extensión utilizada para devolver un objeto **null** que permita establecer explícitamente una referencia **null**.

```
<Button Content="Botón" Name="bt1" Background="{x:Null}" />
```

- **x:Type**. Extensión que devuelve un objeto **Type** que representa al tipo especificado.

```
<Style TargetType="{x:Type Button}" >...</Style>
```

- **x:Static**. Extensión utilizada para obtener valores de propiedades o campos estáticos.

```
<TextBlock Name="textBlock1"
  Background="{x:Static SystemColors.ActiveCaptionBrush}" />
```

- **x:Array**. Extensión que permite crear una tabla de elementos. Requiere que el tipo de la tabla se especifique a través de su propiedad **Type**.

- **RelativeSource**. Extensión que describe la ubicación del origen del enlace relativa a la posición del destino para un objeto **Binding**. Su sintaxis es como se indica a continuación (para más detalles recurra a la ayuda proporcionada por *Microsoft Developer Network* - MSDN):

```
<Binding RelativeSource="{RelativeSource modeEnumValue}" .../>
```

- **Binding**. Extensión que proporciona un valor enlazado a datos para una propiedad.

```
<TextBlock Name="textBlock1" Text="{Binding TextoDoc}" />
```

  Este ejemplo vincula la propiedad **Text** de *textBlock1* a la propiedad *TextoDoc* de un objeto que esté en el contexto de datos.

- **TemplateBinding**. Extensión que admite el enlace entre el valor de una propiedad en una plantilla y el valor de alguna otra propiedad expuesta en el control con plantilla.

Recuerde que el prefijo **x:** denota el espacio de nombres XML del XAML y que la ausencia del prefijo denota el espacio de nombres XML de WPF.

Propiedades asociadas

Se denominan así a las propiedades propias de un tipo que pueden ser establecidas, además de por un elemento del propio tipo, por cualquier otro elemento. La sintaxis es: *TipoPrimario.NombrePropiedad*. El propósito es permitir que los elementos secundarios informen a un elemento primario de los valores de una propiedad, aunque el elemento primario y los secundarios no posean esa propiedad como parte de las listas de miembros de su clase. Por ejemplo:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Button Grid.Row="1"
          Content="Haga clic aquí" Name="btSaludo"
          Height="23" Width="120" />
</Grid>
```

En el ejemplo anterior se observa cómo un elemento secundario, **Button**, informa al primario, **Grid**, de cómo se va a presentar en la interfaz de usuario (en la fila 1 del **Grid**; la primera fila es la 0). Si esto no fuera posible, propiedades como *Grid.Row* tendrían que ser creadas en la clase base, por ejemplo en **FrameworkElement**, lo cual, dada la especificidad de la propiedad, no es una buena idea.

Propiedades de dependencia

WPF proporciona un conjunto de servicios que se pueden utilizar para extender la funcionalidad de una propiedad CLR (una propiedad .NET con sus descriptores de acceso **get** y **set** respaldada por un campo). Este conjunto de servicios se conoce como *sistema de propiedades de WPF*. Pues bien, una propiedad que esté respaldada por la clase **DependencyProperty** y registrada en este sistema se conoce como propiedad de dependencia, implementación alternativa al modelo estándar de respaldar la propiedad con un campo privado.

El propósito de estas propiedades es proporcionar una manera de calcular el valor de una propiedad en función del valor de otras entradas. De esta forma, una propiedad de dependencia puede heredar su valor de un elemento padre; por ejemplo, todos los elementos tienen una propiedad **Cursor** para controlar el cursor del ratón, de tal forma que cuando fijamos esta propiedad en uno de ellos, todos los elementos hijo fijan su propiedad **Cursor** en el mismo valor; también, una propiedad de dependencia puede tomar su valor de expresiones de vinculación de datos, de activadores o de estilos, según hemos visto en algunos de los ejemplos anteriores, características no presentes en las propiedades .NET. Por ejemplo, una propiedad de dependencia, como **Background**, puede hacer referencia mediante un enlace de datos, definido por un objeto **Binding** (el enlace de datos se estudiará en los capítulos siguientes), a un valor que será obtenido durante la ejecución (en el ejemplo, de la propiedad **Background** del elemento *btAceptar*):

```
<Button Content="Haga clic aquí" Name="btSaludo"
        Background="{Binding Background, ElementName=btAceptar}" />
```

Además, se puede implementar para que proporcione validación autónoma, valores predeterminados, devoluciones de llamada que supervisen los cambios de otras propiedades y un sistema que pueda forzar valores de la propiedad en función de la información que pueda estar disponible durante la ejecución; el sistema de propiedades de dependencia de WPF gestiona de forma automática el almacenamiento de los valores, por eso sabe cuándo una propiedad cambia.

Los elementos WPF normalmente utilizan el sistema de propiedades de dependencia de WPF para definir sus propiedades, ya que al derivar **FrameworkElement** indirectamente de la clase **DependencyObject**, tienen todo lo que necesitan. En este caso, los descriptores de acceso **get**/**set** se proporcionan para envolver el sistema de dependencia de la propiedad. En el ejemplo siguiente, que muestra cómo se define una propiedad de dependencia, se puede ver que los descriptores de acceso tan sólo hacen una llamada; a cambio de esta llamada, el sistema de propiedades de dependencia de WPF añade un número de características que las propiedades .NET no ofrecen, como que la propiedad pueda establecerse en un estilo, que admita enlace de datos, etc.

```
Public Shared Readonly NombrePropiedadProperty As DependencyProperty =
    DependencyProperty.Register(
    "NombrePropiedad", GetType(TipoDeLaPropiedad),
...

    )
Public Property NombrePropiedad() As TipoDeLaPropiedad
  Get
    Return CType(GetValue(NombrePropiedadProperty), TipoDeLaPropiedad)
  End Get
  Set(ByVal value As TipoDeLaPropiedad)
    SetValue(NombrePropiedadProperty, value)
  End Set
End Property
```

Vemos que WPF proporciona envoltorios para las propiedades de dependencia. De esta forma podemos gestionar las propiedades de dependencia como si se tratasen de propiedades del CLR, al mismo tiempo que nos beneficiamos de la funcionalidad adicional ofrecida por la definición de la propiedad de dependencia. Obsérvese que *NombrePropiedad***Property** identifica la propiedad de dependencia *NombrePropiedad*.

Por convenio, la propiedad de dependencia tiene un nombre básico, *NombrePropiedad* en este ejemplo, que se proporciona como primer parámetro de **Register** y un identificador formado por el nombre anterior y el sufijo **Property**. El identificador de tipo **DependencyProperty** será definido como un campo **Public**, **Shared** y **Readonly**.

No obstante, en la mayoría de los casos, no tendrá que preocuparse de las propiedades de dependencia. En la referencia del SDK, puede identificar qué propiedad es una propiedad de dependencia por la presencia de la sección de *Información de propiedad de dependencia* en la página correspondiente a la propiedad.

Crear un elemento

La implementación de un elemento (una ventana o un control) abarca tanto su aspecto como su comportamiento, donde el aspecto define la apariencia del elemento para los usuarios y el comportamiento define cómo funciona ese elemento cuando los usuarios interactúan con él. En WPF, puede implementar el aspecto y el comportamiento de un elemento utilizando el marcado XAML. Sin embargo, en general, el aspecto de un elemento se implementa utilizando el marcado XAML, y su comportamiento se implementa utilizando código subyacente, por ejemplo Visual Basic, como se describe en los apartados siguientes.

Controles más comunes

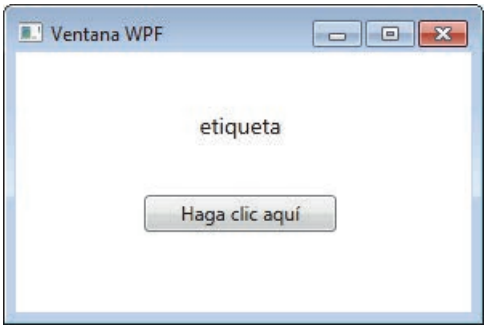
Hay controles para cada uno de los elementos que usted ya ha visto, más de una vez, en alguna ventana de la interfaz gráfica de su sistema Windows, Linux u otro. A continuación se muestra de forma resumida una lista de los más comunes:

- *Etiquetas*. Se implementan a partir de la clase **Label**.
- *Botones*. Se implementan a partir de la clase **Button**.
- *Cajas de texto*. Se implementan a partir de la clase **TextBox** las de una sola línea de texto, las de varias líneas y las de "palabra de paso".
- *Bloques de texto*. Se implementan a partir de la clase **TextBlock**. Se trata de un control ligero, preparado específicamente para integrar pequeños fragmentos de contenido dinámico.
- *Bordes*. Se implementan a partir de la clase **Border**. Dibuja alrededor de otro elemento un borde, un fondo o ambos.
- *Casillas de verificación*. Se implementan a partir de la clase **CheckBox**.
- *Botones de opción*. Se implementan a partir de la clase **RadioButton**.
- *Listas*. Se implementan a partir de las clases **ListBox** y **ComboBox**.
- *Barras de desplazamiento*. Se implementan a partir de la clase **ScrollBar**.

Los controles descritos se localizan en el espacio de nombres **System.Windows.Controls** y son objetos de subclases de la clase **Control** que a su vez se deriva de la clase **System.Windows.FrameworkElement**. La funcionalidad común

de los controles disponibles en WPF está implementada en su mayoría en estas cuatro clases: **UIElement**, **ContentElement**, **FrameworkElement** y **FrameworkContentElement**, todas ellas derivadas indirectamente de **DependencyObject**.

Como ejemplo, vamos a añadir a la ventana de la aplicación anterior dos controles: una etiqueta y un botón de pulsación.



Añadir una etiqueta y editar sus propiedades

Para añadir una etiqueta al formulario proporcionado por el elemento *VentanaWPF* siga estos pasos:

1. Añada a *VentanaWPF* un control de tipo **Label**:

```
<Label Content="Label" Name="label1" />
```

2. Modifique las propiedades de la etiqueta para asignarle el nombre "etSaludo" y para que muestre inicialmente el texto "etiqueta", centrado, con estilo normal y tamaño 14. Para ello, complete la línea anterior como se indica a continuación:

```
<Label Content="etiqueta" Name="etSaludo"
  HorizontalContentAlignment="Center"
  VerticalContentAlignment="Center"
  FontStyle="Normal" FontSize="14" />
```

   La propiedad **Name** permite identificar el control en el código. La propiedad **Content** se utiliza para establecer el contenido de los controles que heredan de **ContentControl**; dicho contenido puede ser cualquier objeto. Observe que en nuestro ejemplo, **Content** almacena el texto que mostrará el control.

   Las propiedades **FontSize** y **FontStyle** indican el tamaño y el estilo, respectivamente, de la fuente que utilizará el control para mostrar su texto.

La propiedad **HorizontalContentAlignment** indica el tipo de alineación que se aplicará al contenido respecto a los límites del control; por ejemplo, el valor *Center* de la enumeración **HorizontalAlignment** del espacio de nombres **System.Windows** indica que el contenido será centrado horizontalmente. Ídem para **VerticalContentAlignment** pero verticalmente.

3. Finalmente establezca su tamaño y su posición en el contenedor.

```
<Label Content="etiqueta" Name="etSaludo"
  HorizontalContentAlignment="Center"
  VerticalContentAlignment="Center"
  FontStyle="Normal" FontSize="14"
  Width="120" Height="30"
  HorizontalAlignment="Left" VerticalAlignment="Top"
  Margin="80,30,0,0" />
```

Las propiedades **Width** y **Height** hacen referencia al tamaño (ancho y alto) del control.

Las propiedades **HorizontalAlignment** y **VerticalAlignment** controlan la posición de un elemento dentro de su contenedor y **Margin** define el espacio que se debe dejar entre este elemento y los elementos adyacentes (80 unidades desde la izquierda, *Left*, y 30 desde arriba, *Top*). **Margin** hace referencia al margen exterior de un elemento.

Los atributos que no se fijan tomarán su valor predeterminado.

Añadir un botón de pulsación y editar sus propiedades

Para añadir un botón de pulsación los pasos son similares a los expuestos para añadir una etiqueta (las propiedades que se vayan repitiendo no las volveremos a comentar por tener un significado análogo; para más detalles recurrir a la ayuda proporcionada por *Microsoft Developer Network* - MSDN). El siguiente código crea un botón de pulsación titulado "Haga clic aquí" y establece como tecla de acceso la *c* (se coloca un _ antes de la letra *c*); de esta forma es posible pulsar el botón, además de haciendo clic sobre el mismo utilizando el ratón, pulsando las teclas *Alt+c*. A continuación, se asigna al botón una descripción abreviada.

1. Añada al formulario *VentanaWPF* un control de tipo **Button** denominado *btSaludo*:

```
<Button Content="Button" Name="btSaludo" />
```

2. Modifique sus propiedades según se indica a continuación:

```
<Button Content="Haga _clic aquí" Name="btSaludo"
   Height="23" Width="120"
   HorizontalAlignment="Left" VerticalAlignment="Bottom"
   Margin="80,0,0,50" />
```

Añadir una descripción abreviada a un elemento

Una descripción abreviada se mostrará cuando el puntero del ratón se sitúe encima del control. Para añadir una descripción abreviada asigne a la propiedad **ToolTip** del control el objeto que almacena dicha descripción según se indica a continuación (este objeto puede ser una simple cadena de caracteres):

```
<Button Content="Haga _clic aquí" Name="btSaludo"
   Height="23" Width="120"
   HorizontalAlignment="Left" VerticalAlignment="Bottom"
   Margin="80,0,0,50"
   ToolTip="Botón de pulsación" />
```

Paneles de diseño

Un requisito fundamental de cualquier diseño es que la posición y el tamaño de los controles (elementos secundarios) puedan adaptarse a los cambios de tamaño del contenedor (elemento primario) y a la configuración de pantalla. Para ello, WPF proporciona un *sistema de diseño* que es invocado automáticamente cada vez que es necesario reorganizar los controles del contenedor: los controles indican a su contenedor qué ubicación y tamaño necesitan y el contenedor responde indicando a los controles de qué espacio dispone.

El sistema de diseño se expone a los elementos secundarios mediante las clases base de WPF. Para los diseños comunes, como son las cuadrículas, el apilamiento y el acoplamiento, WPF incluye varios controles de diseño derivados del control **Panel**, alguno de los cuales describimos a continuación.

Cada tipo de panel ofrece un mecanismo de diseño sencillo, pero como WPF permite componer elementos combinando otros y de muchas maneras diferentes, esta forma flexible en la que se pueden combinar los elementos pone a nuestra disposición un sistema de diseño muy potente.

Canvas

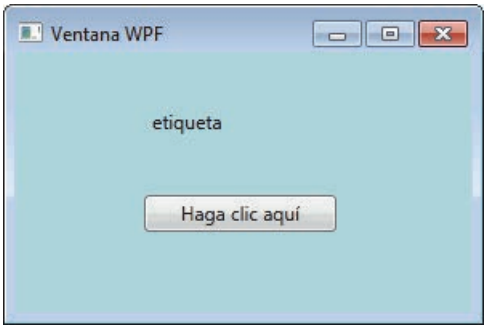
Es un contenedor que define un área en la que pueden colocarse explícitamente los elementos secundarios utilizando coordenadas relativas al área definida por este control. Por ejemplo:

```
<Window x:Class="WpfAplicacion.MainWindow"
```

```
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="200" Width="300" Name="VentanaWPF">
  <Canvas Background="LightBlue" >
    <Label Canvas.Left="80" Canvas.Top="30"
      Content="etiqueta" Name="label1" Width="120" Height="30" />
    <Button Canvas.Left="80" Canvas.Bottom="50"
      Content="Haga clic aquí" Name="button1"
      Height="23" Width="120" />
  </Canvas>
</Window>
```

Cuando a un atributo se le asigna un valor directamente entre comillas, no significa que el valor sea un **String**. Por ejemplo, cuando escribimos *<Label Width="120"... />* "120" corresponde al valor entero 120 en el CLR de .NET, y cuando escribimos *<Canvas Background="LightBlue"... />* el **String** realmente corresponde al objeto brocha *Brushes.LightBlue*. WPF se encarga de hacer las transformaciones al tipo apropiado para ese atributo.

Este código da lugar al formulario siguiente:

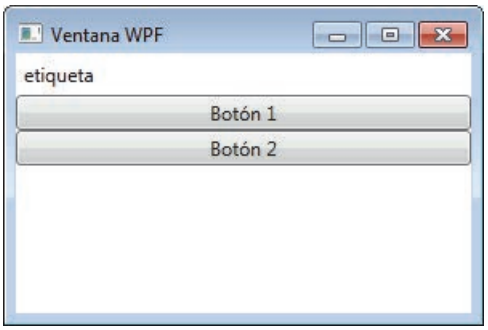


StackPanel

Con este contenedor los elementos secundarios se apilan vertical u horizontalmente. Por ejemplo:

```
<Window x:Class="WpfAplicacion.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="200" Width="300" Name="VentanaWPF">
  <StackPanel Orientation="Vertical">
    <Label Content="etiqueta" Name="label1" />
    <Button Content="Botón 1" Name="button1" />
    <Button Content="Botón 2" Name="button2" />
  </StackPanel>
</Window>
```

Este código da lugar al formulario siguiente:

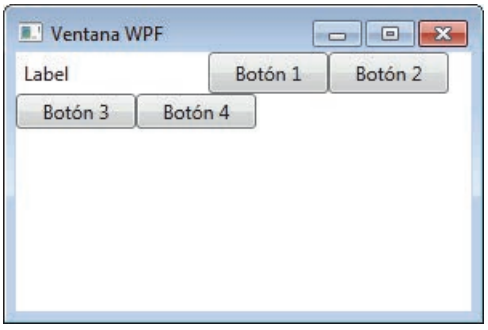


WrapPanel

Con este contenedor los elementos secundarios se sitúan por orden de izquierda a derecha y se ajustan a la línea siguiente cuando hay más controles de los que caben en la línea actual. Por ejemplo:

```
<Window x:Class="WpfAplicacion.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="200" Width="300" Name="VentanaWPF">
  <WrapPanel>
    <Label Content="Label" Name="label1" Width="120" />
    <Button Content="Botón 1" Name="button1" Width="75" />
    <Button Content="Botón 2" Name="button2" Width="75" />
    <Button Content="Botón 3" Name="button3" Width="75" />
    <Button Content="Botón 4" Name="button4" Width="75" />
  </WrapPanel>
</Window>
```

Este código da lugar al formulario siguiente:

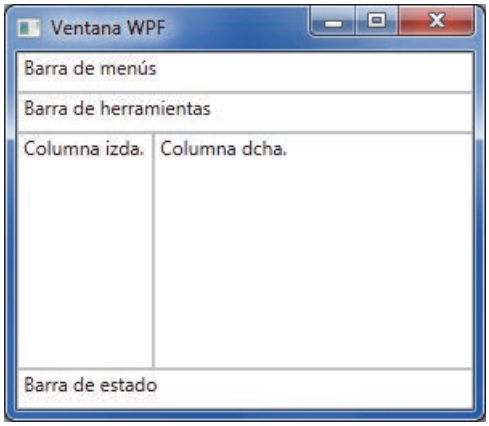


DockPanel

Con este contenedor los elementos secundarios se alinean con los bordes del mismo. Por defecto se alinean a la izquierda. Por ejemplo:

```
<Window x:Class="WpfAplicacion.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="260" Width="300" Name="VentanaWPF">
  <DockPanel>
    <Border DockPanel.Dock="Top"
            BorderBrush="Silver" BorderThickness="1" Height="25">
      <TextBlock Name="textBlock1" Text=" Barra de menús" />
    </Border>
    <Border DockPanel.Dock="Top"
            BorderBrush="Silver" BorderThickness="1" Height="25">
      <TextBlock Name="textBlock2" Text=" Barra de herramientas" />
    </Border>
    <Border DockPanel.Dock="Bottom"
            BorderBrush="Silver" BorderThickness="1" Height="25">
      <TextBlock Name="textBlock3" Text=" Barra de estado" />
    </Border>
    <Border DockPanel.Dock="Left"
            BorderBrush="Silver" BorderThickness="1" Width="85">
      <TextBlock Name="textBlock4" Text=" Columna izda." />
    </Border>
    <Border BorderBrush="Silver" BorderThickness="1">
      <TextBlock Name="textBlock5" Text=" Columna dcha." />
    </Border>
  </DockPanel>
</Window>
```

Este código da lugar al formulario siguiente:



Grid

Este contenedor define un área de cuadrícula flexible, compuesta por filas y columnas, para situar los elementos.

Puede ser utilizado para lograr casi todo lo que se puede hacer con los otros contenedores, aunque, en algunos casos, no con la misma facilidad. Usando XAML, **Grid** permite definir filas (*rows*) y columnas (*columns*) en un panel y luego colocar los elementos secundarios haciendo referencia a los atributos **Row** y **Column** del **Grid**. Los controles pueden abarcar varias filas (**RowSpan**) o columnas (**ColumnSpan**).

Un **Grid** puede definir el alto de sus filas y el ancho de sus columnas de forma automática en función del tamaño del contenido (por ejemplo, *Height="Auto"* o *Width="Auto"*); también puede especificar un tamaño proporcional al espacio restante disponible empleando, en XAML, el *, 2*,..., o bien especificar un valor absoluto en unidades independientes de dispositivo (1/96 de pulgada por unidad *pixel*: *px*) o no especificarlo, en cuyo caso se supone el valor *.

La propiedad **ColumnDefinitions** del **Grid** define una colección de elementos **ColumnDefinition**, cada uno de los cuales define las propiedades específicas (por ejemplo **Width**) de la columna que representa y la propiedad **RowDefinitions** define una colección de elementos **RowDefinition**, cada uno de los cuales define las propiedades específicas (por ejemplo **Height**) de la fila que representa.

En el ejemplo siguiente, además de ver cómo se especifica el tamaño de las filas y columnas, se puede ver también el uso del atributo **ShowGridLines** del **Grid** para mostrar o no las líneas que delimitan cada celda. Lo aquí expuesto sólo es una pequeña introducción de lo que se puede hacer con un **Grid**.

```
<Window x:Class="WpfAplicacion.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="300" Width="400" Name="VentanaWPF">
  <Grid ShowGridLines="True">

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="2*" />
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

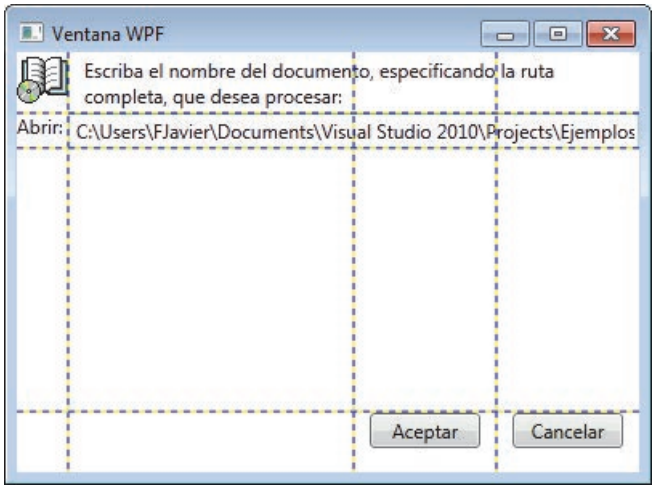
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
```

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

    <Image Grid.Row="0" Grid.Column="0" Source="libro.ico" />
    <TextBlock Padding="10,3,10,3" Grid.Row="0" Grid.Column="1"
               Grid.ColumnSpan="3" TextWrapping="Wrap">
      Escriba el nombre del documento, especificando la ruta
      completa, que desea procesar:
    </TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="0">Abrir:</TextBlock>
    <TextBox Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" />
    <Button Margin="10, 1, 10, 15"
            Grid.Row="3" Grid.Column="2">Aceptar</Button>
    <Button Margin="10, 1, 10, 15"
            Grid.Row="3" Grid.Column="3">Cancelar</Button>
  </Grid>
</Window>
```

Cuando se utiliza asterisco (*) como alto de una fila o ancho de una columna, esa fila o columna recibe una proporción ponderada del espacio restante disponible. Este valor en XAML se expresa como *, 2*,... Por ejemplo, la fila o la columna con 2* recibe del espacio disponible el doble que la fila o la columna con un solo *. Por el contrario, *Auto* distribuye el espacio de manera uniforme basándose en el tamaño del contenido que está dentro de una columna o una fila.

El código XAML del ejemplo anterior da lugar al formulario siguiente:



Ejecute la aplicación, pruebe a modificar el tamaño del formulario y podrá observar cómo la posición y el tamaño de los controles se ajustan al nuevo tamaño del mismo.

Combinando esta técnica con otros controles como **ScrollViewer**, **GridSplitter**, **Viewbox**, etc., puede dar una gran flexibilidad a sus diseños.

Todos los paneles de diseño expuestos, excepto **Canvas**, ajustan la posición y el tamaño de los controles (elementos secundarios) cuando se cambia el tamaño del contenedor (elemento primario) o la configuración de pantalla.

Aplicando la teoría expuesta anteriormente a nuestra aplicación *WPFAplicacion* obtendríamos un código XAML análogo al siguiente:

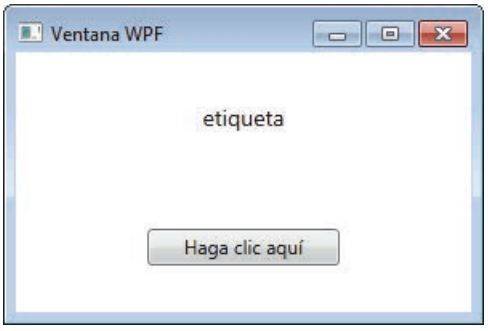
```
<Window x:Class="WPFAplicacion.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ventana WPF" Height="200" Width="300" Name="VentanaWPF">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <Label Grid.Row="0" Content="etiqueta" Name="etSaludo"
      HorizontalContentAlignment="Center"
      VerticalContentAlignment="Center"  Width="120" Height="30"
      FontStyle="Normal" FontSize="14" />

    <Button Grid.Row="1" Content="Haga _clic aquí" Name="btSaludo"
      Height="23" Width="120"
      ToolTip="Botón de pulsación" />
  </Grid>
</Window>
```

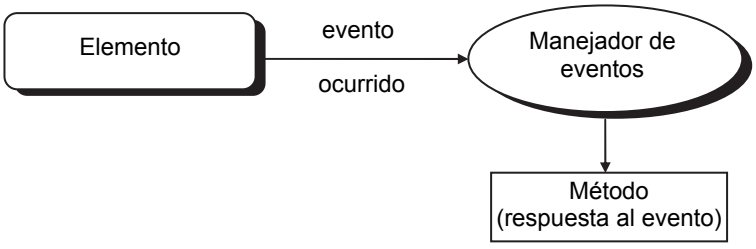
Este código da lugar al formulario siguiente:



El proceso de añadir los elementos resulta muy sencillo cuando utilizamos *Visual Studio* ya que, como puede verse en el apéndice *Entorno de desarrollo integrado*, simplemente tendremos que tomar dichos elementos de una paleta, arrastrarlos sobre el formulario utilizando el ratón y establecer sus propiedades a través de la ventana de propiedades. Esto hará que se añada automáticamente todo el código descrito anteriormente.

MANEJO DE EVENTOS

Cuando una acción sobre un elemento genera un evento, se espera que suceda algo, entendiendo por evento un mensaje que un objeto envía a algún otro objeto. Entonces hay un remitente del evento, por ejemplo, un botón, y hay un receptor del evento, por ejemplo, la ventana que contiene ese botón. Lógicamente ese algo hay que programarlo y para ello, hay que saber cómo manejar ese evento. Los eventos que se producen sobre un elemento se manipulan a través de los manejadores de esos eventos. Un manejador de eventos es un objeto en el que un elemento delega la tarea de manipular un tipo particular de eventos. En la figura siguiente puede ver que, cuando un elemento genera un evento, un manejador de eventos vinculado con el elemento se encarga de responder al mismo ejecutando el método programado para ello (la clase que implementa el manejador de un evento se dice que se suscribe a ese evento publicado por otra clase).



Como sabemos, los elementos de una aplicación WPF se relacionan entre sí a través de un árbol de elementos. Pues bien, cuando una acción sobre un elemento genera un evento, dicho evento, generalmente, partiendo del elemento donde se originó, se propaga en sentido ascendente por el árbol de elementos hasta que llega a la raíz (normalmente una ventana o una página) en busca de un manejador que responda a dicho evento. De aquí que estos eventos se denominen *eventos enrutados*. Por ejemplo, en el árbol de elementos de nuestra aplicación *WPFAplicacion*, un clic sobre el botón generará un evento **Click**. Pues bien, el primer elemento que tiene la oportunidad de controlar el evento es el botón origen del evento, pero si ninguno de los manejadores asociados al mismo responde a ese evento, este se propagará hacia arriba hasta el elemento primario en el árbol de elementos, que es la ventana (la ruta es: elemento **Button** > elemento **Grid** > elemento **Window**).

El manejador de eventos, que recibe también el nombre de *delegado*, es un objeto de la clase **RoutedEventHandler** del espacio de nombres **System.Windows** que guarda una referencia al método que responderá a ese evento. Este método recibe habitualmente el nombre de controlador de eventos.

Asignar manejadores de eventos a un objeto

Un objeto, generalmente un elemento, puede tener asociados uno o más manejadores para responder a un determinado evento.

Para agregar un manejador de eventos en XAML, basta con añadir al elemento el nombre del evento como un atributo y asignarle como valor el nombre del método que manejará el evento:

```
<Button Grid.Row="1" Content="Haga _clic aquí" Name="btSaludo"
  Height="23" Width="120" ToolTip="Botón de pulsación"
  Click="btSaludo_Click" />
```

El método *btSaludo_Click* contendrá el código que responde al evento **Click** y debe tener la misma firma o prototipo que el delegado **RoutedEventHandler**, que es el delegado de los manejadores de eventos enrutados:

```
Public Delegate Sub RoutedEventHandler (
    sender As Object,
    e As RoutedEventArgs
)
```

Por ejemplo, para que la etiqueta *etSaludo* muestre el mensaje "¡¡¡Hola mundo!!!" cuando el usuario de nuestra aplicación haga clic en el botón *btSaludo*, según lo expuesto, basta con añadir a la clase *MainWindow* el siguiente método:

```
Private Sub btSaludo_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  etSaludo.Content = "¡¡¡Hola mundo!!!"
End Sub
```

Ese código XAML que agrega un manejador de eventos es equivalente al código C# subyacente sombreado que se muestra a continuación:

```
Private Sub btSaludo_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs) Handles btSaludo.Click
  etSaludo.Content = "¡¡¡Hola mundo!!!"
End Sub
```

El primer parámetro, *sender*, especifica el objeto al que está asociado el manejador de eventos y el segundo, *e*, especifica los datos para el evento.

Atendiendo al comportamiento de un evento enrutado, no se puede considerar que el parámetro *sender* del evento sea necesariamente el origen del evento. El origen se especifica en la propiedad **Source** de los datos del evento:

```
Dim control As FrameworkElement = TryCast(e.Source, FrameworkElement)
Dim nombre As String = control.Name ' btSaludo
```

El parámetro *sender* y la propiedad **Source** harán referencia al mismo objeto cuando éste coincida con el elemento donde se generó el evento.

```
Dim control As FrameworkElement = TryCast(sender, FrameworkElement)
Dim nombre As String = control.Name ' btSaludo
```

EVENTOS ADJUNTOS

Para permitir que los elementos controlen eventos que se declaran en un elemento diferente, WPF admite lo que denominamos eventos adjuntos. Esto es, mediante un evento adjunto podemos agregar a un elemento cualquiera (por ejemplo, a un **Grid**) un controlador correspondiente a un evento definido en un elemento determinado (por ejemplo, en un **Button**), en lugar de añadirlo al elemento que realmente define o hereda dicho evento. Por ejemplo, si desea que el elemento **Grid** esté atento al paso de un evento **Button.Click** de varios botones, sólo tendría que indicarlo de esta forma:

```
<Grid Button.Click="Boton_Click" Margin="5">
  <Button Name="Boton1" >Acción 1</Button>
  <Button Name="Boton2" >Acción 2</Button>
  <Button Name="Boton3" >Acción 3</Button>
</Grid>
```

INYECTAR CÓDIGO XAML DURANTE LA EJECUCIÓN

En ocasiones puede ser interesante inyectar código XAML durante la ejecución. Este código puede estar almacenado en una base de datos o en un fichero, como veremos en el ejemplo mostrado a continuación. Una manera sencilla de hacer esto es utilizar la funcionalidad de la clase **XamlReader** para leer y analizar el código XAML sobre la marcha. Por ejemplo, supongamos que tenemos el código XAML mostrado a continuación en un fichero denominado *MainWindow.xaml*:

```
<Grid
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
```

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

  <Label Grid.Row="0" Content="etiqueta" Name="etSaludo"
      HorizontalContentAlignment="Center"
      VerticalContentAlignment="Center"  Width="120" Height="30"
      FontStyle="Normal" FontSize="14" />
  <Button Grid.Row="1" Content="Haga _clic aquí" Name="btSaludo"
      Height="23" Width="120"
      ToolTip="Botón de pulsación"/>
</Grid>
```

Cuando añada este fichero al proyecto, indique desde la ventana de propiedades que no se realice ninguna acción de compilación sobre él y que se copie siempre en el directorio de resultados.

El contenido de este fichero se puede cargar durante la ejecución en una ventana ya creada de la clase *MainWindow* derivada de **Window**. A continuación mostramos el código que describe cómo hacerlo:

```
Imports System.IO
Imports System.Windows.Markup

Public Class MainWindow
  Inherits Window
  Private btSaludo As Button
  Private etSaludo As Label

  Public Sub New(ficheroXaml As String)
    InitializeComponent(ficheroXaml)
  End Sub

  Private Sub InitializeComponent(ficheroXaml As String)
    ' Configurar la ventana MainWindow
    Me.Title = "Ventana WPF: XAML dinámico"
    Me.Height = 200
    Me.Width = 300
    ' Cargar el XAML desde un fichero externo
    Dim elementoRaiz As DependencyObject
    Using fs As New FileStream(ficheroXaml, FileMode.Open)
      elementoRaiz = DirectCast(XamlReader.Load(fs), DependencyObject)
    End Using
    ' Insertar el código XAML en la ventana
    Me.Content = elementoRaiz
```

```
    ' Buscar los controles btSaludo y etSaludo
    btSaludo = DirectCast(LogicalTreeHelper.FindLogicalNode(
               elementoRaiz, "btSaludo"), Button)
    etSaludo = DirectCast(LogicalTreeHelper.FindLogicalNode(
               elementoRaiz, "etSaludo"), Label)
    ' Asignar el controlador para manejar el evento Click
    ' de btSaludo
    AddHandler btSaludo.Click, AddressOf btSaludo_Click
  End Sub

  Private Sub btSaludo_Click(sender As Object, e As RoutedEventArgs)
    etSaludo.Content = "¡¡¡Hola mundo!!!"
  End Sub
End Class

Class Application
  Public Shared Sub Main()
    Dim app As New Application()
    app.Run(New MainWindow("MainWindow.xaml"))
  End Sub
End Class
```

Analice el código anterior. El constructor recibe como argumento el nombre del fichero XAML (en nuestro caso, *MainWindow.xaml*) que llama al método *InitializeComponent* pasando este mismo argumento. Este método crea un flujo **FileStream** desde ese fichero y, utilizando el método **Load** de la clase **XamlReader**, convierte el contenido de ese fichero en un objeto de dependencia de la clase **DependencyObject**, clase base de la que se derivan todos los controles, que es asignado como contenido de la ventana *MainWindow*.

A continuación, para trabajar sobre cualquier elemento de la interfaz cargada dinámicamente, en nuestro caso sobre el botón y la etiqueta, necesitamos obtener la referencia a dichos elementos. Eso es lo que hace el método **FindLogicalNode** de la clase **LogicalTreeHelper**; este método busca y devuelve el objeto especificado realizando la búsqueda hacia los objetos secundarios. Una vez obtenidas las referencias, en nuestro caso al botón y a la etiqueta, asignamos el método que manejará el evento **Click** del botón y escribimos el código de este método para que la etiqueta muestre un mensaje.

Otra alternativa para obtener las referencias a los elementos de la interfaz cargada dinámicamente es utilizar el método **FindName** de la clase **FrameworkElement** según se indica a continuación:

```
Dim elementoRaiz As FrameworkElement
Using fs As New FileStream(ficheroXaml, FileMode.Open)
  elementoRaiz = DirectCast(XamlReader.Load(fs), FrameworkElement)
End Using
```

```
' Insertar el código XAML en la ventana
Me.Content = elementoRaiz
' Buscar los controles btSaludo y etSaludo
btSaludo = DirectCast(elementoRaiz.FindName("btSaludo"), Button)
etSaludo = DirectCast(elementoRaiz.FindName("etSaludo"), Label)
```

Obviamente, la carga dinámica de XAML no será tan eficiente como compilar el XAML y luego cargar el BAML durante la ejecución, sobre todo si la interfaz de usuario es compleja. Sin embargo, abre una serie de posibilidades para la creación de interfaces de usuario dinámicas.

CICLO DE VIDA DE UNA VENTANA

Como con cualquier objeto de cualquier clase, el ciclo de vida de una ventana comienza cuando se crea por primera vez un objeto de su clase, después se abre, se activa, se desactiva y, finalmente, se cierra.

Cuando se crea una ventana se agrega automáticamente su referencia a la colección de ventanas referenciada por la propiedad **Windows** del objeto **Application** y si además se trata de la primera ventana, se establece, de forma predeterminada, como la ventana principal de la aplicación, asignando su referencia a la propiedad **MainWindow** del objeto **Application**; esto está implícito cuando se establece la propiedad **StartupUri**, según vimos anteriormente.

Cuando se inicia la aplicación, la ventana especificada por el valor de **StartupUri** se abre de forma no modal (internamente, la ventana se abre llamando a su método **Show**). Una ventana *no modal* permite a los usuarios activar otras ventanas en la misma aplicación; lo contrario sería una ventana *modal*, que se abre con **ShowDialog**.

Cuando se abre una ventana, ésta se convierte en la ventana activa y se genera el evento **Activated** (la ventana activa es aquella que está capturando los datos proporcionados por el usuario, tales como las teclas pulsadas y los clics del ratón) y a continuación se generan los eventos **Loaded** y **ContentRendered**, en este orden; cuando se produce este último evento puede considerarse la ventana abierta. También se puede especificar la posición inicial de la ventana la primera vez que se muestra estableciendo su propiedad **WindowStartupLocation**.

Cuando una ventana está activa, un usuario puede activar otra ventana de la misma aplicación o activar otra aplicación. Entonces, la ventana activa pasa a estar desactiva produciéndose el evento **Deactivated** y cuando se vuelva a activar, se volverá a producir el evento **Activated**. Para verificar si una ventana está activa se puede inspeccionar la propiedad **IsActive**.

Cuando se desactiva una ventana puede que la aplicación continúe ejecutando código en segundo plano. En este caso, cuando se complete la tarea en segundo plano es posible avisar al usuario invocando al método **Activate**, lo cual hará parpadear el botón de la barra de tareas de la ventana en el supuesto de que el usuario esté interactuando con otra aplicación o traerá la ventana al primer plano si el usuario está interactuando con la aplicación actual.

Una ventana minimizada se contrae en un botón en la barra de tareas si su propiedad **ShowInTaskbar** vale **True**.

Cuando se cierra una ventana se generan los eventos **Closing**, antes de que la ventana se cierre y con la posibilidad de detener esta acción, **Deactivated** y **Closed**; este último evento se genera justo en el instante en el que la ventana se va a cerrar y sin posibilidad de detener esta acción.

Puede probar el ciclo de vida de la ventana de la aplicación *WPFAplicacion* añadiendo los controladores para los distintos eventos comentados. Por ejemplo:

```
Private Sub VentanaWPF_Loaded(sender As Object, e As RoutedEventArgs)
  System.Diagnostics.Debug.WriteLine("Gestión del evento Loaded")
End Sub
```

Una vez añadidos estos controladores, ejecute la aplicación en modo depuración (*F5*) para ver los distintos mensajes en la ventana de resultados.

PROPIEDADES BÁSICAS DE LA VENTANA

Anteriormente hemos visto algunas propiedades de la ventana y también hemos hablado del ciclo de vida de la misma, pero no hemos hablado mucho de sus propiedades, que es lo que vamos a hacer a continuación.

Una ventana WPF se compone de dos áreas distintas: *área no cliente* y *área cliente*. El *área no cliente* comprende los elementos gráficos comunes a todas las ventanas: menú del sistema, icono, título, botón para minimizar, botón para maximizar, botón para cerrar y un borde, y el *área cliente* es la parte de la ventana destinada a ubicar los controles que formarán la interfaz gráfica.

La clase **Window** proporciona servicios para administrar la duración de la ventana, para administrarla y para establecer su apariencia y comportamiento.

Administración de la duración

A continuación describimos la funcionalidad más destacada relacionada con la duración de la ventana.

- Método **Activate**. Permite activar la ventana situándola en primer plano. Cuando esto sucede se produce el evento **Activated** y cuando se desactiva, porque se active otra ventana, se produce el evento **Deactivated**.

- Método **Close**. Permite cerrar la ventana. Cuando se cierra una ventana se produce el evento **Closing**, se quita el objeto **Window** de la colección **Application.Windows**, se produce el evento **Closed** y se eliminan los recursos no administrados creados por el objeto **Window**.

- Métodos **Show** y **Hide**. **Show** muestra una ventana sin impedir que los usuarios interactúen con otras ventanas de la aplicación y **Hide** la oculta. Cuando se oculta una ventana, ésta no se cierra y la propiedad **Visibility** toma el valor *Visibility.Hidden*.

- Propiedad **IsActive**. Esta propiedad vale **True** si la ventana está activa y **False** en caso contrario. El valor predeterminado es **False**.

Administración de ventanas

A continuación describimos la funcionalidad más destacada relacionada con la administración de las ventanas.

- Método **Shared GetWindow**. Devuelve la referencia al objeto **Window** que define el objeto de dependencia (objeto de la clase **DependencyObject**) pasado como argumento.

- Propiedad **OwnedWindows**. Referencia la colección **WindowCollection** de ventanas de las que esta ventana es la propietaria.

- Propiedad **Owner**. Obtiene o establece la referencia a la ventana propietaria de este objeto **Window**.

Apariencia y comportamiento

A continuación describimos la funcionalidad más destacada relacionada con la apariencia y comportamiento de las ventanas.

- Propiedad **AllowsTransparency**. Esta propiedad vale **True** si la ventana admite la transparencia y **False** en caso contrario.

- Método **DragMove**. Permite arrastrar una ventana haciendo clic con el ratón en cualquier parte de la ventana, no necesariamente en la barra de título. Para ello, cuando se llama a **DragMove** el botón primario del ratón debe estar presionado para lo cual puede invocar a este método desde el controlador del evento **MouseLeftButtonDown**.

- Propiedad **Icon**. Permite establecer el icono de una ventana.

- Propiedades **Top** y **Left**. Estas propiedades permiten obtener o establecer la posición de los bordes superior e izquierdo de la ventana, respectivamente, con respecto al escritorio. Por lo tanto, podrían ser utilizadas para establecer la posición inicial de la ventana.

- Evento **LocationChanged**. Se produce cuando una ventana cambia de posición.

- Propiedad **ResizeMode**. El tamaño de todas las ventanas se puede cambiar. Sin embargo, este comportamiento puede ser modificado por medio de la propiedad **ResizeMode** que puede tomar estos valores: *NoResize*, no se puede cambiar el tamaño de la ventana, *CanMinimize*, la ventana sólo se puede minimizar y restaurar desde la barra de tareas, *CanResize*, se tiene total capacidad para cambiar el tamaño de la ventana, y *CanResizeWithGrip* que ofrece la misma funcionalidad que *CanResize*, pero agrega un "control de ajuste de tamaño" en la esquina inferior derecha de la ventana.

- Propiedad **RestoreBounds**. Esta propiedad se puede usar para guardar el tamaño y la ubicación (objeto **Rect**) de una ventana antes de que se cierre la aplicación y recuperar esos valores la próxima vez que se inicie la aplicación para que la ventana se muestre con esos valores.

- Propiedad **ShowActivated**. Cuando se abre una ventana que tiene esta propiedad establecida en **False**, la ventana no se activa hasta que se active manualmente al seleccionarla.

- Propiedad **ShowInTaskbar**. Si el valor de esta propiedad es **True**, valor predeterminado, y se minimiza la ventana, ésta se muestra en la barra de tareas.

- Propiedad **SizeToContent**. El valor de esta propiedad indica si una ventana ajustará automáticamente su tamaño al de su contenido. Su valor predeterminado es *Manual*, esto es, el tamaño se ajustará manualmente; este valor se es-

tablece automáticamente siempre que el usuario cambie el tamaño de la ventana. Ahora bien, si su valor es *WidthAndHeight*, establecer la propiedad **Height** o **Width** no tiene ningún efecto y el tamaño se ajustará automáticamente; análogamente, si su valor es *Height*, establecer la propiedad **Height** no cambiará el alto de la ventana y si su valor es *Width*, establecer la propiedad **Width** no cambiará el ancho de la ventana.

- Propiedad **Topmost**. Orden Z de las ventanas. Cuando esta propiedad vale **True**, la ventana correspondiente aparece encima de todas las ventanas cuya propiedad **Topmost** valga **False** y dentro de las ventanas que tienen esta propiedad a **True**, la ventana actualmente activada es la ventana de nivel superior.

- Propiedad **WindowStartupLocation**. Permite obtener o establecer la posición de una ventana cuando se muestra por primera vez. Si su valor es *Manual*, valor predeterminado, la posición de la ventana queda definida por sus propiedades **Top** y **Left**; si estas propiedades no se especifican, entonces Windows determinará sus valores. Si su valor es *CenterScreen*, la ventana se coloca en el centro de la pantalla que contiene el cursor del ratón. Y si su valor es *CenterOwner*, la ventana se coloca en el centro de la ventana propietaria si se especificó alguna.

- Propiedad **WindowState**. El valor de esta propiedad indica si una ventana está restaurada (*Normal*), minimizada (*Minimized*) o maximizada (*Maximized*).

- Evento **StateChanged**. Se produce cuando cambia la propiedad **WindowState** de la ventana.

- Propiedad **WindowStyle**. Indica el estilo del borde de una ventana. Su valor puede ser: *None*, sólo está visible el área cliente pero tiene un pequeño borde para poder cambiar de tamaño, *SingleBorderWindow*, borde único (es el valor predeterminado), *ThreeDBorderWindow*, borde 3D, y *ToolWindow*, ventana de herramientas fija.

CONFIGURACIÓN DE UNA APLICACIÓN

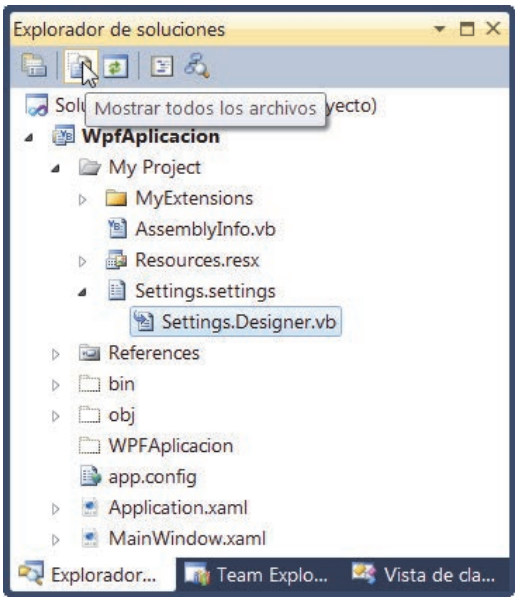
El ejemplo que se muestra a continuación indica como al iniciar una aplicación se pueden restaurar los valores de tamaño, ubicación y estado de la ventana principal que se guardaron la última vez que se cerró la aplicación. Los valores guardados corresponden a las propiedades **RestoreBounds** y **WindowState**.

```
Imports System.Reflection
```

```
Class MainWindow
  Public Sub New()
    ' Configuración
    InitializeComponent()
    Try
      ' Restaurar el estado desde los atributos del objeto
      ' de la clase MySettings referenciado por Default
      Dim restoreBounds As Rect = MySettings.Default.MainRestoreBounds
      Left = restoreBounds.Left
      Top = restoreBounds.Top
      Width = restoreBounds.Width
      Height = restoreBounds.Height
      WindowState = MySettings.Default.MainWindowState
    Catch
    End Try
    ' Recursos
    Me.Title = My.Resources.TituloAplicacion
  End Sub

  Private Sub VentanaWPF_Closing(sender As Object,
  e As System.ComponentModel.CancelEventArgs)
    ' Guardar el estado desde los atributos
    MySettings.Default.MainRestoreBounds = RestoreBounds
    MySettings.Default.MainWindowState = WindowState
    MySettings.Default.Save()
  End Sub
End Class
```

El código anterior requiere un par de parámetros de configuración que hemos denominado *MainRestoreBounds* de tipo **System.Windows.Rect** y *MainWindowState* de tipo **System.Windows.WindowState**. Para añadirlos ejecute *Proyecto > Propiedades > Configuración*. Estos parámetros quedarán registrados en el fichero *app.config* y serán definidos como propiedades de la clase *MySettings* localizada en el fichero *Settings.Designer.vb* de la carpeta *My Project* del proyecto, según muestra la figura siguiente (haga clic en el botón *Mostrar todos los archivos* de la barra de herramientas del explorador de soluciones), y pueden ser recuperados a través de las propiedades del mismo nombre de dicha clase. En el código anterior, *Default* es una propiedad **Shared** que representa el objeto *Settings* que da acceso a las propiedades mencionadas (para más detalles vea en el proyecto la definición de esta clase).



Observe que cuando se inicia la aplicación, el constructor de la ventana principal fija la posición y el tamaño de la ventana, así como el estado de la misma, con los valores guardados cuando se cerró la ventana por última vez. ¿Cuándo salvamos estos valores? Cuando se solicite cerrar la ventana: evento **Closing**. En este momento guardamos el valor de la propiedad **RestoreBounds** en el parámetro *MainRestoreBounds* y el valor de la propiedad **WindowState** en el parámetro *MainWindowState*.

RECURSOS DE UNA APLICACIÓN

Para almacenar y administrar recursos específicos de una aplicación, el espacio de nombres **System.Resources** proporciona diversas clases e interfaces, de las cuales la más importante es la clase **ResourceManager**. La funcionalidad proporcionada por esta clase permite al usuario acceder y controlar los recursos almacenados en el ensamblado principal o en ensamblados satélite de recursos; por ejemplo, los métodos **GetObject** y **GetString** permiten recuperar, respectivamente, objetos y cadenas específicos.

Como ejemplo, vamos a almacenar el título de la ventana como un recurso *TituloAplicacion* de tipo **String**. Para añadirlo ejecute *Proyecto* > *Propiedades* > *Recursos*. Después, podrá recuperar este recurso así:

```
Public Sub New()
  InitializeComponent()
  ...
```

```
  Me.Title = My.Resources.TituloAplicacion
End Sub
```

Este recurso ha sido almacenado en el fichero XML *Resources.resx* de la carpeta *My Project* del proyecto, según muestra la figura anterior, y para acceder al mismo, utilizaremos la funcionalidad proporcionada por la clase *Resources* del espacio de nombres *Resources* localizada en el fichero de código subyacente *Resources.Designer.vb*. Si echa una ojeada a esta clase, observará que define una propiedad **Shared** que da acceso al recurso, cuyo código se muestra a continuación, del mismo nombre que el recurso:

```
Friend ReadOnly Property TituloAplicacion() As String
  Get
    Return ResourceManager.GetString("TituloAplicacion",
                                     resourceCulture)
  End Get
End Property
```

Una de las ventajas de definir un recurso así es que podemos modificarlo editando el fichero XML *Resources.resx* sin que sea necesario volver a compilar la aplicación.

ATRIBUTOS GLOBALES DE UNA APLICACIÓN

Los atributos globales se aplican a todo un ensamblado. Su sintaxis es:

```
<assembly: atributo("valor")>
```

Por ejemplo, el fichero *AssemblyInfo.vb* localizado en la carpeta *Properties* del proyecto, según puede ver en la figura anterior, define los siguientes atributos cuyo valor puede usted editar:

```
<assembly: AssemblyTitle("WPFAplicacion")>
<assembly: AssemblyDescription("Arquitectura de una aplicación WPF")>
<assembly: AssemblyCompany("FJCS")>
<assembly: AssemblyProduct("WPFAplicacion")>
<assembly: AssemblyCopyright("Copyright © Fco. Javier Ceballos")>
<assembly: AssemblyTrademark("")>
...
<assembly: AssemblyFileVersion("1.0.0.0")>
```

Estos atributos globales se pueden especificar después de las directivas de nivel superior **Imports** y antes de las declaraciones de tipo o de espacio de nombres.

Definir estos atributos en el código fuente tendría poco valor si no se dispusiese de un método para recuperar la información que guardan y actuar sobre ella. Pues bien, esto puede hacerse mediante el uso de la reflexión: característica que permite almacenar y obtener información durante la ejecución sobre casi cualquier objeto o tipo presente en un módulo gracias a la funcionalidad proporcionada por las clases del espacio de nombres **System.Reflection**.

A este espacio de nombres pertenece la clase **Assembly**. Un objeto de esta clase representa un ensamblado; concretamente, el ensamblado de la aplicación actualmente en ejecución es proporcionado por el método **Shared GetExecutingAssembly**. El método más importante del ensamblado es **GetCustomAttributes**, el cual devuelve una matriz de objetos que son los equivalentes durante la ejecución de los atributos del código fuente. Por ejemplo, el siguiente código obtiene la información dada por el atributo **AssemblyTitleAttribute**:

```
Dim atributos As Object() = Assembly.GetExecutingAssembly().
  GetCustomAttributes(GetType(AssemblyTitleAttribute), False)
Dim titulo As String = DirectCast(atributos(0),
  AssemblyTitleAttribute).Title
```

De acuerdo con los atributos globales definidos en el ejemplo anterior, el *titulo* obtenido sería *WPFAplicacion*.

Esta técnica la podemos aplicar para mostrar los créditos de una aplicación en un diálogo (véase el apartado *Diálogo acerca de* del capítulo *Controles y cajas de diálogo*).

CICLO DE VIDA DE UNA APLICACIÓN

Cuando se ejecuta una aplicación y se invoca al método **Run** se genera el evento **Startup** del objeto **Application**. A continuación, cuando la aplicación pasa a primer plano se genera el evento **Activated** y cuando vuelve a segundo plano se produce el evento **Deactivated**. Finalmente, cuando cerramos la aplicación se genera el evento **Exit**. También, la aplicación genera el evento **SessionEnding** cuando cerramos la sesión Windows y el evento **DispatcherUnhandledException** cuando se lanza una excepción no controlada en cualquier parte de la aplicación.

El código mostrado a continuación es la forma más simple de poner en marcha una aplicación:

```
Partial Public Class Application
  Inherits System.Windows.Application
  Public Shared Sub Main()
    Dim app As Application = New Application()
```

```
    app.Run(New MainWindow())
  End Sub
End Class
```

Cuando se invoca el método **Run** pasando como argumento una ventana, esa ventana pasará a ser la ventana principal de la aplicación, asignando su referencia a la propiedad **MainWindow** del objeto **Application**. A continuación, el método **Run** lanza el evento **Startup** del objeto aplicación (*app*) y muestra la ventana principal. No obstante, el patrón que Visual Studio utiliza para crear una aplicación WPF no es exactamente el anterior, como ya explicamos anteriormente en este mismo capítulo, sino este otro:

```
Partial Public Class Application
  Inherits System.Windows.Application

  Public Sub InitializeComponent()
      Me.StartupUri = New System.Uri("MainWindow.xaml",
                                     System.UriKind.Relative)
  End Sub

  Public Shared Sub Main()
    Dim app As Application = New Application()
    app.InitializeComponent
    app.Run
  End Sub
End Class
```

La única diferencia introducida por el código generado por Visual Studio es que la clase aplicación utiliza la propiedad **StartupUri**, en lugar de establecer la propiedad **MainWindow** o pasar la ventana principal como argumento al método **Run**. Dicha propiedad se establece en el método *InitializeComponent* para que haga referencia al recurso que define la interfaz de usuario que se mostrará automáticamente cuando se inicia la aplicación. Por lo tanto, si optamos por crear una clase de aplicación personalizada es necesario crear un objeto **Uri** que haga referencia al documento XAML que define la ventana principal, según se puede ver en el código mostrado anteriormente. Este documento XAML será compilado e incrustado como un recurso en el ensamblado de la aplicación.

Cuando el documento XAML corresponde a la ventana principal de la aplicación, éste también es cargado desde el método *InitializeComponent* de la clase de esta ventana, como muestra el código siguiente, pero esto es así porque es el patrón seguido por Visual Studio para generar cualquier ventana WPF que añadamos a la aplicación.

```
Partial Public Class MainWindow
  Inherits System.Windows.Window
```

```
  Private _contentLoaded As Boolean
  Public Sub InitializeComponent()
    If _contentLoaded Then
      Return
    End If
    _contentLoaded = True
    Dim resourceLocater As System.Uri =
      New System.Uri("/WpfAplicacion;component/mainwindow.xaml",
                     System.UriKind.Relative)
    System.Windows.Application.LoadComponent(Me, resourceLocater)
  End Sub
End Class
```

Resumiendo, una aplicación generalmente tendrá varias ventanas y la propiedad **StartupUri** del objeto aplicación hará referencia a la ventana que, entre todas ellas, se tiene que mostrar automáticamente cuando se inicie la aplicación.

Permitir una sola instancia de la aplicación

Si quisiéramos impedir que se creara más de una instancia de nuestra aplicación, básicamente, lo que tendríamos que hacer sería verificar, cuando se desencadene el evento **Startup**, si ya hay otra instancia de la aplicación en ejecución. La forma más sencilla de hacerlo es utilizar un **Mutex** del sistema (un objeto de sincronización que proporciona el sistema operativo que permite la comunicación entre procesos). Según lo expuesto, podríamos escribir el método que responda al evento **Startup** así:

```
Class Application
  Private exmut As System.Threading.Mutex

  Private Sub Application_Startup(sender As Object, e As StartupEventArgs)
    ' Verificar si ya existe una instancia de la aplicación
    Dim nombre_exmut As String = "WPFAplicacion.App"
    Dim nueva As Boolean = False
    exmut = New System.Threading.Mutex(True, nombre_exmut, nueva)
    If Not nueva Then
      Shutdown()
    End If
  End Sub
  ' ...
End Class
```

El siguiente código XAML vincula el método anterior con el evento **Startup** del objeto aplicación:

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

```
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml"
  Startup="Application_Startup">
  <Application.Resources>

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

Observe que la clase *Application* define un atributo *exmut* para referenciar a un objeto de exclusión mutua. Dicho objeto lo creamos con nombre en el método *Application_Startup* para que actúe como una exclusión mutua del sistema, ya que este tipo de **Mutex** es visible en todo el sistema operativo, cosa que no ocurre con la exclusión mutua local sin nombre. La llamada al constructor **Mutex** tiene tres argumentos: el primero es **True** para otorgar al subproceso que realiza la llamada la propiedad inicial del objeto *exmut*, el segundo para asignar un nombre al **Mutex** y el tercero devolverá un valor booleano que será **True** si se creó la exclusión mutua especificada o **False** si ésta ya existe, que es lo que sucederá cuando se intente crear una segunda instancia de la aplicación.

Cómo se genera un evento

Para cada evento del objeto aplicación, con alguna excepción, la clase **Application** proporciona un método que es llamado para generar ese evento. El nombre de este método es el mismo que el nombre del evento con el prefijo **On**; por ejemplo, **Startup** (se genera cuando se llama al método **Run** del objeto **Application**) es generado invocando el método **OnStartup**, y así sucesivamente. Este patrón es muy común en .NET.

Según lo expuesto, otra alternativa para escribir el método anterior sería reescribir el método virtual **OnStartup** heredado de la clase **Application**, que es invocado para generar el evento **Startup**. Es este caso, es una buena idea comenzar llamando a la implementación de ese método en la clase base que, por lo general, no hace más que generar el evento correspondiente a la solicitud.

```
Protected Overrides Sub OnStartup(e As StartupEventArgs)
  MyBase.OnStartup(e)
  ' ...
End Sub
```

Especificar cuándo se cerrará la aplicación

También, estableciendo la propiedad **ShutdownMode** del objeto aplicación se puede especificar cuándo se cerrará la aplicación. Por ejemplo:

```
<Application x:Class="WPFAplicacion.App"
             xmlns="http://schemas.microsoft.com/..."
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml"
             ShutdownMode="OnExplicitShutdown"
>
```

Los valores posibles para esta propiedad son definidos por la enumeración del mismo nombre y son:

- *OnExplicitShutdown*. La aplicación se cierra sólo cuando se llama a **Shutdown** del objeto aplicación.

- *OnLastWindowClose*. La aplicación se cierra cuando se cierra la última ventana o se llama a **Shutdown**. Es el valor predeterminado y es útil cuando la aplicación tiene varias ventanas de nivel superior.

- *OnMainWindowClose*. La aplicación se cierra cuando se cierra la ventana principal o se llama a **Shutdown**. Es útil cuando la aplicación tiene una única ventana de nivel superior.

Pantalla de presentación

En ocasiones, mostrar al usuario la ventana principal de la aplicación puede resultar lento simplemente porque previamente tienen que ejecutarse una serie de pasos necesarios para iniciar dicha aplicación. En estos casos, puede ser una buena idea mostrar una pantalla de presentación para informar al usuario de que la iniciación de la aplicación está en curso.

Una pantalla de presentación puede construirse a partir de un objeto de la clase **SplashScreen** y se muestra utilizando código nativo antes de que se cree la instancia de la aplicación WPF. La pantalla de presentación se muestra en el centro de la pantalla de la máquina del usuario y cuando la aplicación se carga, la pantalla de presentación se desvanece hasta desaparecer.

Para añadir una pantalla de presentación a la aplicación WPF siga los pasos indicados a continuación:

1. Añada una imagen al proyecto (por ejemplo, *PantallaDePresentacion.png*).
2. Seleccione el fichero con la imagen en el explorador de soluciones.
3. En la ventana de propiedades, especifique *SplashScreen* como acción de compilación para ese fichero.

Ahora, cuando ejecute la aplicación será mostrada inmediatamente la imagen en el centro de la pantalla.

La acción de añadir una pantalla de presentación genera el siguiente código al principio del método **Main**:

```
Public Shared Sub Main()
  Dim splashScreen As SplashScreen =
    New SplashScreen("imagenes/pantalladepresentacion.png")
  splashScreen.Show(true)
  Dim app As Application = New Application()
  app.InitializeComponent
  app.Run
End Sub
```

Argumentos en la línea de órdenes

Otra de las tareas de iniciación que puede hacer una aplicación WPF es procesar los argumentos pasados en la línea de órdenes. Estos parámetros están disponibles en la propiedad **Args** del objeto de tipo **StartupEventArgs** que se pasa al controlador del evento **Startup**. La propiedad **Args** está definida como una matriz unidimensional de objetos **String**.

Por ejemplo, supongamos que queremos dar al usuario de una aplicación la opción de iniciar la misma con la ventana principal maximizada. Esto sería fácil hacerlo arrancando la aplicación desde la línea de órdenes con una opción que indique tal acción. Por ejemplo:

```
WPFAplicacion -max
```

Para realizar el proceso descrito anteriormente, vamos a añadir a la clase aplicación una propiedad *VentanaMaximizada* que devuelva **True** si se pasó el argumento */max* o *–max*. A continuación, en el controlador del evento **Startup** verificamos si se pasó algún argumento y, en caso afirmativo, si ese argumento (*Args[0]*) es */max* o *–max*, en cuyo caso asignamos al atributo *ventanaMaximizada* el valor **True**.

```
Class Application
  Private m_ventanaMaximizada As Boolean = False

  Public ReadOnly Property VentanaMaximizada() As Boolean
    Get
      Return m_ventanaMaximizada
    End Get
  End Property
```

```
  Private Sub Application_Startup(sender As Object,
  e As StartupEventArgs)
    ' Argumentos en la línea de órdenes
    If e.Args.Length > 0 Then
      If e.Args(0) = "/max" OrElse e.Args(0) = "-max" Then
        m_ventanaMaximizada = True
      End If
    End If
  End Sub
End Class
```

Finalmente, en el constructor de la ventana principal, según se muestra en el ejemplo del siguiente apartado, preguntaremos por el valor de la propiedad *VentanaMaximizada* para conocer si el usuario solicitó iniciar la aplicación con la ventana principal maximizada.

Acceso a la aplicación actual

Para que las ventanas de una aplicación puedan comunicarse entre sí o con el objeto aplicación, es necesario tener acceso a este objeto desde cualquier parte de la aplicación. Pues bien, podemos obtener una referencia al objeto aplicación por medio de la propiedad estática **Current** de la clase **Application**. Por ejemplo, la siguiente línea de código obtiene una referencia *app* al objeto **Application**:

```
Dim app As Application = DirectCast(Application.Current, Application)
```

Esto permite a las ventanas de una aplicación comunicarse entre ellas, ya que cualquiera de ellas puede acceder a la propiedad **MainWindow** del objeto aplicación y obtener una referencia a la ventana principal. Por ejemplo, suponiendo que *MainWindow* es la clase de la ventana principal, la siguiente línea de código obtiene una referencia *ventanaPpal* a dicha ventana:

```
Dim ventanaPpal As MainWindow = DirectCast(app.MainWindow, MainWindow)
```

Continuando con el ejemplo anterior, en el constructor de la ventana principal, preguntaremos por el valor de la propiedad *VentanaMaximizada* del objeto aplicación para saber si hay que mostrar la ventana principal maximizada, lo que implica asignar a su propiedad **WindowState** el valor **Maximized**.

```
Class MainWindow
  Public Sub New()
    InitializeComponent()
    ' Maximizar la ventana si se pasó el argumento /max o -max
    Dim app As Application = DirectCast(Application.Current, Application)
    If app.VentanaMaximizada Then
      VentanaWPF.WindowState = WindowState.Maximized
```

```
    End If
  End Sub
  ' ...
End Class
```

RESUMEN

En este capítulo se ha expuesto cómo es la estructura de una aplicación WPF que presenta una interfaz gráfica y cómo se diseña dicha interfaz. Esto ha requerido crear elementos XAML, por lo que se ha hecho una introducción a XAML, y explicar lo que son los paneles de diseño, por lo que también se ha hecho una introducción al diseño. Una vez diseñada la interfaz, se ha explicado cómo manipular los eventos que desencadenarán los objetivos perseguidos. Otros conceptos definidos son: tecla de acceso y descripción abreviada asociadas con un control, eventos adjuntos, inyectar código XAML dinámicamente, ciclo de vida de una ventana, propiedades básicas de una ventana, configuración y recursos de la aplicación, ciclo de vida de la aplicación y tareas comunes que podemos ejecutar durante la iniciación de una aplicación, entre otros.

EJERCICIOS PROPUESTOS

1. Modifique la aplicación *WPFAplicacion* desarrollada anteriormente en este capítulo, para que presente una interfaz gráfica igual, pero utilizando administradores de diseño diferentes al **Grid**.

   Para agregar componentes a una ventana cuando se utilizan paneles de diseño, en muchas ocasiones es aconsejable utilizar *contenedores* intermedios; esto facilitará la realización de otras operaciones posteriores como, por ejemplo, añadir un borde alrededor de los componentes.

   Siempre que sea necesario, un contenedor puede incluir otros contenedores, lo que dará lugar a una jerarquía de contenedores que facilitará la distribución de los componentes. La raíz de esa jerarquía es el *contenedor del nivel superior* definido por la ventana.

   Haga que el texto cambie de color cada vez que el usuario haga clic sobre el botón.

CAPÍTULO 2

© F.J.Ceballos/RA-MA

INTRODUCCIÓN A WPF

Según se expuso en el capítulo anterior, WPF proporciona una biblioteca de clases para crear sistemas de presentación dinámicos y controlados por datos. Las aplicaciones tradicionales crean una presentación y, a continuación, enlazan a algunos datos. En WPF, todo lo concerniente al control, cada aspecto de la presentación, está generado por algún tipo de enlace de datos. Por ejemplo, el texto situado en un botón se muestra creando un control compuesto dentro de éste y enlazando su presentación a la propiedad de contenido del botón.

A continuación se proporciona un resumen de la jerarquía de clases de *Windows Presentation Foundation* (WPF), explicando los aspectos más fundamentales de la misma, así como una explicación de los elementos WPF más comunes y de las operaciones más frecuentes que intervienen al utilizarlos.

CLASES WPF

Muchas de las clases de la biblioteca WPF se derivan de estas cuatro clases: **UIElement**, **FrameworkElement**, **ContentElement** y **FrameworkContentElement** del espacio de nombres **System.Windows**. A continuación mostramos la posición de estas clases en la jerarquía de clases WPF y su significado:

- System.Object
  - System.Windows.Threading.DispatcherObject
    - System.Windows.Application
    - System.Windows.FrameworkTemplate
    - System.Windows.Style
    - System.Windows.DependencyObject
      - System.Windows.Media.Visual
        - System.Windows.**UIElement**

- ◊ System.Windows.**FrameworkElement**
- ◊ System.Windows.**ContentElement**
  - ◊ System.Windows.**FrameworkContentElement**

Object es la raíz de la jerarquía de clases. Sus métodos son heredados por todas las clases de la jerarquía y son los siguientes:

- **Equals** para comparar objetos.
- **Finalize** para realizar operaciones de limpieza antes de que un objeto sea reclamado por el recolector de basura.
- **GetHashCode** genera un número que se corresponde con el valor del objeto que admite el uso de una tabla *hash*.
- **ToString** crea una cadena de texto que describe un objeto de la clase.

DispatcherObject. La mayoría de los objetos de WPF se derivan de esta clase que proporciona las estructuras básicas para tratar con la simultaneidad y el subprocesamiento. WPF está basado en un sistema de mensajería similar al conocido sistema de mensajes de Win32, incluso utiliza los mensajes de *User32* para realizar las llamadas entre subprocesos.

Application es una clase que encapsula la funcionalidad específica de las aplicaciones de WPF.

FrameworkTemplate permite la creación de instancias de un árbol de objetos FrameworkElement y/o FrameworkContentElement.

Style es una clase para permitir el uso compartido de propiedades, recursos y controladores de eventos entre los objetos de un tipo. Un estilo normalmente se declara como un recurso en la sección *Resources*.

DependencyObject. Una de las principales filosofías arquitectónicas utilizadas al crear WPF fue la preferencia por las propiedades en lugar de por los métodos o por los eventos. Por eso, para conseguir que más partes del sistema estuviesen controladas por propiedades, fue necesario un sistema de propiedades más completo que el proporcionado por el CLR. Pues bien, esta clase habilita los servicios del sistema de propiedades de WPF en sus numerosas clases derivadas. La función primaria de este sistema de propiedades es calcular los valores de propiedades y notificar de los valores que han cambiado.

Visual es una clase que proporciona soporte para la representación de los objetos WPF visuales.

UIElement define los subsistemas básicos: diseño, entrada y eventos. En cuanto al diseño, todos los controles WPF admiten la capacidad de ajustarse al

tamaño natural de su contenido (recuerde cómo trabaja el panel de diseño **Grid**). También vimos que la salida de WPF es proporcionada por **Visual** y los objetos relacionados. Pues bien, muchas de las entradas de los elementos en general (incluyendo los eventos de teclado, ratón y la entrada del lápiz), así como el comportamiento del foco, también se definen en esta clase. En cuanto a los eventos menmencionados, muchos de ellos son eventos enrutados y de éstos, los relacionados con la entrada de datos, muchos tienen una versión de propagación y otra de tunelización del evento.

FrameworkElement extiende la funcionalidad proporcionada por **UIElement**. Por ejemplo, propiedades como HorizontalAlignment, VerticalAlignment, MinWidth o Margin proporcionan a todos los elementos derivados de esta clase un comportamiento coherente dentro de los paneles de diseño. No obstante, las dos características más importantes introducidas por **FrameworkElement** son el enlace de datos y los estilos (la aplicación de estilos es realmente una forma ligera de enlace de datos).

ContentElement es la clase base para los elementos de contenido. Define las características de contenido para que todas sus clases derivadas proporcionen compatibilidad con la captura de entrada básica del teclado, el ratón, las operaciones de arrastrar y colocar, los controles de lápiz y aceleradores, y puedan recibir potencialmente el foco. Además, incluye eventos relativos a la entrada y al foco, así como eventos para los cambios de estado.

FrameworkContentElement extiende la funcionalidad proporcionada por **ContentElement** para, entre otras cosas, agregar compatibilidad con otras API de entrada y con estilos.

A continuación, se presenta la jerarquía de las clases más utilizadas, derivadas de **FrameworkElement**, así como una breve descripción de las mismas, ya que muchos elementos WPF se derivan de ella directamente o indirectamente a través de clases base intermedias como **Panel** o **Control**:

- **FrameworkElement**. Define diseño, entrada y eventos de los elementos WPF.
  - **Control**. Representa la clase base de los elementos de la interfaz de usuario que utilizan un objeto **ControlTemplate** para definir su aspecto.
    - **PasswordBox**. Control utilizado para escribir una contraseña.
    - **Separator**. Separador de elementos.
    - **TextBoxBase**. Clase para los controles de edición de texto.
      - **TextBox**. Control para mostrar o editar texto sin formato.
      - **RichTextBox**. Control para mostrar o editar texto con formato.

- **ContentControl**. Control que contiene sólo un elemento (propiedad Content). Soporta contenido enriquecido (texto, imágenes, etc.).
  - **UserControl**. Permite crear un control.
  - **Window**. Permite trabajar con ventanas y cuadros de diálogo.
  - **ListBoxItem**. Representa un elemento seleccionable en un ListBox.
    - **ComboBoxItem**. Representa un elemento seleccionable en un ComboBox.
  - **Label**. Etiqueta de texto de un control. Proporciona compatibilidad con las teclas de acceso.
  - **ToolTip**. Descripción abreviada.
  - **StatusBarItem**. Elemento de un control StatusBar.
  - **ButtonBase**. Clase base para todos los botones.
    - **Button**. Botón de pulsación.
    - **ToggleButton**. Clase base para los controles que pueden intercambiar estados.
      - **CheckBox**. Casilla de verificación que muestra gráficamente su estado: seleccionada o deseleccionada.
      - **RadioButton**. Botón de opción que muestra gráficamente su estado: seleccionado o deseleccionado.
  - **ScrollViewer**. Área desplazable que puede contener otros elementos visibles.
- **ItemsControl**. Control para presentar una colección de elementos. Soporta contenido enriquecido (texto, imágenes, etc.).
  - **MenuBase**. Control que define opciones para seleccionar.
    - **Menu**. Menú de una barra de menús.
    - **ContextMenu**. Menú contextual (emergente).
  - **Selector**. Control que permite seleccionar elementos.
    - **ListBox**. Lista fija de elementos seleccionables.
    - **ComboBox**. Lista desplegable de elementos seleccionables.
  - **StatusBar**. Barra de estado.
  - **HeaderedItemsControl**. Control que contiene varios elementos y un encabezado.
    - **MenuItem**. Elemento de un menú (Menu).
    - **ToolBar**. Barra de herramientas.
- **RangeBase**. Controles de rango definido.
  - **ScrollBar**. Barras de desplazamiento.
  - **Slider**. Control deslizante.
  - **ProgressBar**. Barra de progreso.
- **TextBlock**. Es un contenedor para elementos de contenido dinámico.
- **Panel**. Permite colocar y organizar los objetos secundarios en las aplicaciones WPF.

- **Canvas**. Contenedor en el que pueden colocarse los elementos secundarios utilizando coordenadas relativas a su área.
- **StackPanel**. Contenedor que apila los elementos secundarios horizontal o verticalmente.
- **WrapPanel**. Contenedor que coloca los elementos secundarios de izquierda a derecha.
- **DockPanel**. Contenedor en el que los elementos secundarios se alinean con los bordes del mismo.
- **Grid**. Contenedor que define un área de cuadrícula flexible, compuesta por filas y columnas, para situar los elementos.

- **Shape**. Clase base para los elementos de formas.
  - **Line**. Dibuja una línea recta.
  - **Rectangle**. Dibuja un rectángulo.
  - **Ellipse**. Dibuja una elipse.
- **Decorator**. Aplica efectos en o alrededor de un elemento secundario.
  - **Border**. Dibuja un borde, un fondo o ambos alrededor de otro elemento.

ETIQUETAS, CAJAS DE TEXTO Y BOTONES

Los controles más comunes en una aplicación Windows son las etiquetas, las cajas de texto y los botones de pulsación.

Las etiquetas son objetos de la clase **Label**. Se utilizan con frecuencia para informar al usuario sobre la función de otros controles como **TextBox** y para permitir el acceso rápido desde el teclado a los mismos (teclas de acceso: *Alt+tecla*). Para esto último hay que hacer dos cosas: una, indicar cuál será la tecla de acceso (insertar un _ antes de la letra correspondiente en el título), y dos, enlazar su propiedad **Target** con el nombre del elemento **UIElement** que recibe el foco cuando se presiona la tecla de acceso; por ejemplo, con el nombre de un **TextBox**. Una etiqueta no puede contener el foco y no es una posición de tabulación, esto es, su propiedad **TabIndex** no tiene efecto.

```
<Label Content="_Texto:"
       Target="{Binding ElementName=textBox1}" />
```

La propiedad **TabIndex** indica el orden *Tab* (0, 1, 2, etc.) de un control. Este valor determina el orden en el que los controles reciben el foco cuando el usuario navega por ellos utilizando la tecla *Tab*. Un control con un índice de tabulación más bajo recibe el foco antes que un control con un índice más alto. Por omisión, los controles reciben el foco en el orden en el que se añadieron al contenedor.

Cuando un control tiene el foco, esto es, está enfocado, es cuando puede recibir las acciones del teclado.

El siguiente ejemplo muestra una etiqueta con la tecla de acceso *t* para permitir el acceso rápido (*Alt+t*) a la caja de texto *textBox1*. También muestra un botón, *button1*, que puede ser activado haciendo clic sobre él, pulsando las teclas *Alt+b* o pulsando la tecla *Entrar* por ser el botón predeterminado, característica ésta que es añadida al botón por medio de la propiedad **IsDefault**.

```
<StackPanel>
  <WrapPanel>
    <Label Content="_Texto:"
           Target="{Binding ElementName=textBox1}" />
    <TextBox Name="textBox1" TextWrapping="Wrap" TabIndex="0"
             Margin="2" Width="230">
    </TextBox>
  </WrapPanel>
  <Button Content="_Botón" Height="23" Name="button1" Width="75"
          TabIndex="1" IsDefault="True" />
</StackPanel>
```

Las cajas de texto son objetos de la clase **TextBox**. Una caja de texto sólo puede contener texto sin formato. Son particularmente importantes porque permiten realizar la entrada de datos para una aplicación así como visualizar los resultados producidos por la misma.

Los botones de pulsación, controles de la clase **Button**, permiten al usuario ejecutar una acción cuando sea preciso. Para ello, hay que administrar su evento **Click** con el fin de responder cuando el usuario hace clic en el botón.

Tanto los botones como las etiquetas son objetos derivados de **ContentControl**, por lo que pueden tomar cualquier objeto como contenido. Por ejemplo, un botón, además de una cadena de texto, puede tomar como contenido una imagen o, incluso, puede utilizar cualquier otro control, como por ejemplo un **TextBox**.

```
<Button Height="23" Name="button1" Width="75">
  <Image Source="smile.jpg" />
</Button>
```

Esto pone de manifiesto que WPF proporciona muchos tipos de objetos, con modelos de contenido diversos, cuyo propósito primario es mostrar contenido. Por ejemplo, hemos visto que **ContentControl** es un tipo que contiene un solo elemento de contenido, pero **ItemsControl** es un tipo que puede contener varios elementos tales como cadenas, objetos u otros elementos, **HeaderedItemsControl** es un tipo que puede contener varios elementos tales como cadenas, objetos u otros elementos, y también tiene un encabezado, etc.

Vamos a realizar un ejemplo. Piense en una aplicación que permita convertir grados centígrados a Fahrenheit, y viceversa. Esta aplicación requiere una interfaz con al menos dos cajas de texto, de manera que cuando el usuario introduzca en una caja una temperatura en grados centígrados, en la otra se visualice la temperatura equivalente en grados Fahrenheit, y cuando en esta otra caja se introduzca una temperatura en grados Fahrenheit, en la primera se visualice la temperatura correspondiente en grados centígrados.

Recuerde que una interfaz se define como el medio de comunicación entre el usuario y la aplicación.

Desarrollo de la aplicación

Antes de crear una aplicación, debemos responder a algunas preguntas como las siguientes:

- ¿Qué objetos forman la interfaz?
- ¿Qué eventos hacen que la interfaz responda?
- ¿Cuáles son los pasos a seguir para un desarrollo ordenado?

Objetos

La conversión de temperaturas implica los siguientes objetos:

- Un formulario que permita implementar nuestra interfaz.
- Un panel de diseño.
- Una caja de texto para visualizar los grados centígrados.
- Una caja de texto para visualizar los grados Fahrenheit.
- Dos etiquetas que informen al usuario de la información que contiene cada caja de texto.
- Un botón de pulsación ligado a la tecla *Entrar*.

Eventos

En esta aplicación se quiere que cuando el usuario escriba una temperatura en una caja y pulse *Entrar*, el contenido de la otra caja se actualice automáticamente, y viceversa. Por lo tanto, el evento para que se actualicen las cajas de texto es pulsar la tecla *Entrar*, o hacer clic en el botón de pulsación asociado (cuando un botón está asociado con la tecla *Entrar* recibe el nombre de botón predeterminado). Y, ¿a partir de qué caja de texto se hace la conversión? Pues, a partir de aquélla en la que el usuario escribió un nuevo valor, hecho que se puede conocer a través del evento "se ha pulsado una tecla" (¿sobre qué caja se escribió la última vez?).

Cuando una ventana tiene más de un botón, uno y sólo uno de ellos puede ser el botón predeterminado, lo que implica que la tecla *Entrar* realice paralelamente la misma función. Como veremos en el desarrollo de la aplicación, para hacer que un botón sea el botón predeterminado, hay que asignar a su propiedad **IsDefault** el valor **True**. El botón predeterminado se distingue de los demás porque aparece rodeado con un borde más oscuro.

Pasos a seguir durante el desarrollo

1. Crear el esqueleto para una nueva aplicación que utilice un formulario como ventana principal.
2. Añadir los controles necesarios al formulario.
3. Definir las propiedades de los controles.
4. Escribir el código subyacente necesario para cada uno de los objetos.
5. Guardar la aplicación.
6. Crear un fichero ejecutable.

El formulario, los controles y sus propiedades

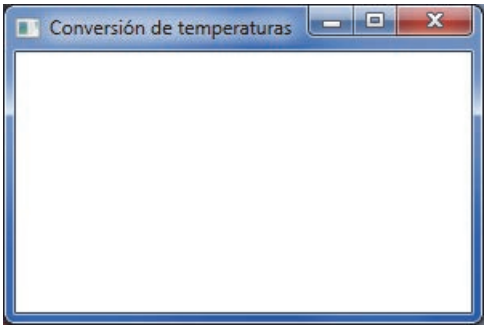
Conocidos los objetos y los eventos, procedemos a dibujar la interfaz. Para ello, creamos el esqueleto para una nueva aplicación, denominada *ConverTemps*, que utilice un formulario como ventana principal.

Los pasos seguidos para escribir el código correspondiente al esqueleto de la aplicación ya fueron explicados en el capítulo *Aplicación WPF*. Por ello, se supone que no presentan ya ninguna dificultad (arranque Visual Studio, diríjase a la barra de menús y ejecute *Archivo* > *Nuevo Proyecto*. En el diálogo que se visualiza, seleccione el tipo de proyecto *Visual Basic* > *Windows* > *Aplicación WPF*, el nombre *ConverTemps* y haga clic en el botón *Aceptar*). Asigne al formulario el nombre *formPrincipal*, el título "Conversión de temperaturas" y fije su tamaño predeterminado (atributos **Height** y **Width**), mínimo (atributos **MinHeight** y **MinWidth**) y máximo (atributos **MaxHeight** y **MaxWidth**).

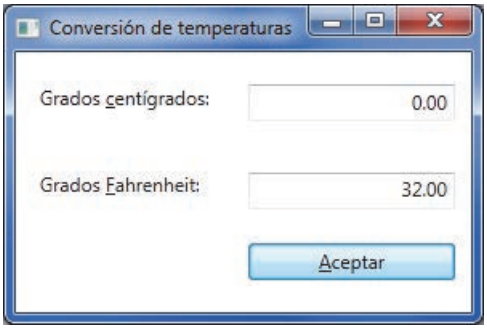
```
<Window x:Class="ConverTemps.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Name="formPrincipal" Title="Conversión de temperaturas"
       Height="200" Width="300"
       MinHeight="200" MinWidth="300"
       MaxHeight="300" MaxWidth="450">
  <Grid>
```

```
  </Grid>
</Window>
```

Cuando se ejecute el código generado, se obtendrá un resultado análogo al de la figura que se muestra a continuación. Los pasos que hay que seguir para ejecutar una aplicación también fueron expuestos en el capítulo *Aplicación WPF* y en los apéndices.



El paso siguiente es colocar sobre el formulario los controles y establecer sus propiedades. Dichos controles los colocaremos sobre un **Grid** de 3 filas por 2 columnas según muestra la figura siguiente.



Según lo explicado en el apartado *Paneles de diseño* del capítulo *Aplicación WPF*, el elemento de tipo **Grid** lo podemos definir así:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
```

```
</Grid>
```

Una vez definida la rejilla, siguiendo el diseño de la figura anterior, colocamos en las celdas de la misma los controles con las propiedades necesarias. La celda (0,0) es la de la esquina superior izquierda y cada elemento secundario puede especificar en qué celda se ubica estableciendo sus propiedades asociadas **Grid.Column** y **Grid.Row** según se explicó en el capítulo *Aplicación WPF*. El valor de estas propiedades por omisión es 0, por lo tanto, si no se establecen, todos los elementos acabarán ubicados en la columna 0, fila 0. Otras propiedades que especificaremos son las siguientes:

Objeto	**Propiedad**	**Valor**
Grid		
Etiqueta	Name Content Margin Target VerticalAlignment	etGradosC Grados _centígrados: 10,15,0,0 ctGradosC Top
Caja de texto	Name Text Margin HorizontalContentAlignment VerticalAlignment	ctGradosC 0,00 20,20,10,0 Right Top
Etiqueta	Name Content Margin Target VerticalAlignment	etGradosF Grados _Fahrenheit: 10,15,0,0 ctGradosF Top
Caja de texto	Name Text Margin HorizontalContentAlignment VerticalAlignment	ctGradosF 32,00 20,20,10,0 Right Top
Botón de pulsación	Name Content Margin	btAceptar _Aceptar 20, 10, 10, 20

Además de estas propiedades, fijaremos la columna y la fila de la rejilla correspondiente a la celda de la misma en la que colocamos el elemento. Por ejemplo, según la figura anterior, el botón estará colocado en la columna 1 fila 2 de la rejilla, lo cual se especifica así:

```
<Button Grid.Column="1" Grid.Row="2"...
```

Una vez finalizado el diseño, el paso siguiente es escribir el código para que cada uno de los elementos del formulario pueda realizar la acción que tenga asignada. Antes, comentamos brevemente las propiedades más comúnmente utilizadas que aún no han sido comentadas.

Tecla de acceso

Se puede observar en el título del botón *Aceptar* que la letra *A* aparece subrayada (si no se ve, pulse la tecla *Alt*). Esto significa que el usuario podrá también ejecutar la acción especificada por el botón pulsando las teclas *Alt+a*. Esta asociación tecla-control recibe el calificativo de nemónico y se realiza escribiendo el símbolo _ (guión bajo o de subrayado) antes de la letra que desea dé acceso al botón. Esto también puede realizarse utilizando explícitamente el elemento **AccessText**:

```
<Button Grid.Column="1" Grid.Row="2" Margin="20,10,10,20"
        Name="btAceptar" >
  <AccessText>_Aceptar</AccessText>
</Button>
```

Botón predeterminado

Si echa un vistazo a los botones de las ventanas de su sistema operativo, observará que, generalmente, hay un botón con un borde más resaltado que los demás; se trata del botón predeterminado: botón que será automáticamente pulsado cuando se pulse la tecla *Entrar*. Para informar al formulario de cuál será el botón predeterminado de entre todos los que contenga, hay que asignar a la propiedad **IsDefault** de ese botón el valor **True**. Sólo puede haber un botón predeterminado.

```
<Button Grid.Column="1" Grid.Row="2" Content="_Aceptar"
        Margin="20,10,10,20" Name="btAceptar" IsDefault="True" />
```

Propiedades comunes

Anteriormente vimos que muchos de los elementos de la interfaz de usuario derivan de la clase **FrameworkElement**, heredando así las propiedades de esta clase, algunas de las cuales exponemos a continuación:

- **Width** y **Height**. Estas propiedades especifican la anchura y altura exactas de su elemento. Utilícelas sólo cuando sea necesario, ya que siempre es mejor permitir que los elementos determinen su propio tamaño.

- **MinWidth**, **MinHeight**, **MaxWidth** y **MaxHeight**. Estas propiedades especifican los límites inferiores y superiores del tamaño de un elemento. Si existe un conflicto entre estos valores, incluidos Width y Height, el orden de aplicación para determinar el ancho real es: primero se deben aplicar las propieda-

des MinWidth y MinHeight, después las propiedades MaxWidth y MaxHeight y, finalmente, si cada una de ellas está dentro de los límites, las propiedades Width y Height.

- **HorizontalAlignment** y **VerticalAlignment**. Estas propiedades controlan la posición de un elemento dentro de su contenedor cuando hay más espacio disponible del necesario. El valor predeterminado para ambas es *Stretch*; esto significa que cuando haya espacio, el elemento se ajustará a ese espacio. Ahora bien, cuando las propiedades Width y Height se establecen explícitamente en un elemento, éstas tienen prioridad y cancelan los efectos de *Stretch*. Los valores *Left*, *Center* y *Right* para la primera propiedad y *Top*, *Center* y *Bottom* para la segunda determinan dónde se situará el elemento en el espacio disponible sin exigirle que se ajuste a este espacio, permitiendo así que utilice su ancho natural.

- **HorizontalContentAlignment** y **VerticalContentAlignment**. Estas propiedades controlan la posición del contenido (elemento secundario) del elemento con el que estamos trabajando (elemento primario). El valor predeterminado para la primera es *Left* (el elemento secundario se alinea a la izquierda) y para la segunda *Top* (el elemento secundario se alinea en la parte superior). Para la primera propiedad, además de *Left*, puede establecer *Right*, *Center* y *Stretch*, y para la segunda, además de *Top*, puede establecer *Bottom*, *Center* y *Stretch*. El valor *Stretch* lo que hace es expandir el elemento secundario hasta que llena el espacio de diseño asignado al elemento principal.

- **Margin**. Esta propiedad especifica la cantidad de espacio que se debe dejar entre este elemento y los elementos adyacentes cuando se cree la interfaz de usuario, por lo tanto, anclan el elemento secundario respecto de los lados especificados (*left*, *top*, *right*, *bottom*) de su contenedor, lo que permitirá su redimensionamiento automático. El valor para esta propiedad puede estar formado por un único valor: mismo espacio por todas las partes, por dos valores: el primer número indica el espacio a dejar por la izquierda y por la derecha y el segundo, el espacio a dejar por arriba y por abajo, o por cuatro: indicando el espacio que se debe dejar por la izquierda, por arriba, por la derecha y por abajo, respectivamente.

  Por ejemplo, observe el elemento **TextBox** definido por el código mostrado a continuación; está ubicado en la fila 0 (valor por omisión), columna 1 de la rejilla. La propiedad **Margin** especifica un valor 0 para *bottom*, lo que haría que la caja de texto aumentara su tamaño vertical hasta que su lado inferior se alineara con el fondo de la celda (0,1), ya que el valor predeterminado de **VerticalAlignment** es *Stretch*, pero este efecto queda anulado al asignar a la propiedad **VerticalAlignment** el valor *Top* que indica que el elemento se ali-

neará con la parte superior de la celda, sin exigirle que se ajuste al espacio disponible, respetando el margen 20 especificado.

```
<TextBox Grid.Column="1" Margin="20,20,10,0" Name="ctGradosC"
         Text="0,00" HorizontalContentAlignment="Right"
         VerticalAlignment="Top" />
```

- **Padding**. Esta propiedad, definida en las clases Control, TextBlock y Border, es aplicable a controles con contenido y especifica el margen (en todo el contorno) que se debe dejar entre el contenido y el borde del elemento.

- **Visibility**. Esta propiedad determina si un elemento es visible. Los valores que puede tomar son: *Visible*: se muestra el elemento, *Hidden*: no se muestra el elemento, pero se reserva espacio para él en la disposición, y *Collapsed*: no se muestra el elemento y no se reserva espacio para él en la disposición.

EVENTOS ENRUTADOS

WPF utiliza eventos enrutados para superar las limitaciones que tienen los eventos normales (eventos del CLR). Por ejemplo, un evento del CLR puede invocar controladores simplemente en el objeto que lo desencadenó, pero en WPF existen diversos controles que tienen un modelo de contenido enriquecido; por ejemplo, se puede colocar una imagen dentro de un control **Button**; evidentemente, la imagen agregada no debe interrumpir que el botón responda al evento **Click**, aun cuando el usuario haga clic en el elemento imagen del botón. Los eventos enrutados, al propagarse entre el elemento que los desencadena y la raíz del árbol de elementos, permiten asociar ese controlador a cualquier elemento en esa ruta. También, en *Windows Forms* era necesario asociar varias veces el mismo controlador para procesar un evento que se desencadenaba desde varios elementos (por ejemplo, el evento **Click** de una orden de un menú y de su botón asociado de la barra de herramientas). Los eventos enrutados, al propagarse hasta la raíz del árbol de elementos, permiten asociar ese controlador una sola vez; por ejemplo, en el elemento padre (contenedor) de esos otros elementos.

Un evento enrutado es un evento del CLR que está respaldado por una instancia de la clase **RoutedEvent** y registrado en el *sistema de eventos de WPF*. Además, al propagarse entre el elemento que lo desencadena y la raíz del árbol de elementos, puede ser respondido por cualquier elemento que esté en esa ruta, en lugar de simplemente por el elemento que lo desencadenó.

En el escenario de nuestra aplicación, por ejemplo, un clic en el botón supone que este elemento sea el primero que tiene la oportunidad de controlar el evento. Pero si ninguno de los controladores asociados al botón actúa sobre el evento, éste

se propagará en el árbol de elementos hasta el elemento primario, la raíz del árbol, buscando un controlador que lo maneje (*Button* > *Grid* > *Window*).

```
<Window x:Class="ConverTemps.MainWindow" ...>

  <Grid>
    ...

    <Button Grid.Column="1" Grid.Row="2" ... />
  </Grid>
</Window>
```

Pero, no todos los eventos enrutados siguen la misma ruta, sino que hay varios tipos de enrutamiento: de propagación (P), directo (D) y de túnel (T). Se dice que el evento enrutado "se propaga" si va subiendo por el árbol desde el elemento que lo desencadenó hasta la raíz y "se tuneliza", si se inicia en la raíz y viaja a un destino situado más abajo en el árbol, teniendo su meta en el elemento que lo desencadenó, pudiendo ser respondido por cualquier elemento en la ruta que tenga un controlador para dicho evento. En cambio, un evento con enrutamiento directo sólo puede ser respondido por el elemento que lo desencadenó.

Los eventos de entrada proporcionados en WPF suelen venir implementados como una pareja de túnel-propagación; por ejemplo **PreviewKeyDown** y **KeyDown** tienen la misma firma, pero el primero es el evento de túnel y el segundo es el de propagación. Los eventos de túnel también se conocen como *eventos de vista previa* (porque su nombre empieza con *Preview*). No obstante, hay también eventos de entrada que sólo utilizan el enrutamiento de propagación o el directo.

Los eventos de entrada de WPF que se presentan por parejas se implementan de forma que una única acción del usuario desde la entrada, como pulsar una tecla en una caja de texto, desencadene los dos eventos enrutados de la pareja, uno detrás de otro. Primero se desencadena el evento de túnel que viaja por su ruta y a continuación, el evento de propagación que también viaja por su ruta. Entonces, la primera oportunidad para filtrar o realizar acciones en respuesta a dicho evento la tienen los elementos de la ruta con controladores para el evento de túnel y, justo a continuación, la segunda oportunidad la tienen los elementos de la ruta con controladores para el evento de propagación, excepto si un elemento a lo largo de la ruta de túnel marca el evento enrutado como controlado/manejado (**Handled**).

Se dice que un evento está controlado o administrado cuando se da por finalizada la ejecución de las acciones programadas para responder al mismo. Para ello, todos los eventos enrutados comparten una misma clase base para los datos relacionados con los mismos, **RoutedEventArgs**, la cual define la propiedad **Handled** de tipo **Boolean**, cuyo propósito es permitir que cualquier controlador de

eventos a lo largo de su ruta marque el evento enrutado como controlado, estableciendo el valor de dicha propiedad a **True**. Por ejemplo:

```
Private Sub CajaTexto_PreviewTextInput(sender As System.Object,
e As System.Windows.Input.TextCompositionEventArgs)
  ' ...
  e.Handled = True
End Sub
```

No hay ninguna regla absoluta que determine cuándo se deben marcar los eventos enrutados como controlados. Generalmente se debería hacer si, existiendo más de una respuesta, el código respondió al evento enrutado de una manera significativa y relativamente completa. Lo normal es que no sea necesario que haya más de una respuesta significativa que necesite implementaciones del controlador independientes. No obstante, para la mayoría de los escenarios, al marcar un evento como controlado se detendrá el enrutamiento para una ruta de tunelización o una ruta de propagación, pero hay mecanismos mediante los cuales los agentes de escucha a lo largo de la ruta pueden seguir ejecutando controladores en respuesta a esos eventos enrutados, independientemente de que **Handled** valga **True**; por ejemplo, **UIElement.AddHandler**: si el tercer parámetro de este método es **True**, el controlador especificado como segundo parámetro será invocado para el evento enrutado especificado por el primer parámetro aunque ya haya sido marcado como controlado por otro elemento a lo largo de la ruta del evento.

```
Public Sub AddHandler (
    evento As RoutedEvent,
    controlador As Delegate,
    controlar As Boolean
)
```

¿Cómo se definen?

El modelo de eventos de WPF es muy similar al modelo de propiedades de WPF. Al igual que con las propiedades de dependencia (véase el capítulo *Aplicación WPF*), los eventos enrutados están representados por variables estáticas de sólo lectura y un envoltorio para proveer acceso tipo CLR para el evento. El siguiente código muestra cómo se define y se registra un evento enrutado:

```
' Definición y registro del evento
Public Shared Readonly NombreEventoEvent As RoutedEvent =
    EventManager.RegisterRoutedEvent(
    "NombreEvento", RoutingStrategy.Bubble,
    GetType(RoutedEventHandler), GetType(ClaseQueDefineElEvento)
  ...
    )
```

```
' Envoltorio para proveer acceso CLR para el evento
Public Custom Event NombreEvento As RoutedEventHandler
  AddHandler(ByVal value As RoutedEventHandler)
    Me.AddHandler(NombreEventoEvent, value)
  End AddHandler

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

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

Vemos que WPF proporciona envoltorios tanto para los eventos enrutados como para las propiedades de dependencia. De esta forma podemos gestionar los eventos enrutados y las propiedades de dependencia como si se tratase de eventos y propiedades del CLR, al mismo tiempo que nos beneficiamos de la funcionalidad adicional ofrecida por ambos. Por ejemplo, el envoltorio del ejemplo anterior permite adjuntar un controlador de eventos utilizando la siguiente sintaxis:

```
elementoWPF.AddHandler(TipoelementoWPF.NombreEventoEvent,
                New RoutedEventHandler(AddressOf ControladorEvento))
```

Responder a los eventos

Cuando en la aplicación ocurre algo de interés, los eventos habilitan un objeto para notificarlo a otros objetos. El objeto que genera el evento recibe el nombre de *editor* y los objetos que reciben o controlan el evento se denominan *suscriptores*.

Continuando con la aplicación, como ya tenemos el formulario con sus controles, el paso siguiente es hacer que estos controles respondan a las acciones que el usuario realice sobre la ventana de la aplicación, lo que requiere añadir a la misma los controladores o manejadores de eventos que vinculen cada control con el método que debe ejecutarse como respuesta al evento generado desde ese control (se dice que la ventana se suscribe al evento del control). Lógicamente, estos métodos conducidos por eventos serán miembros de la clase *MainWindow* derivada de **System.Windows.Window**. Por ejemplo:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  ' Código que responderá al evento
End Sub
```

En el capítulo *Aplicación WPF* explicamos cómo asignar controladores de eventos a un objeto. Allí explicamos que un controlador de eventos enrutados, que

recibe también el nombre de delegado, es un objeto de la clase **RoutedEventHandler** del espacio de nombres **System.Windows** que guarda una referencia al método que responderá a ese evento. Por ejemplo, la siguiente línea de código Visual Basic agrega al botón *btAceptar* un controlador de eventos para controlar/manejar el evento **Click**. En este caso, la respuesta a este evento será la ejecución del método *btAceptar_Click* expuesto anteriormente:

```
btAceptar.AddHandler(Button.ClickEvent,
            New RoutedEventHandler(AddressOf btAceptar_Click))
```

AddHandler añade un controlador de eventos enrutados, para el evento enrutado especificado, a la colección de controladores del elemento actual. Alternativamente, la línea de código anterior puede escribirse también así:

```
AddHandler btAceptar.Click, AddressOf btAceptar_Click
```

El código Visual Basic anterior correspondiente al controlador de eventos es generado por el procesador de XAML cuando añadimos al código XAML del botón *btAceptar* el código sombreado que se muestra a continuación:

```
<Button Grid.Column="1" Grid.Row="2" Content="_Aceptar"
        Margin="20,10,10,20" Name="btAceptar" IsDefault="True"
        Click="btAceptar_Click" />
```

Si añade este código XAML manualmente y, a continuación, hace doble clic sobre el botón desde la ventana de diseño del EDI o a través del panel de eventos de la ventana de propiedades, se añadirá automáticamente el método *btAceptar_Click* expuesto anteriormente. En otro caso, al hacer doble clic, se añadirá el mismo método pero seguido de la cláusula **Handles** que especifica el evento que será controlado por el método (esta cláusula sustituye al código XAML anterior):

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs) Handles btAceptar.Click

End Sub
```

Siguiendo con el ejemplo, el proceso que deseamos realizar es que cuando un usuario escriba una temperatura en una caja de texto y pulse la tecla *Entrar*, se actualice automáticamente el contenido de la otra caja con el valor resultante de la conversión correspondiente. Para realizar la conversión de una temperatura en grados centígrados a la misma en grados Fahrenheit y viceversa, utilizaremos las fórmulas siguientes:

```
GradosFahr = (GradosCent x 9 / 5) + 32
GradosCent = (GradosFahr - 32) x 5 / 9
```

Para aplicar una u otra fórmula, necesitamos saber en qué caja de texto se ha escrito el valor a convertir, información que obtendremos a través de alguno de los eventos que se generan cuando se escribe en una caja de texto. Para saber en qué caja de texto se ha escrito, definiremos en la clase *MainWindow* la variable miembro *objTextBox* de tipo **TextBox**. Para ello, diríjase a la ventana de código (fichero *MainWindow.xaml.vb*) y escriba:

```
Class MainWindow
  Private objTextBox As TextBox = Nothing

  ' ...
End Class
```

Antes de continuar con el desarrollo de la aplicación, vamos a estudiar los eventos relacionados con el teclado.

Eventos relacionados con el teclado

Sabemos que todos los elementos de la interfaz gráfica se derivan de la clase **UIElement**, la cual define los eventos del teclado, además de otros eventos.

Cuando el usuario escribe en una caja de texto, cada pulsación produce todos o algunos de los eventos siguientes en el orden especificado:

- **PreviewKeyDown** (T) y **KeyDown** (P). Tienen lugar cuando se presiona una tecla en el elemento que tiene el foco.

- **PreviewTextInput** (T) y **TextInput** (P). Se producen cuando en un elemento se introduce texto, independientemente del dispositivo (el teclado es el medio predeterminado, pero la voz, escritura a mano y otros dispositivos de entrada también pueden generar este evento). Ojo, en algunos controles, como en un **TextBox**, el evento **TextInput** puede estar marcado como administrado (**Handled** igual **True**) porque esos controles necesitan interpretar algunos tipos de entradas, como las teclas de dirección, que tienen un significado especial para los mismos. En estos casos, utilizaremos el evento **PreviewTextInput**.

- **PreviewKeyUp** (T) y **KeyUp** (P). Tienen lugar cuando se suelta una tecla en el elemento que tiene el foco.

Para interceptar este tipo de eventos, por ejemplo **PreviewTextInput**, vamos a añadir a la clase *MainWindow*, para cada una de las cajas de texto, un controlador de eventos según se indica a continuación:

```
Private Sub nombre_PreviewTextInput(Object sender,
TextCompositionEventArgs e)

End Sub
```

Este método recibe un primer argumento, *sender*, de tipo **Object** que hace referencia al objeto que en un instante determinado está controlando el evento (ejecutando el controlador de eventos) y el segundo, *e*, de tipo **TextCompositionEventArgs**, clase derivada de **RoutedEventArgs**, proporciona las siguientes propiedades relacionadas con los datos para el evento:

- **Handled**. Esta propiedad de tipo **Boolean** permite obtener o establecer si se controló (**True**) o no (**False**) el evento. Si el evento no está controlado, *Handled=False*, viajará por su ruta y se seguirá procesando, y si *Handled=True*, el evento se detiene considerándolo controlado. *Handled=True* en un controlador **Preview...** no solamente detendrá el evento de túnel, sino que también detendrá el evento de propagación.

- **Source**. El valor de esta propiedad hace referencia al elemento que desencadenó el evento y coincidirá con el valor del parámetro *sender* cuando el elemento que controle el evento sea el que lo desencadenó. En otro caso, el evento viaja por su ruta, *Source* hace referencia al elemento que desencadenó el evento y *sender*, al elemento que lo está controlando actualmente.

- **Text**. Esta propiedad de tipo **String** permite obtener el texto de entrada asociado a un evento del tipo que estamos describiendo: **TextCompositionEventArgs**.

- **OriginalSource**. Hay controles que, en función de su modelo de contenido, intentan ajustar el origen del evento con el fin de proporcionar el más útil. Esto es, el origen donde se desencadenó el evento puede ser ajustado para un control dentro de su modelo de contenido; por ejemplo, el contenido de una lista notificará como *Source* a la lista (ListBox) y como *OriginalSource* al elemento real (TextBlock, por ejemplo) dentro del elemento de lista (ListBoxItem), posibilitando así poder elegir la fuente original donde se desencadenó el evento si es que tiene mayor relevancia para el control del mismo.

Para añadir el controlador mencionado, sitúese en la ventana de diseño, seleccione una caja de texto, diríjase a la ventana de propiedades, haga clic en su botón *Eventos* para mostrar la lista de eventos y haga doble clic sobre el evento **PreviewTextInput**. Repita este proceso para la otra caja de texto. ¿Cuál será la respuesta a este evento? En ambos casos la aplicación responderá de la misma forma: almacenando la referencia a la caja de texto en la variable *objTextBox*, que poste-

riormente interrogaremos para saber sobre qué caja de texto escribió el usuario, y actualizar a partir de este valor el contenido de la otra.

```
Private Sub ctGradosC_PreviewTextInput(sender As System.Object,
e As System.Windows.Input.TextCompositionEventArgs)
  objTextBox = TryCast(sender, TextBox)
End Sub

Private Sub ctGradosF_PreviewTextInput(sender As System.Object,
e As System.Windows.Input.TextCompositionEventArgs)
  objTextBox = TryCast(sender, TextBox)
End Sub
```

TryCast realiza la conversión explícita de tipos requerida y devuelve **Nothing** si la conversión no es viable.

Se habrá dado cuenta de que podríamos haber utilizado un método único para ambas cajas de texto. En este caso porque, para ambas, la respuesta es la misma y en otros casos porque a través de los parámetros *sender* y *e* podemos identificar la caja para la que fue invocado el método. Incluso, apoyándonos en la estrategia de enrutamiento de cada evento, podríamos haber asociado el controlador para el evento de tunelización **PreviewTextInput** una sola vez, en el elemento padre de las cajas de texto: la rejilla. Otra opción podría ser controlar el evento de propagación **TextChanged**.

Una vez que el usuario haya escrito un valor en una caja de texto, su siguiente acción será pulsar la tecla *Entrar* o, lo que es lo mismo, hacer clic en el botón *Aceptar*. Entonces, para que el botón *Aceptar* pueda responder al evento **Click**, le asociaremos el controlador de eventos correspondiente. Para ello, procediendo de forma análoga a como lo ha hecho anteriormente, añada ese controlador de eventos para que haga referencia al siguiente método de la clase *MainWindow*:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)

End Sub
```

Este método tiene que interrogar la variable miembro *objTextBox* para saber en qué caja de texto introdujo el usuario un nuevo valor y aplicar así la fórmula de conversión adecuada, visualizando el resultado de la conversión en la otra caja.

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Try
    Dim grados As Double
    ' Si se escribió en la caja de texto grados centígrados...
```

```
    If objTextBox Is ctGradosC Then
      grados = Convert.ToDouble(ctGradosC.Text) * 9.0 / 5.0 + 32.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosF.Text = String.Format("{0:F2}", grados)
    End If
    ' Si se escribió en la caja de texto grados Fahrenheit...
    If objTextBox Is ctGradosF Then
      grados = (Convert.ToDouble(ctGradosF.Text) - 32.0) * 5.0 / 9.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosC.Text = String.Format("{0:F2}", grados)
    End If
  Catch exc As FormatException
    ctGradosC.Text = "0,00"
    ctGradosF.Text = "32,00"
  End Try
End Sub
```

En el código anterior se puede observar que la propiedad **Text** hace referencia a un objeto **String** que almacena el contenido de la caja de texto, que es convertido a un valor **Double** utilizando el método **Convert.ToDouble**. Finalmente, el resultado de tipo **Double** es convertido a un **String** invocando al método **Format** de esta clase y visualizado en la caja de texto correspondiente a través de su propiedad **Text**. En el caso de que el dato introducido no se corresponda con un valor numérico, será lanzada una excepción de tipo **FormatException** que atraparemos para que las cajas vuelvan a mostrar sus valores iniciales.

El código completo de esta aplicación puede obtenerlo del CD en la carpeta *Cap02\ConverTemps*. Compile ahora la aplicación, ejecútela y observe cómo trabaja.

Eventos relacionados con el foco

Cuando un control posee el punto de inserción se dice que dicho control está enfocado o que tiene el foco.

Un usuario de una aplicación puede enfocar un determinado control haciendo clic sobre él o bien pulsando la tecla *Tab* una o más veces hasta situar el foco sobre él. Así mismo, un control también puede ser enfocado desde la propia aplicación; por ejemplo, si usted quiere que el control *ctGradosC* quede enfocado cuando la ventana se visualice, puede hacerlo de dos maneras:

1. Invocando al método **Focus** para el control que requiere el foco una vez abierto el formulario. Este proceso puede realizarse como respuesta al evento **Loaded** que se genera cuando se ha cargado el formulario por primera vez.

Según lo expuesto, para enfocar el control *ctGradosC* de este formulario, añada el siguiente controlador a la clase *MainWindow*:

```
Private Sub formPrincipal_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctGradosC.Focus()
End Sub
```

El método *formPrincipal_Loaded* se ejecutará como respuesta al mensaje "formulario cargado" que se produce una vez que la ventana derivada de la clase **Window** se crea para ser visualizada.

2. Según hemos indicado, el usuario puede pasar de un control a otro (entre los controles que admiten el foco) pulsando la tecla *Tab*. El orden que se sigue coincide con el orden en el que fueron colocados los controles en la ventana. Por lo tanto, otra forma de que un determinado control reciba inicialmente el foco es colocándolo el primero o asignando a su propiedad **TabIndex** un valor 0 a través de la ventana de propiedades. Esto lo podemos hacer extensivo a todos los controles de un formulario asignándoles un valor 0, 1, 2, 3, etc., que coincida con el orden en el que deseamos sean enfocados cuando el usuario se desplace por los mismos utilizando la tecla *Tab*.

WPF distingue entre el foco de teclado y el foco lógico. El foco de teclado se refiere al elemento que recibe las acciones del teclado y el foco lógico se refiere al elemento que tiene el foco dentro de un ámbito de foco. Solamente un elemento puede tener el foco del teclado en un instante determinado. Ahora bien, piense que el usuario puede cambiarse a otra aplicación, la cual recibirá el foco del teclado que la anterior ha perdido. Cuando esto ocurre, el último elemento que tuviera el foco del teclado en esa anterior aplicación retiene el foco lógico para que cuando ésta vuelva a recibir el foco del teclado, éste regrese al elemento que tenía el foco lógico. WPF proporciona la clase **Keyboard** para administrar el foco del teclado y **FocusManager** para el foco lógico.

Para que un elemento obtenga el foco del teclado, sus propiedades **Focusable** e **IsVisible** deben estar establecidas en **True**.

Los eventos relacionados con el foco son los siguientes:

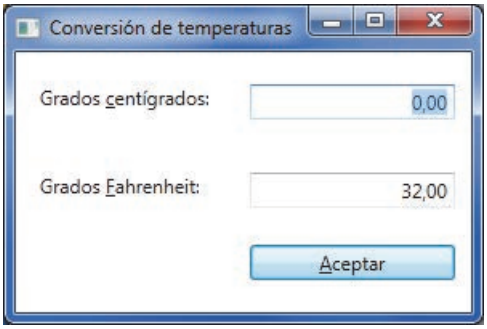
- **PreviewGotKeyboardFocus** (T) y **GotKeyboardFocus** (P). Tienen lugar cuando el elemento recibe el foco del teclado.
- **PreviewLostKeyboardFocus** (T) y **LostKeyboardFocus** (P). Tienen lugar cuando el elemento pierde el foco del teclado.
- **GotFocus** (P). Tiene lugar cuando el elemento recibe el foco lógico.

- **LostFocus** (P). Tiene lugar cuando el elemento pierde el foco lógico.

Como ejemplo, vamos a añadir a nuestra aplicación el código necesario para que cuando una caja de texto obtenga el foco todo su contenido quede seleccionado. Esto es fácil si sabemos que cuando un control obtiene el foco, genera el mensaje "foco obtenido" y cuando lo pierde, "foco perdido". Estos eventos se producen al cambiar el foco, por ejemplo, mediante el teclado (tecla *Tab*).

Seleccionar el texto de una caja de texto

Muchas veces requeriremos de una caja de texto que su contenido sea seleccionado automáticamente cuando reciba el foco. Esto permitirá escribir un nuevo contenido sin tener que preocuparse de borrar el existente.



Según lo expuesto, vamos a asociar con cada una de las cajas de texto un controlador de eventos **GotKeyboardFocus**. Para interceptar este tipo de eventos, añadiremos a la clase *MainWindow* un controlador como el siguiente:

```
Private Sub ctGrados_GotKeyboardFocus(sender As System.Object,
e As System.Windows.Input.KeyboardFocusChangedEventArgs)
  Dim ObjTextBox As TextBox = TryCast(sender, TextBox)
  ObjTextBox.SelectAll()
End Sub
```

Lo que hace este método es obtener el identificador de la caja de texto que obtiene el foco y enviarle el mensaje **SelectAll**. El método **SelectAll** selecciona todo el texto de la caja de texto.

Otras propiedades/métodos relacionados con la selección de texto en un control de texto son las siguientes:

- **SelectionStart**. Propiedad que permite obtener o establecer el punto de inicio del texto seleccionado en la caja de texto. Por ejemplo:

```
TextBox1.SelectionStart = 10
```

fija el punto de inserción en la posición 10 de la caja *TextBox1* y

```
pos = TextBox1.SelectionStart
```

devuelve la posición del punto de inicio del texto seleccionado.

- **SelectionLength**. Propiedad que permite obtener o establecer el número de caracteres seleccionados en la caja de texto. Por ejemplo:

  ```
  TextBox1.SelectionLength = 5
  ```

  selecciona cinco caracteres a partir del punto de inserción en la caja *TextBox1* y

  ```
  n = TextBox1.SelectionLength
  ```

  devuelve el número de caracteres seleccionados.

- **SelectedText**. Propiedad que permite obtener el texto seleccionado, o bien reemplazar el texto seleccionado (puede ser nulo) por otro. Por ejemplo, el código siguiente muestra una forma rápida de añadir texto a una caja de texto sin necesidad de reescribir el contenido de la caja:

  ```
  TextBox1.SelectionStart = TextBox1.Text.Length 'posición final
  TextBox1.SelectedText = NuevoTexto 'añadir
  ```

- **Select(int** *pos_inicial*, **int** *ncars***)**. Selecciona *ncars* del texto a partir de la posición especificada.

Eventos relacionados con el ratón

Sabemos que todos los elementos de la interfaz gráfica se derivan de la clase **UIElement**, la cual define los eventos del ratón, además de otros eventos.

La entrada de datos del ratón es recibida por cualquier elemento que esté directamente debajo del ratón. Algunos de estos eventos son los siguientes:

- **PreviewMouseDown** (T) y **MouseDown** (P). Tienen lugar cuando se hace clic sobre el elemento con un botón del ratón.

- **PreviewMouseMove** (T) y **MouseMove** (P). Tienen lugar cuando se mueve el puntero del ratón sobre el elemento.

- **PreviewMouseUp** (T) y **MouseUp** (P). Tienen lugar cuando se suelta un botón del ratón sobre el elemento en el que se está actuando.

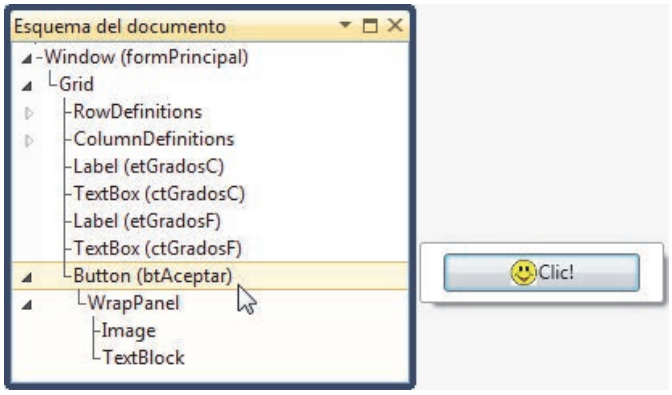
- **MouseEnter** (D). Tiene lugar cuando el puntero del ratón entra en los límites del elemento.

- **MouseLeave** (D). Tiene lugar cuando el puntero del ratón sale de los límites del elemento.

Además de estos eventos (hay otros), **UIElement** define una propiedad **IsMouseOver** que devuelve **True** si el puntero del ratón está sobre el elemento en cuestión o sobre cualquiera de sus hijos e **IsMouseDirectlyOver** que devuelve **True** sólo si el puntero del ratón está sobre el elemento en cuestión.

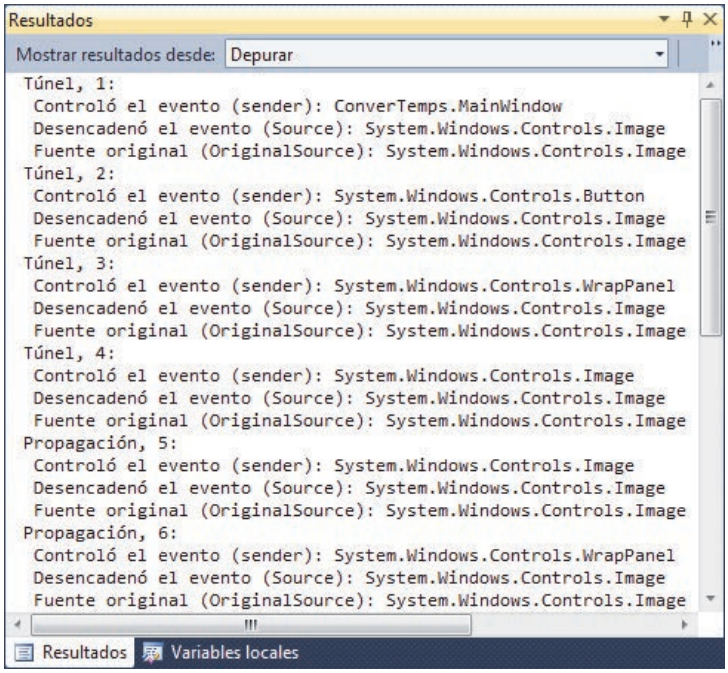
También, la biblioteca .NET incluye la clase **Mouse** que proporciona eventos, propiedades y métodos relacionados con el ratón; por ejemplo, el método **GetPosition** para obtener la posición del ratón.

Como ejemplo, vamos a realizar una pequeña modificación en la aplicación anterior con la intención de mostrar en la ventana de *Resultados* de Visual Studio la tunelización del evento **PreviewMouseDown** y la propagación del evento **MouseDown**. Para ello, vamos a modificar el botón *btAceptar* para que su contenido sea un elemento **WrapPanel** que incluya un control **Image** y un control **TextBlock** (asegúrese de que el fichero que contiene la imagen forma parte del proyecto y de que su propiedad *Acción de compilación* vale *Resource*). Después escribimos los controladores para los eventos citados de los controles **Window**, **Button**, **WrapPanel**, **Image** y **TextBlock** para mostrar los datos de quién controla el evento y quién lo desencadena.



Ahora, cuando hagamos clic en cualquiera de los elementos del botón (**Image**, **TextBlock**, **WrapPanel** o **Button**) se desencadenará la pareja de eventos **PreviewMouseDown** y **MouseDown** y se mostrará en la ventana de *Resultados* de Visual Studio (tendremos que ejecutar la aplicación en modo depuración, *F5*) los resultados producidos por los controladores que hay en la ruta para responder a dichos eventos.

La figura siguiente muestra el resultado que presenta la ventana de *Resultados* inmediatamente después de hacer clic en la imagen del botón. Se puede observar que el evento **PreviewMouseDown** desciende desde la raíz, la ventana, hasta la imagen (elemento donde se desencadenó el evento) y que el evento **MouseDown** asciende desde la imagen, hacia la raíz, pudiendo ser ambos respondidos por cualquier elemento en la ruta que tenga un controlador para dicho evento. También se puede ver que primero se procesa el evento de túnel y después el de propagación.



Para crear los elementos descritos y controlar los eventos mencionados, es necesario añadir el código indicado a continuación:

```
<Window x:Class="ConverTemps.MainWindow"
        ...
        PreviewMouseDown="ResponderEventoTunel"
        MouseDown="ResponderEventoPropa">
  <Grid>
    ...
    <Button Name="btAceptar"...
            PreviewMouseDown="ResponderEventoTunel"
            MouseDown="ResponderEventoPropa" >
      <WrapPanel  PreviewMouseDown="ResponderEventoTunel"
                  MouseDown="ResponderEventoPropa">
        <Image Source="Imagenes/smile.jpg" Width="16"
               Height="16" Margin="1"
```

```
                PreviewMouseDown="ResponderEventoTunel"
                MouseDown="ResponderEventoPropa" />
          <TextBlock Grid.Column="1"
                     PreviewMouseDown="ResponderEventoTunel"
                     MouseDown="ResponderEventoPropa" >Clic!
          </TextBlock>
        </WrapPanel>
    </Button>
  </Grid>
</Window>
```

Y para responder a los eventos del ratón mencionados anteriormente, es necesario escribir los controladores siguientes:

```
Private Sub ResponderEventoTunel(sender As Object, e As MouseButtonEventArgs)
  Dim ruta As String = "Túnel, "
  Resultados(sender, e, ruta)
End Sub

Private Sub ResponderEventoPropa(sender As Object, e As MouseButtonEventArgs)
  Dim ruta As String = "Propagación, "
  Resultados(sender, e, ruta)
End Sub

Private Sub Resultados(sender As Object, e As MouseButtonEventArgs, ruta As
String)
  ContarEventos += 1
  System.Diagnostics.Debug.WriteLine(
    ruta & ContarEventos.ToString() & ":" & vbLf &
    " Controló el evento (sender): " & sender.ToString() & vbLf &
    " Desencadenó el evento (Source): " & Convert.ToString(e.Source) & vbLf &
    " Fuente original (OriginalSource): " & Convert.ToString(e.OriginalSource))
    'e.Handled = true;
End Sub
```

Hay algunos detalles más en este ejemplo. Los dos eventos, el de túnel y el de propagación, comparten literalmente el mismo objeto de datos del evento. El evento se puede desencadenar en un control y controlar en otro. Si añadimos al final del método *Resultados* la sentencia que aparece comentada, sólo el objeto **Window** responderá al evento de túnel, como habíamos explicado anteriormente.

También podemos observar que el evento de propagación **MouseDown** no se controla en el botón (lo mismo ocurriría con **MouseUp**); esto es, si hacemos clic en el elemento **Image**, el evento de propagación **MouseDown** finaliza en el panel (no se procesa en el botón) y si hacemos clic en el propio elemento **Button**, el evento de túnel **PreviewMouseDown** finaliza en el botón. Esto pone de manifiesto que se pueden programar situaciones en las que se desee que el evento no se propague. ¿Cómo? Asignando a la propiedad **Handled** del objeto de datos del

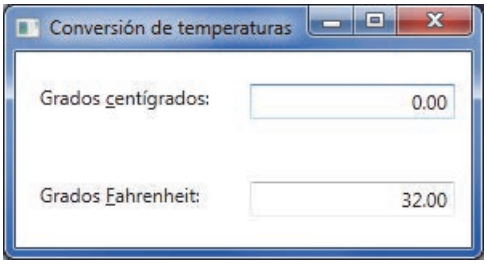
evento el valor **True**, lo que marca el evento como controlado deteniéndose el enrutamiento para la ruta de tunelización o la ruta de propagación. ¿Por qué hacer esto? Pues porque a lo mejor se quiere convertir el evento en algo distinto. Por ejemplo, el elemento **Button** está programado para que un evento **MouseDown** seguido de otro **MouseUp** se conviertan en un solo evento **Click**; esto es, se suprimen los eventos del ratón más primitivos para dejar que sólo se propague el evento **Click** del control, razón ésta que justifica el comportamiento anterior.

INTERCEPTAR LA TECLA PULSADA

La aplicación *ConverTemps*, expuesta anteriormente, requiere el botón por omisión *Aceptar* para que cuando el usuario modifique el contenido de una caja de texto y pulse la tecla *Entrar*, se realice la modificación correspondiente en la otra caja. Seguramente usted habrá pensado en eliminar el botón *Aceptar* y en su lugar interceptar la tecla *Entrar*, instante en el que hay que realizar la conversión.

Una forma de interceptar la tecla *Entrar* es agregando el controlador del evento **PreviewKeyDown** o **PreviewTextInput** del componente de texto que recibe tal evento y verificar si se pulsó la tecla *Entrar*.

Según lo expuesto, vuelva a reproducir la aplicación *ConverTemps* anterior, pero ahora sin el botón de pulsación, sin la variable *objTextBox* y sin los controladores del evento **PreviewTextInput**. Denominaremos a esta nueva versión de la aplicación *ConverTemps2*.



Para interceptar la pulsación de la tecla *Entrar* que el usuario realizará después de escribir la cantidad que desea convertir, añadiremos para ambas cajas de texto el controlador del evento **PreviewKeyDown** siguiente:

```
Private Sub Grid_PreviewKeyDown(sender As Object, e As KeyEventArgs)
  If e.Key = Key.Enter Then
    ' Se pulsó la tecla Entrar
    e.Handled = True
    Conversion(e.Source)
  End If
End Sub
```

Que un método, como el anterior, sea único para ambas cajas de texto podemos hacerlo de dos formas: asignando a cada caja de texto un controlador para el evento **PreviewKeyDown** que haga referencia a ese método o, apoyándonos en la estrategia de enrutamiento de cada evento, asignando al elemento padre de las cajas de texto, la rejilla, el controlador para el evento **PreviewKeyDown** que haga referencia a ese método. Esto es lo que hemos hecho en este ejemplo, con el fin de utilizar un único controlador:

```
<Grid PreviewKeyDown="Grid_PreviewKeyDown">
```

Como se puede observar, el método *Grid_PreviewKeyDown* utiliza la propiedad **Key** del objeto *e* de tipo **KeyEventArgs** para comprobar si se presionó la tecla *Entrar* (la enumeración **Key** proporciona los valores de las teclas posibles en un teclado). Si se presionó la tecla *Entrar*, la propiedad **Handled** se establece en **True**, lo que indica que el evento está controlado, impidiendo que siga viajando por su ruta y, a continuación, se invoca al método *Conversion* pasando como argumento la referencia *e.Source* al objeto (caja de texto) que produjo el evento para que convierta la cantidad tecleada en los grados correspondientes:

```
Private Sub Conversion(source As Object)
  Dim objTextBox As TextBox = TryCast(source, TextBox)
  Try
    Dim grados As Double
    ' Si se escribió en la caja de texto grados centígrados...
    If objTextBox Is ctGradosC Then
      grados = Convert.ToDouble(ctGradosC.Text) * 9.0 / 5.0 + 32.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosF.Text = String.Format("{0:F2}", grados)
    End If
    ' Si se escribió en la caja de texto grados Fahrenheit...
    If objTextBox Is ctGradosF Then
      grados = (Convert.ToDouble(ctGradosF.Text) - 32.0) * 5.0 / 9.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosC.Text = String.Format("{0:F2}", grados)
    End If
  Catch exc As FormatException
    ctGradosC.Text = "0,00"
    ctGradosF.Text = "32,00"
  End Try
End Sub
```

El código completo de esta aplicación puede obtenerlo del CD en la carpeta *Cap02\ConverTemps2*.

Estado del teclado

La clase **Keyboard** proporciona propiedades, métodos y eventos que facilitan información sobre el estado del teclado. De la funcionalidad proporcionada por la misma cabe destacar la propiedad **Modifiers** que nos permitirá obtener información acerca de las teclas modificadoras, *Alt*, *Mayús* (*Shift*) o *Ctrl* (*Control*), que se han pulsado. Algunos ejemplos pueden ser *Ctrl+F2*, *Ctrl+Alt+I* o *Ctrl*/*Shift+F1*. El siguiente ejemplo muestra cómo detectar si se han producido estas pulsaciones:

```
Private Sub textBox1_PreviewKeyDown(sender As Object,
e As KeyEventArgs)
  ' Si se pulsó Ctrl + F2...
  If Keyboard.Modifiers = ModifierKeys.Control AndAlso
     e.Key = Key.F2 Then
    ' ...

  End If

  ' Si se pulsó Ctrl + Alt + I...
  If Keyboard.Modifiers = (ModifierKeys.Control Or
     ModifierKeys.Alt) AndAlso e.Key = Key.I Then
    ' ...

  End If

  ' Si se pulsó (Ctrl o Shift) + F1...
  Dim teclaControl As Boolean =
        (Keyboard.Modifiers And ModifierKeys.Control) <> 0
  Dim teclaMayus As Boolean =
        (Keyboard.Modifiers And ModifierKeys.Shift) <> 0
  If (teclaControl OrElse teclaMayus) AndAlso e.Key = Key.F1 Then
    ' ...

  End If
End Sub
```

La clase **Keyboard** proporciona también los métodos **IsKeyDown**, que indica si la tecla especificada está pulsada, **IsKeyToggled**, que informa de si la tecla especificada está en estado de alternancia, e **IsKeyUp**, que indica si la tecla especificada está sin presionar.

También, a través de la propiedad **FocusedElement** podemos obtener el elemento que tiene el foco de teclado y, por medio del método **Focus**, establecer el foco de teclado en el elemento especificado.

VALIDACIÓN DE UN CAMPO DE TEXTO

Validar un campo de texto equivale a restringir su contenido al conjunto de caracteres válidos para dicho campo. Si la validación de los datos se hace después de pulsar la tecla *Entrar* en la caja de texto, el campo podría contener un dato no válido, pero podría ser validado antes de utilizarlo. En cambio, si la validación se hace verificando la validez de cada tecla pulsada (evento **PreviewTextInput**), el campo de texto ya estará validado una vez finalizada la entrada.

¿Qué sucede cuando una caja de texto tiene el foco y el usuario pulsa una tecla? Pues que el control genera todos o algunos de los eventos del teclado expuestos anteriormente, **PreviewKeyDown**, **KeyDown**, **PreviewTextInput**, **PreviewKeyUp** y **KeyUp**, dependiendo de la tecla pulsada; los dos primeros eventos se generan cuando se pulsa la tecla, el tercero cuando se va a escribir el carácter y los dos siguientes cuando se suelta la tecla. Estos eventos podrán ser interceptados y respondidos por los correspondientes métodos, si el control tiene asociado un controlador de eventos para cada uno de ellos. Por ejemplo, si quisiéramos manejar algunos de estos eventos cuando sean generados por una caja de texto *textBox1*, añadiríamos los siguientes controladores:

```
Private Sub textBox1_PreviewKeyDown(sender As Object,
e As KeyEventArgs)
    ' ...
End Sub

Private Sub textBox1_PreviewTextInput(sender As Object,
e As TextCompositionEventArgs)
    ' ...
End Sub
```

A diferencia de los eventos **PreviewKeyDown**, **KeyDown**, **PreviewKeyUp** y **KeyUp**, el evento **PreviewTextInput** se genera solamente cuando se introduce un carácter ASCII imprimible; por ejemplo, los dígitos del *0* al *9* o las teclas de la *a* a la *z*. Al hablar de caracteres imprimibles se excluyen teclas especiales, como la tecla de *retroceso* (*BackSpace*), las teclas de función (*F1* a *F12*), las teclas de movimiento del cursor (←, ↑, →, ↓) o la tecla *Supr* (*Del*). Para interceptar estas teclas especiales se utilizarán los eventos **PreviewKeyDown**, **KeyDown**, **PreviewKeyUp** y **KeyUp** que se generen.

Apliquemos lo expuesto a la aplicación *ConverTemps2*. El contenido de las cajas de texto *ctGradosC* y *ctGradosF* será válido cuando sus caracteres pertenezcan al siguiente conjunto: +–,1234567890. El signo + o – sólo puede aparecer al principio del dato y éste sólo puede contener una coma decimal.

Según lo expuesto, para validar el contenido de las cajas de texto verificando cada tecla pulsada podríamos reemplazar el controlador *Grid_PreviewKeyDown* por el controlador *ValidarTecla_PreviewTextInput* del evento **PreviewTextInput** generado cuando se escribe un carácter. De forma resumida, este método realizará las siguientes operaciones:

1. Si se pulsó la tecla *Entrar* (código ASCII \r), da por controlado el evento invocando el método *Conversion*.

2. Si se pulsó la tecla *coma* y ya había una, invalida esta última dando por controlado el evento.

3. Si se pulsó la tecla + o –, verifica que se trata del primer carácter tecleado. De no ser así, invalida este último dando por controlado el evento. En este caso, se tiene en cuenta que al escribir el signo como primer carácter, la caja de texto puede que no esté vacía, pero sí todo el texto seleccionado, en cuyo caso será automáticamente reemplazado. Si en la primera posición ya hay un carácter, invalida este último dando por controlado el evento.

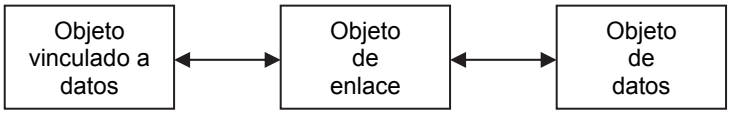
4. Si se pulsó otra tecla que no se corresponde con uno de los dígitos 0 a 9, se invalida dando por controlado el evento.

```
Private Sub ValidarTecla_PreviewTextInput(sender As Object,
e As TextCompositionEventArgs)
  If e.Text = vbCr Then
    ' Se pulsó la tecla Entrar
    e.Handled = True
    Conversion(e.Source)
  ElseIf e.Text = "," Then
    Dim ObjTextBox As TextBox = DirectCast(e.Source, TextBox)
    If ObjTextBox.Text.IndexOf(","c) <> -1 Then
      ' Sólo puede haber una coma
      e.Handled = True
    End If
  ElseIf e.Text = "-" OrElse e.Text = "+" Then
    Dim ObjTextBox As TextBox = DirectCast(e.Source, TextBox)
    ' Admitir - o + sólo en la primera posición:
    If ObjTextBox.SelectionLength = ObjTextBox.Text.Length Then
      ' Todo el texto está seleccionado: se sobrescribe con el signo
      e.Handled = False
    ElseIf ObjTextBox.Text.Length <> 0 Then
      ' La primera posición ya está ocupada
      e.Handled = True
    End If
  ElseIf e.Text.CompareTo("0") < 0 OrElse e.Text.CompareTo("9") > 0 Then
    ' Desechar los caracteres que no son dígitos
    e.Handled = True
```

```
  End If
End Sub
```

Obsérvese que la tecla pulsada se valida antes de que el procedimiento predeterminado por el sistema añada el carácter a la caja de texto.

No obstante, para facilitar el proceso de validación, WPF proporciona un modelo de enlace de datos que permite aplicar reglas de validación a un objeto vinculado a datos (destino de los datos). El escenario sería éste:



En este escenario, el motor de enlace de datos de WPF comprobará cada regla de validación asociada a un enlace cada vez que se transfiera un valor entre el objeto vinculado a datos y el objeto que almacena los datos.

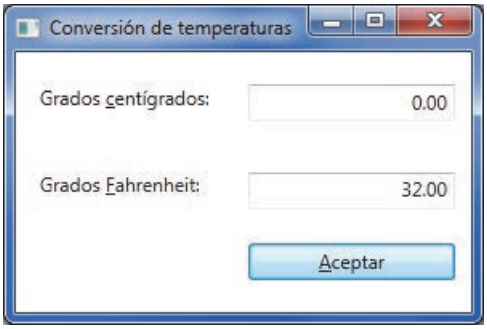
Según lo expuesto, antes de abordar la validación de datos proporcionada por el motor de enlace de datos de WPF, vamos a estudiar cómo se implementa la vinculación de datos.

ENLACE DE DATOS

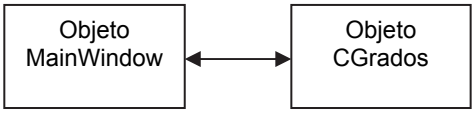
La finalidad de la mayoría de las aplicaciones es mostrar datos a los usuarios con el fin de que puedan editarlos. Esto sugiere pensar que habrá que transportar datos entre un origen y un destino; por ejemplo, entre el modelo de objetos obtenido del modelo de datos relacional de una base de datos y la interfaz gráfica de la aplicación que consume esos datos. Este proceso, sin un motor para transportar los datos adelante y atrás, requeriría escribir bastante código, pero con el motor de enlace de datos de WPF comprobaremos que se trata de un proceso sencillo. Para aclarar lo expuesto, vamos a estudiar a continuación ambos casos.

Enlace de datos sin el motor de WPF

Consideremos de nuevo la primera versión de la aplicación conversión de temperaturas, como muestra la siguiente figura, y añadamos la validación para las cajas de texto expuesta en el apartado anterior.



Supongamos ahora que los datos que va a mostrar la ventana principal proceden de un objeto de una clase *CGrados* y que ambos, ventana y objeto, deberán permanecer sincronizados.



Podemos escribir la clase *CGrados*, que representa los datos que muestra la ventana principal, como se indica a continuación:

```
Public Class CGrados
  Private _gradosC As Double = 0.0
  Private _gradosF As Double = 32.0

  Public Property GradosC() As Double
    Get
      Return _gradosC
    End Get
    Set(value As Double)
      _gradosC = value
    End Set
  End Property

  Public Property GradosF() As Double
    Get
      Return _gradosF
    End Get
    Set(value As Double)
      _gradosF = value
    End Set
  End Property
End Class
```

Añada al proyecto una carpeta *Clases* y guarde esta clase dentro de esta carpeta.

Apoyándonos en la clase *CGrados* modificamos la clase *MainWindow* como se muestra a continuación:

```
Class MainWindow
  Private objTextBox As TextBox = Nothing
  Private objGrados As New CGrados()

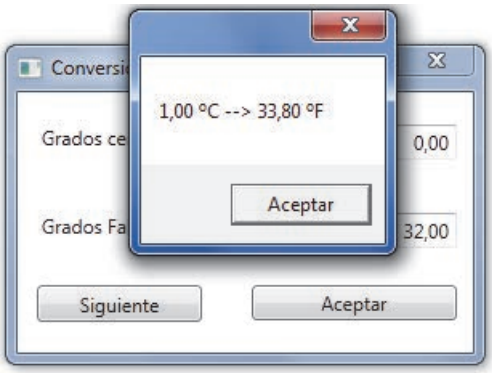
  Private Sub formPrincipal_Loaded(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ctGradosC.Focus()
    ctGradosC.Text = String.Format("{0:F2}", objGrados.GradosC)
    ctGradosF.Text = String.Format("{0:F2}", objGrados.GradosF)
  End Sub

  Private Sub btAceptar_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ' ...
  End Sub

  ' ...
End Class
```

El código anterior crea un objeto *CGrados*, iniciado con los valores especificados en esta clase, e inicia las cajas de texto con las propiedades *GradosC* y *GradosF* de dicho objeto.

Continuando con el ejemplo, añadamos a la interfaz de usuario un botón *Siguiente* que, partiendo del objeto *CGrados* actual, proporcione el siguiente objeto *CGrados* que sea de un grado centígrado más que el anterior.



Para ello, añada el controlador del evento **Click** de *btSiguiente* para que invoque al método **MessageBox.Show** y muestre una ventana, como la que muestra en primer plano la figura anterior, con la información requerida:

```
Private Sub btSiguiente_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  objGrados.GradosC += 1
  objGrados.GradosF = objGrados.GradosC * 9.0 / 5.0 + 32.0
  MessageBox.Show(String.Format("{0:F2} ºC --> {1:F2} ºF",
                  objGrados.GradosC, objGrados.GradosF))
End Sub
```

Cuando se pulsa el botón *Siguiente* observamos que los grados centígrados del objeto *CGrados* se incrementan en una unidad y los grados Fahrenheit se actualizan al valor correspondiente, pero no se actualiza la interfaz gráfica. Una forma de actualizar la interfaz gráfica del usuario sería escribir código que actualice las propiedades **Text** de las cajas cada vez que se modifique el objeto *CGrados*.

```
Private Sub btSiguiente_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  objGrados.GradosC += 1
  objGrados.GradosF = objGrados.GradosC * 9.0 / 5.0 + 32.0
  ctGradosC.Text = String.Format("{0:F2}", objGrados.GradosC)
  ctGradosF.Text = String.Format("{0:F2}", objGrados.GradosF)
  MessageBox.Show(String.Format("{0:F2} ºC --> {1:F2} ºF",
                  objGrados.GradosC, objGrados.GradosF))
End Sub
```

Con dos líneas de código hemos solucionado el problema planteado, pero también observamos que cuando escribimos un nuevo valor en las cajas de texto, el objeto *CGrados* no se actualiza. Para solucionarlo, podemos añadir al método *btAceptar_Click* las dos líneas de código indicadas a continuación:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Try
    Dim grados As Double
    ' ...
    objGrados.GradosC = Double.Parse(ctGradosC.Text)
    objGrados.GradosF = Double.Parse(ctGradosF.Text)
  Catch exc As FormatException
    ctGradosC.Text = "0,00"
    ctGradosF.Text = "32,00"
  End Try
End Sub
```

Otra forma de registrar en el objeto *CGrados* los cambios que ocurren en la interfaz gráfica de usuario es observar cuándo cambia la propiedad **Text** de las cajas de texto. Cuando cambia el contenido de cualquiera de los elementos de texto se genera, entre otros, el evento **TextChanged** que podemos interceptar para actualizar la propiedad correspondiente del objeto *CGrados* (asignar a **Text** la mis-

ma cadena dos veces consecutivas no genera este evento dos veces, porque la segunda asignación no cambia el valor de **Text**). Según esto, deje el método *btAceptar_Click* como estaba, añada los controladores para manejar el evento **TextChanged** de las cajas de texto y edite los métodos correspondientes así:

```
Private Sub ctGradosC_TextChanged(sender As System.Object,
e As System.Windows.Controls.TextChangedEventArgs)
  Dim gradosC As Double = 0
  If Double.TryParse(ctGradosC.Text, gradosC) Then
    objGrados.GradosC = gradosC
  End If
End Sub

Private Sub ctGradosF_TextChanged(sender As System.Object,
e As System.Windows.Controls.TextChangedEventArgs)
  Dim gradosF As Double = 0
  If Double.TryParse(ctGradosF.Text, gradosF) Then
    objGrados.GradosF = gradosF
  End If
End Sub
```

Evidentemente, el ejemplo es muy sencillo y hemos requerido escribir muy poco código para sincronizar la interfaz gráfica del usuario con un origen de datos *CGrados*, pero no resultará así de sencillo cuando la aplicación sea bastante más compleja, por lo que esperamos que WPF nos proporcione algo mejor.

Notificar cuándo cambia una propiedad

Una forma más profesional y más robusta de que la interfaz gráfica registre los cambios del objeto con el que está vinculada es que el objeto genere un evento cuando cambie alguna de sus propiedades para que la interfaz gráfica lo pueda interceptar y responder actualizándose. Una forma sencilla de hacer esto es que la clase del objeto implemente la interfaz **INotifyPropertyChanged** del espacio de nombres **System.ComponentModel**.

La interfaz **INotifyPropertyChanged** se utiliza para notificar a los clientes, generalmente enlazados, que una propiedad ha cambiado. Esta interfaz proporciona el evento **PropertyChanged** que habrá que generar cuando el valor de una propiedad cambie.

Por ejemplo, consideremos el objeto *CGrados* con sus propiedades *gradosC* y *gradosF*. Si este objeto implementa la interfaz **INotifyPropertyChanged** puede notificar que cambió una de sus propiedades generando el evento **PropertyChanged** según se indica a continuación:

```
Imports System.ComponentModel

Class CGrados
  Implements INotifyPropertyChanged
  Private _gradosC As Double = 0.0
  Private _gradosF As Double = 32.0

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged

  Private Sub NotificarCambio(nombreProp As String)
    RaiseEvent
      PropertyChanged(Me, New PropertyChangedEventArgs(nombreProp))
  End Sub

  Public Property GradosC() As Double
    Get
      Return _gradosC
    End Get
    Set(value As Double)
      _gradosC = value
      NotificarCambio("GradosC")
    End Set
  End Property

  Public Property GradosF() As Double
    Get
      Return _gradosF
    End Get
    Set(value As Double)
      _gradosF = value
      NotificarCambio("GradosF")
    End Set
  End Property
End Class
```

El código anterior muestra que un objeto *CGrados* generará un evento **PropertyChanged** cada vez que cambie el valor de alguna de sus propiedades. Observe que cuando se asigna un nuevo valor a la propiedad se invoca al método *NotificarCambio* que generará dicho evento. También podemos observar que el método que responderá a este evento tiene dos parámetros: el primero es una referencia al objeto *CGrados* que genera el evento y el segundo, los datos de tipo **PropertyChangedEventArgs** relacionados con el evento, que en este caso se corresponden con el nombre de la propiedad que ha cambiado.

Ahora, podemos utilizar este evento para mantener la interfaz del usuario sincronizada con el objeto *CGrados*, operación que antes realizábamos manualmente en el método *btSiguiente_Click* y que ahora, al no realizarla, quedará así:

```
Private Sub btSiguiente_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  objGrados.GradosC += 1
  objGrados.GradosF = objGrados.GradosC * 9.0 / 5.0 + 32.0
  MessageBox.Show(String.Format("{0:F2} ºC --> {1:F2} ºF",
                  objGrados.GradosC, objGrados.GradosF))
End Sub
```

La modificación que realiza el método anterior sobre cada una de las propiedades *gradosC* y *gradosF* generará para cada una de ellas el evento **PropertyChanged** que la ventana principal, clase *MainWindow*, interceptará para actualizar la caja de texto correspondiente. Según esto, añada el controlador para este evento y complete el código como se indica a continuación:

```
Imports System.ComponentModel;

Class MainWindow
  Private objTextBox As TextBox = Nothing
  Private objGrados As New CGrados()

  Private Sub formPrincipal_Loaded(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ctGradosC.Focus()
    ctGradosC.Text = String.Format("{0:F2}", objGrados.GradosC)
    ctGradosF.Text = String.Format("{0:F2}", objGrados.GradosF)
    AddHandler objGrados.PropertyChanged,
        AddressOf objGrados_PropertyChanged
  End Sub

  Private Sub objGrados_PropertyChanged(sender As Object,
  e As PropertyChangedEventArgs)
    Select Case e.PropertyName
      Case "GradosC"
        ctGradosC.Text = String.Format("{0:F2}", objGrados.GradosC)
        Exit Select
      Case "GradosF"
        ctGradosF.Text = String.Format("{0:F2}", objGrados.GradosF)
        Exit Select
    End Select
  End Sub
    ' ...
End Class
```

Observe en el código anterior que el método que responderá al evento **PropertyChanged** es *objGrados_PropertyChanged* y que éste, en función de la propiedad del objeto *CGrados* que haya cambiado, actualiza la caja de texto correspondiente con ese nuevo valor.

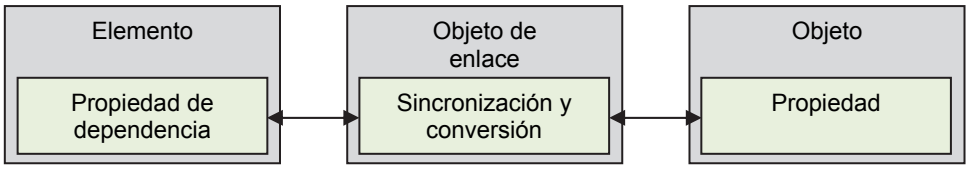
Ahora, con independencia de dónde cambien los datos, tanto el objeto *CGrados* como la interfaz gráfica permanecen sincronizados. Resumiendo, conseguir esta sincronización ha supuesto:

- Establecer los valores iniciales de las cajas de texto y definir el controlador para el evento **PropertyChanged** en el método que responde al evento **Loaded** de *MainWindow*.
- Escribir el método *objGrados_PropertyChanged* que responderá al evento **PropertyChanged** para actualizar las cajas de texto con los nuevos valores del objeto *CGrados*.
- Añadir a la clase *MainWindow* el controlador del evento **TextChanged** para cada una de las cajas de texto.
- Escribir los métodos *ctGradosC_TextChanged* y *ctGradosF_TextChanged* que responderán al evento **TextChanged** para actualizar el objeto *CGrados* con los nuevos valores de las cajas de texto.

Según lo expuesto, cabe pensar que a medida que aumente el número de propiedades del objeto o el número de objetos que estemos manejando, el código que tendríamos que escribir sería mayor. Para minimizar este problema WPF proporciona un modelo de enlace de datos que, además, permite aplicar reglas de validación para los datos, aspectos ambos que estudiaremos a continuación.

Enlace de datos con el motor de WPF

Según lo expuesto hasta ahora, podemos decir que el enlace de datos es un mecanismo que establece una conexión entre la interfaz gráfica del usuario de una aplicación (en nuestro ejemplo, la definida por la ventana *MainWindow*) y los datos proporcionados por objetos pertenecientes a la lógica de negocio de dicha aplicación (en nuestro ejemplo, por el objeto *CGrados*). Dicho mecanismo permite que los elementos que están enlazados a los objetos de datos reflejen los cambios automáticamente cuando los datos cambian su valor y viceversa, que los objetos reflejen los cambios automáticamente cuando los elementos de la interfaz cambian su valor. Según lo expuesto, la base de este mecanismo es vincular/enlazar pares de propiedades y proveer un mecanismo de notificación de cambios; en el ejemplo que acabamos de desarrollar, cada par está compuesto por una propiedad del objeto *CGrados* y la propiedad **Text** de la caja de texto correspondiente y el mecanismo de notificación de cambios lo proporciona la interfaz **INotifyPropertyChanged**. WPF sigue el mismo mecanismo pero dejando todo el proceso de sincronización y conversión de los datos para el objeto de enlace, según muestra la figura siguiente. El objeto de enlace es proporcionado por la clase **Binding**.



En WPF, normalmente, cada enlace tiene estos cuatro componentes:

- Un objeto destino del enlace (el *elemento*).
- Una propiedad de destino, que debe ser una *propiedad de dependencia*.
- Un objeto origen del enlace (el *objeto*).
- Y una ruta de acceso al valor que se va a usar del objeto origen del enlace (la *propiedad* del objeto).

Por ejemplo, en la aplicación anterior el objeto destino del enlace es una caja de texto, la propiedad de destino es su propiedad **Text**, el objeto origen del enlace es el objeto *CGrados* y el valor que se usa del objeto origen del enlace es su propiedad *GradosC* o *GradosF*.

El ejemplo expuesto no nos debe hacer pensar que el objeto asociado con el objeto origen del enlace esté restringido a ser un objeto CLR personalizado. Por ejemplo, el objeto origen del enlace puede ser también un objeto **UIElement**, cualquier objeto de lista, un objeto CLR asociado a datos de ADO.NET o servicios webs, o bien un objeto **XMLNode** que contiene datos XML. Tampoco nos debe hacer pensar que la dirección del flujo de datos tiene que ser en los dos sentidos. Esto es, el flujo de datos de un enlace puede ir desde el destino al origen (*OneWayToSource*), desde el origen al destino (*OneWay*) o en ambas direcciones (*TwoWay*). Este comportamiento será establecido por medio de la propiedad **Mode** del objeto de enlace (objeto de tipo **Binding**); el valor predeterminado de esta propiedad varía para cada propiedad de dependencia. En general, las propiedades de un control que puedan ser modificadas por el usuario, como las de las cajas de texto y casillas de verificación, tienen como valor predeterminado enlaces bidireccionales, mientras que la mayoría de las demás propiedades tienen como valor predeterminado enlaces unidireccionales.

Recordar también que, según comentamos en el capítulo anterior, las propiedades de un objeto **UIElement** son propiedades de dependencia y la mayoría de las propiedades de dependencia, excepto las de sólo lectura, admiten el enlace de datos de forma predeterminada.

Hemos dicho que los enlaces *TwoWay* y *OneWayToSource* permiten actualizar el objeto origen del enlace pero, ¿cuándo ocurre la actualización? Esto está definido por la propiedad **UpdateSourceTrigger** del enlace (objeto **Binding**). Por ejemplo, si el valor de esta propiedad es *PropertyChanged*, el objeto origen del

enlace se actualizará en cuanto cambie la propiedad de destino, pero si el valor es *LostFocus*, se actualizará cuando la propiedad de destino pierda el foco. El valor predeterminado para la propiedad **UpdateSourceTrigger** del enlace es *Default*, lo que significa que se utilizará el valor predeterminado de **UpdateSourceTrigger** de la propiedad de dependencia del destino. Esto es, igual que dijimos para la propiedad **Mode**, cada propiedad de dependencia tiene asignado un valor de **UpdateSourceTrigger** predeterminado; este valor, para la mayoría de las propiedades de dependencia es *PropertyChanged*, mientras que para la propiedad **Text** es *LostFocus*.

La clase Binding

Un enlace de datos en WPF viene definido por un objeto de la clase **Binding** o de una clase derivada de **BindingBase**. Este objeto conecta, de forma transparente para el desarrollador, las propiedades de dos objetos diferentes: uno se corresponde con el destino del enlace (normalmente, elementos de WPF) y el otro, con el origen del enlace (objeto de datos; por ejemplo, una base de datos, un archivo XML o cualquier objeto que contenga datos), con el fin de sincronizar los valores de sus propiedades.

La clase **Binding** proporciona muchas facilidades para gestionar el enlace entre dos propiedades, pero la que más a menudo utilizará será la propiedad **Path** para especificar la ruta de acceso a los datos (o **XPath** si los datos están en XML). Para la mayoría de los casos, **Path** hará referencia al nombre de la propiedad del objeto de datos (por ejemplo *GradosC*). Otras propiedades son **Mode** y **UpdateSourceTrigger** que ya hemos comentado, **ElementName** utilizada cuando el origen de datos es otro elemento de la interfaz, **Source** que permite especificar el objeto que se va a utilizar como origen de los datos, invalidando el contexto de datos predeterminado, o la propiedad **ValidationRules** que permite asociar al enlace reglas que comprueban la validez de los datos proporcionados por el usuario.

El siguiente ejemplo sincroniza, en las dos direcciones, las propiedades *GradosC* del objeto *objGrados* y **Text** de la caja de texto *ctGradosC*:

```
Dim objGrados As New CGrados()
Dim enlace As New Binding("GradosC")
enlace.Source = objGrados
enlace.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged
enlace.ConverterCulture =
  New System.Globalization.CultureInfo("es-ES", [False])
ctGradosC.SetBinding(TextBox.TextProperty, enlace)
```

TextProperty identifica la propiedad de dependencia **Text** de **TextBox** (véase el apartado *Propiedades de dependencia* en el capítulo *Aplicación WPF*).

Contexto de datos

Todo elemento **FrameworkElement** y **FrameworkContentElement** tiene una propiedad **DataContext** que define un contexto de datos. El aspecto más importante de un contexto de datos es hacer referencia al origen de datos que se utiliza para un enlace y permitir a los elementos secundarios del árbol de elementos que define la interfaz gráfica heredar esa información del enlace cuando haya sido definida en un elemento primario.

Por ejemplo, una alternativa al código anterior podría ser ésta:

```
Dim objGrados As New CGrados()
ctGradosC.DataContext = objGrados
Dim enlace As New Binding()
enlace.Path = New PropertyPath("GradosC")
enlace.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged
enlace.ConverterCulture =
  New System.Globalization.CultureInfo("es-ES", [False])
ctGradosC.SetBinding(TextBox.TextProperty, enlace)
```

El contexto de datos para el origen de datos también podría ser el correspondiente al elemento padre de la caja de texto (el **Grid**); de esta forma, sería heredado por todos los elementos hijo:

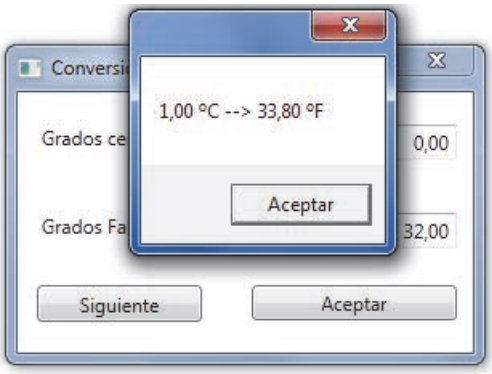
```
Dim objGrados As New CGrados()
gridVentPpal.DataContext = objGrados
Dim enlace As New Binding()
enlace.Path = New PropertyPath("GradosC")
enlace.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged
enlace.ConverterCulture =
  New System.Globalization.CultureInfo("es-ES", [False])
ctGradosC.SetBinding(TextBox.TextProperty, enlace)
```

Estas mismas definiciones se pueden hacer en XAML o como una combinación de código subyacente y XAML, como veremos en los siguientes apartados.

Crear un enlace

Para aplicar los conocimientos expuestos en el apartado anterior, vamos a explicar cómo configurar un enlace. Recuperemos de nuevo la primera versión de la aplicación conversión de temperaturas, a la que añadiremos la clase *CGrados*, del ejercicio anterior, que implementa la interfaz **INotifyPropertyChanged**, ya que esta interfaz es la que utiliza el motor de enlace a datos de WPF para mantener la interfaz gráfica sincronizada con el objeto origen del enlace.

Definimos en la clase *MainWindow* el objeto *objGrados* de la clase *CGrados* y añadimos también el botón *Siguiente* para que realice la misma función que en la última versión, según muestra la figura siguiente.



Una vez ejecutados los pasos anteriores, si ejecuta la aplicación observará que la interfaz gráfica y el objeto *CGrados* aún no están sincronizados. Para sincronizarlos hay que crear un enlace; recuerde que un enlace se establece mediante un objeto **Binding** y que cada enlace tiene normalmente cuatro componentes: el objeto destino del enlace (en nuestro caso el objeto **TextBox**), la propiedad de destino (la propiedad **Text** de **TextBox**), el objeto origen del enlace (el objeto *CGrados*) y una ruta de acceso al valor del objeto origen (propiedad de *CGrados* que se va a utilizar).

Origen de datos implícito

WPF obtiene el origen de datos de forma predeterminada del contexto de datos especificado por la propiedad **DataContext** del objeto destino de los datos o de alguno de sus elementos padre. En nuestro ejemplo, como son dos las cajas de texto que tenemos que enlazar con el origen de datos, estableceríamos el origen de datos en el contexto de datos del **Grid**:

```
Private Sub formPrincipal_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctGradosC.Focus()
  gridVentPpal.DataContext = objGrados
End Sub
```

En el código anterior, *gridVentPpal* es el nombre que hemos asignado al **Grid** que contiene las dos cajas de texto.

Para especificar los otros componentes del enlace, el objeto destino, la propiedad de destino y la ruta de acceso al valor del origen, podemos hacerlo en

XAML en la definición del elemento destino, utilizando el formato de propiedad como elemento, así:

```
<TextBox ... >
  <TextBox.Text>
    <Binding Path="GradosC"
     UpdateSourceTrigger=PropertyChanged
     ConverterCulture="es-ES" />
  </TextBox.Text>
</TextBox>
```

O bien, utilizando el formato de propiedad como atributo, así:

```
<TextBox ... Text="{Binding Path=GradosC,
                    UpdateSourceTrigger=PropertyChanged,
                    ConverterCulture=es-ES}"
 ... />
```

O de forma más abreviada, así:

```
<TextBox ... Text="{Binding GradosC,
                    UpdateSourceTrigger=PropertyChanged,
                    ConverterCulture=es-ES}"
 ... />
```

En el código anterior podemos observar las propiedades utilizadas con el objeto de enlace que se crea a partir de la clase System.Windows.Data.**Binding**: **Path**, para establecer la ruta de acceso a la propiedad del objeto origen del enlace (propiedad *GradosC* del objeto *CGrados*), **UpdateSourceTrigger** para establecer que la actualización del objeto *CGrados* (objeto origen del enlace) ocurra cuando la propiedad **Text** de la caja de texto cambie (de forma predeterminada se hace cuando pierde el foco: *LostFocus*, con lo que, en nuestro caso, sólo se actualizaría la propiedad de *CGrados* enlazada con la caja de texto que pierde el foco y no las dos propiedades de *CGrados*) y la propiedad **ConverterCulture** para establecer la referencia cultural; si no establece esta propiedad, el motor de enlaces utiliza la propiedad **Language** del objeto destino del enlace (en XAML, el valor predeterminado es "en-US" o se hereda si se ha establecido en algún elemento más arriba en la jerarquía).

Aplique lo expuesto para cada una de las cajas de texto. Después, ejecute la aplicación y observe los resultados. Comprobará que tanto las cajas de texto como el objeto de datos están sincronizados en ambas direcciones.

En el apartado *Contexto de datos* pudimos ver cómo se puede realizar la configuración de un enlace de datos utilizando código Visual Basic. A continuación, en esta nueva versión de la aplicación conversión de temperaturas que acabamos

de exponer, para configurar el enlace hemos combinado código Visual Basic, en el fichero de código subyacente, para definir el objeto *CGrados* y asignarlo al contexto de datos del **Grid**, y código XAML para el resto de la configuración. Esta parte que hemos implementado en código Visual Basic podría escribirse también en XAML así:

```
<Window ...
        xmlns:local="clr-namespace:ConverTemps" >
  <Window.Resources>
    <local:CGrados x:Key="objGrados" GradosC="0.00" GradosF="32.00" />
  </Window.Resources>
  <Grid Name="gridVentPpal"
        DataContext="{StaticResource objGrados}">
  ...
</Window>
```

En XAML, cualquier objeto de una clase con un constructor por omisión (predeterminado) se puede iniciar, pero no con un constructor específico porque XAML no tiene sintaxis para invocar a un constructor específico (la sintaxis a la que nos referimos está previsto introducirla en una próxima versión de XAML). En el código anterior, en la sección de recursos de la ventana, se ha definido un recurso *CGrados* nombrado *objGrados* que dará lugar a un objeto de esa clase iniciado con los valores ahí especificados. La utilización de la clase *CGrados* en esta sección nos exige declarar el espacio de nombres al que pertenece, para lo cual hemos utilizado la sintaxis *clr-namespace*. El espacio de nombres ha sido asignado al prefijo *local* (el nombre del prefijo es elegido por el desarrollador). Finalmente, hemos fijado declarativamente el origen de datos definido por el recurso *objGrados* en el contexto de datos del **Grid**.

El trasladar la creación del objeto *CGrados* al código XAML nos obliga a modificar, según se indica a continuación, el controlador del evento **Click** del botón *Siguiente* para que pueda acceder a este objeto.

```
Private Sub btSiguiente_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim objGrados As CGrados = TryCast(Me.FindResource("objGrados"), CGrados)
  objGrados.GradosC += 1
  objGrados.GradosF = objGrados.GradosC * 9.0 / 5.0 + 32.0
  MessageBox.Show(String.Format("{0:F2} ºC --> {1:F2} ºF",
                  objGrados.GradosC, objGrados.GradosF))
End Sub
```

En el método anterior vemos que para tener acceso al objeto *CGrados* declarado en XAML en los recursos de la ventana principal (`<Window.Resources>`), obtenemos una referencia a dicho objeto invocando al método **FindResource** de la ventana pasando como argumento el nombre del recurso.

Origen de datos explícito

En lugar de obtener el origen de datos de forma predeterminada del contexto de datos especificado por la propiedad **DataContext** del objeto destino de los datos o de alguno de sus elementos padre, podemos especificar explícitamente en este objeto destino cuál es su fuente (**Source**) de datos. Esto es útil especialmente cuando hay más de una fuente de datos. Por ejemplo:

```
Dim obj1 As New CDatos()
Dim obj2 As New CDatos()

Dim enlace1 As New Binding("Prop1")
enlace1.Source = obj1
textBox1.SetBinding(TextBox.TextProperty, enlace1)

Dim enlace2 As New Binding("Prop2")
enlace2.Source = obj2
textBox2.SetBinding(TextBox.TextProperty, enlace2)
```

El código anterior enlaza dos cajas de texto con dos objetos *CDatos* diferentes, estableciendo la propiedad **Source** de cada objeto de enlace con el correspondiente objeto *CDatos*.

Enlaces con otros controles

También es posible enlazar la propiedad de un elemento con la propiedad de otro elemento especificado por la propiedad **ElementName** del enlace. Por ejemplo, sincronizar el color del texto de la caja de texto *ctGradosF* con el color del texto de *ctGradosC*; de esta forma, cuando cambiemos el color del texto de la caja *ctGradosC* (por ejemplo, a rojo cuando el valor mostrado sea menor o igual que 3 ºC) también cambiará al mismo color el texto de *ctGradosF*.

```
<TextBox Name="ctGradosF" ...
  Text="{Binding Path=GradosF, ...}"
  Foreground="{Binding Path=Foreground, ElementName=ctGradosC}"
... />
```

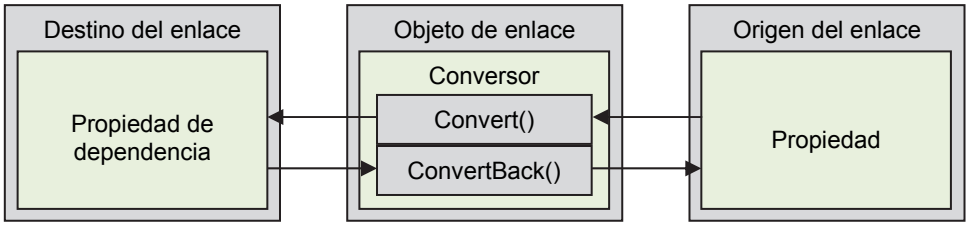
El código anterior enlaza la propiedad **Foreground** de *ctGradosF* con la propiedad **Foreground** del elemento denominado *ctGradosC*. Para probar cómo trabaja este código, vamos a hacer que el color del texto de *ctGradosC* cambie a rojo cuando el valor mostrado por esta caja sea menor o igual que 3 ºC. Supongamos, en principio, que el cambio de color propuesto lo hacemos desde el controlador del botón *Siguiente*. En este caso, procederíamos así:

```
Private Sub btSiguiente_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  objGrados.GradosC += 1
  objGrados.GradosF = objGrados.GradosC * 9.0 / 5.0 + 32.0
  If objGrados.GradosC <= -3 Then
    ctGradosC.Foreground = Brushes.Red
  Else
    ctGradosC.Foreground = Brushes.Black
  End If
  MessageBox.Show(String.Format("{0:F2} ºC --> {1:F2} ºF",
                  objGrados.GradosC, objGrados.GradosF))
End Sub
```

Pero esta forma de proceder sólo es válida cuando actuamos sobre el botón *Siguiente* y no por el hecho de que la temperatura mostrada por *ctGradosC* tome un valor por debajo de -3 ºC debido a una entrada del usuario. Esto es, lo deseado es que cuando esta caja de texto muestre un valor de -3 o menos ºC, el color del texto de ambas cajas cambie a rojo y vuelva a su color en caso contrario. Esto lo podemos implementar utilizando conversores. Hay conversores predefinidos, por ejemplo para convertir un **Double** en un **String** y viceversa, que es lo que sucede entre los elementos **TextBox** y *CGrados* de nuestro ejemplo, pero otros, como convertir un **Double** en un objeto **Brush**, los tendremos que construir. Esto será lo siguiente que vamos a hacer; deshagamos entonces los cambios que hicimos antes en el controlador del botón *Siguiente*.

Conversores

Un conversor es una implementación de la interfaz **IValueConverter**. Esta interfaz aporta dos métodos: **Convert** y **ConvertBack**. El primero se invoca cuando la conversión ocurre en la dirección origen-destino y el segundo cuando la conversión ocurre en la dirección contraria.



Los objetos **Binding** y **MultiBinding** de WPF definen una propiedad **Convert** que permite proporcionar un convertidor que se utilizará durante el proceso de enlace. Cuando no se especifique explícitamente un convertidor de valores, WPF utilizará su base de conversores intrínsecos como son **DefaultValueCon-**

verter, **ObjectTargetConverter** o **SystemConvertConverter**, definidos, pero no documentados, en el espacio de nombres **MS.Internal.Data**.

Por ejemplo, si ejecuta de nuevo la última versión de la aplicación e introduce en la caja de texto de grados centígrados una cadena que no pueda ser convertida a un valor de tipo **Double**, observará que el borde de la caja se muestra en rojo. Si la vuelve a iniciar en modo depuración (*F10*), podrá verificar también que al ejecutarse el método **SystemConvertConverter.ConvertBack** se lanza una excepción de tipo **System.FormatException** porque la cadena de entrada no tiene el formato correcto. Esto quiere decir que cuando los conversores implícitos no satisfagan nuestras necesidades, tendremos que implementarlos a medida.

Por ejemplo, el siguiente código XAML permitirá asignar a la propiedad **Foreground** de tipo **Brush** de la caja de texto *ctGradosC* el valor de la propiedad *GradosC* de tipo **Double** de un objeto *CGrados*; evidentemente, esta asignación sólo será posible si hacemos corresponder a ese valor un objeto de tipo **Brush**. Precisamente ésta es la finalidad del conversor *converGradosCaBrush* especificado por el atributo **Converter** del enlace: asignar a la propiedad **Foreground** de *ctGradosC* el valor *GradosC* del objeto *CGrados* convertido en un objeto **Brush**.

```
<TextBox Name="ctGradosC" ...
  Foreground="{Binding Path=GradosC,
               Converter={StaticResource converGradosCaBrush}}"
... />
```

Siguiendo con el ejemplo, vamos a implementar el conversor *converGradosCaBrush* para convertir el valor de tipo **Double** de la propiedad *GradosC* del objeto *CGrados* en el objeto de tipo **Brush** válido para la propiedad **Foreground** de la caja de texto *ctGradosC*. Según lo expuesto, para crear este conversor hay que crear una clase que implemente la interfaz **IValueConverter** y, a continuación, definir los métodos **Convert** y **ConvertBack** aportados por dicha interfaz. La sintaxis de estos métodos es la siguiente:

```
Public Function Convert(value As Object, targetType As Type,
parameter As Object, culture As System.Globalization.CultureInfo) _
As Object Implements IValueConverter.Convert
   ' ...
End Function

Public Function ConvertBack(value As Object, targetType As Type,
parameter As Object, culture As System.Globalization.CultureInfo) _
As Object Implements IValueConverter.ConvertBack
   ' ...
End Function
```

El significado de los parámetros de estos métodos es el siguiente: *value* proporciona el valor a convertir, *targetType* especifica el tipo del objeto destino de la conversión, *parameter* permite usar la misma instancia del convertidor con parámetros distintos y *culture* indica la información de referencia cultural.

Pues bien, añada a la carpeta *Clases* del proyecto una nueva clase denominada *ConversorDoubleBrush* que implemente la interfaz **IValueConverter**. Después, edite el método **Convert** como se indica a continuación:

```
Public Class ConversorDoubleBrush
  Implements IValueConverter
  Public Function Convert(value As Object, targetType As Type,
  parameter As Object, culture As System.Globalization.CultureInfo) _
  As Object Implements IValueConverter.Convert
    ' Hacer la conversión sólo si se requiere un pincel
    If targetType <> GetType(Brush) Then
      Return Nothing
    End If
    Dim grados As Double = Double.Parse(value.ToString())
    Return If(grados <= -3, Brushes.Red, Brushes.Black)
  End Function

  Public Function ConvertBack(value As Object, targetType As Type,
  parameter As Object, culture As System.Globalization.CultureInfo) _
  As Object Implements IValueConverter.ConvertBack
    ' No se utiliza en este ejemplo
    Throw New NotImplementedException()
  End Function
End Class
```

El método **Convert** devuelve el pincel que se asignará a la propiedad **Foreground** de la caja de texto *ctGradosC*, propiedad que a su vez mantiene un enlace, realizado en el apartado anterior, con la propiedad **Foreground** de la caja de texto *ctGradosF*. Este pincel será rojo si la temperatura en grados centígrados es menor o igual que -3 y negro en otro caso.

El conversor implementado lo utilizaremos en la caja de texto *ctGradosC* así:

```
<TextBox Name="ctGradosC" ...
     Text="{Binding Path=GradosC, ...}"
     Foreground="{Binding Path=GradosC,
                  Converter={StaticResource converGradosCaBrush}}"
... />
```

En este código observamos que la propiedad **Foreground** de la caja de texto *ctGradosC* toma el valor de la propiedad *GradosC* del objeto *CGrados* convertido a **Brush** por medio del recurso *converGradosCaBrush*; eso es lo que especifica la

propiedad **Converter** del enlace. Este recurso hace referencia a un objeto de la clase *ConversorDoubleBrush* que acabamos de implementar. ¿Dónde definimos ese recurso? Según expusimos anteriormente en este mismo capítulo, lo definiremos en la sección de recursos de la ventana así:

```
<Window ...
        xmlns:local="clr-namespace:ConverTemps">
  <Window.Resources>
    <local:ConversorDoubleBrush x:Key="converGradosCaBrush" />
  </Window.Resources>
  <Grid Name="gridVentPpal">
...
</Window>
```

Ejecute ahora la aplicación y pruebe los resultados. Observará que cuando cambia la propiedad *GradosC* del objeto *CGrados* y toma un valor menor o igual que -3, el color del texto de la caja *ctGradosC* cambia a rojo y volverá a cambiar a negro cuando el valor de esa propiedad sea mayor que -3 grados. Así mismo, el color del texto de la caja *ctGradosF* se ajusta al mismo valor, ya que las propiedades **Foreground** de ambas cajas están sincronizadas.

Validación de datos

Los mecanismos de enlace de datos en WPF incluyen varias opciones para evaluar la validez de los datos de entrada cuando trabajamos con una vista editable. Además, la funcionalidad proporcionada por las plantillas y los estilos de WPF para los controles permite personalizar con facilidad la manera de indicar los errores de validación al usuario. La validación se produce durante la transferencia del valor del destino al origen de enlace y antes de que se llame al convertidor.

Las comprobaciones de validación pueden basarse en el tipo, formato, intervalo u otros requisitos específicos de la aplicación. En el apartado anterior vimos cómo el conversor **SystemConvertConverter** nos avisaba de si el formato de la cadena introducida era o no adecuado para el tipo de datos en el origen del enlace. Pues bien, en WPF es posible añadir reglas de validación a un enlace para verificar los datos de entrada cuando trabajamos con una vista editable. Estas reglas serán añadidas a la colección **ValidationRules** del objeto **Binding**. Cada regla es un objeto **ValidationRule** que comprueba si el valor de una propiedad es válido.

WPF proporciona dos tipos de objetos **ValidationRule**:

- **ExceptionValidationRule**. Representa una regla que verifica las excepciones que se producen durante la actualización de la propiedad origen del enlace.

Para establecer implícitamente esta regla basta con asignar el valor **True** a la propiedad **ValidatesOnExceptions** del enlace.

- **DataErrorValidationRule**. Representa una regla que comprueba si la implementación de **IDataErrorInfo** del objeto de origen ha generado errores. Por lo tanto, si el objeto origen del enlace implementa la interfaz **IDataErrorInfo**, puede utilizar esta regla para verificar la validez de los datos. Esta regla se puede también establecer implícitamente asignando el valor **True** a la propiedad **ValidatesOnDataErrors** del enlace.

También podemos crear nuestra propia regla de validación derivando de la clase **ValidationRule** e implementando el método **Validate**.

Regla ExceptionValidationRule

¿Cómo se añade la regla de validación **ExceptionValidationRule** a un enlace para verificar los datos de entrada cuando trabajamos con una vista editable? Hay dos formas de hacer esto; por ejemplo, en el código XAML siguiente se enlaza un control **TextBox** a la propiedad *GradosC* y se establece implícitamente la **ExceptionValidationRule**.

```
<TextBox ...
   Text="{Binding Path=GradosC,
     UpdateSourceTrigger=PropertyChanged,
     ConverterCulture=es-ES,
     ValidatesOnExceptions=True , NotifyOnValidationError=True }"
   ...
 />
```

Como alternativa, podemos también establecer explícitamente la regla **ExceptionValidationRule**. Para ello, tendremos que añadirla a la colección **ValidationRules** del objeto **Binding**. Por ejemplo, en el código XAML siguiente se enlaza un control **TextBox** a la propiedad *GradosC* y se agrega un objeto **ExceptionValidationRule** a la propiedad **Binding.ValidationRules**.

```
<TextBox ... >
  <TextBox.Text>
    <Binding Path="GradosC" UpdateSourceTrigger="PropertyChanged"
      ConverterCulture="es-ES" NotifyOnValidationError="True" >
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

En el ejemplo anterior, *GradosC* es de tipo **Double**. Entonces, cuando el usuario especifica un valor que no se puede convertir en un **Double**, se lanza una excepción. Cuando se lanza una excepción, el motor de enlace de datos crea un objeto **ValidationError** y lo añade a la propiedad adjunta **Validation**.**Errors** del elemento enlazado. Si no se lanza ninguna excepción, la colección estará vacía, a menos que otra regla genere un problema de validación. Además, si la propiedad **NotifyOnValidationError** de **Binding** es **True**, el motor de enlace generará en el elemento el evento adjunto **Validation.Error**. Para agregar un controlador para este evento, podemos hacerlo en XAML de la forma habitual o utilizando el método **AddErrorHandler** de **Validation**, como veremos un poco más adelante.

Según se desprende del párrafo anterior, WPF tiene una clase **Validation**, declarada **Shared** en el espacio de nombres **System.Windows.Controls**, la cual proporciona un número de propiedades asociadas que podemos aplicar a cualquier control. Las más interesantes son:

- **Errors**. Colección de todos los objetos **ValidationError** activos en el elemento enlazado. Esta colección se vacía cada vez que se inicia una transferencia entre el destino y el origen.
- **HasError**. Propiedad que vale **true** si la colección referenciada por la propiedad adjunta **Validation**.**Errors** del destino del enlace tiene algún error (objetos **ValidationError**).
- **ErrorTemplate**. Objeto **ControlTemplate** que se utiliza para generar los comentarios de los errores de validación en la capa de adorno.

Las propiedades asociadas están destinadas a utilizarse como un tipo de propiedad global configurable en cualquier objeto.

Por ejemplo, anteriormente hemos verificado que cuando se introduce en la caja de texto *ctGradosC* una cadena no convertible a **Double** el borde de la caja se muestra en rojo, pero no se muestra información de lo ocurrido. Si queremos obtener esta información, tendremos que interceptar el evento **Error**. Según esto, y suponiendo que ya está establecida la regla **ExceptionValidationRule**, añada a la clase *MainWindow* el código que se indica a continuación:

```
Public Sub New()
  InitializeComponent()
  gridVentPpal.DataContext = objGrados
  Validation.AddErrorHandler(Me.ctGradosC,
      AddressOf ctGrados_ValidationError)
End Sub

Private Sub ctGrados_ValidationError(sender As Object,
e As ValidationErrorEventArgs)
```

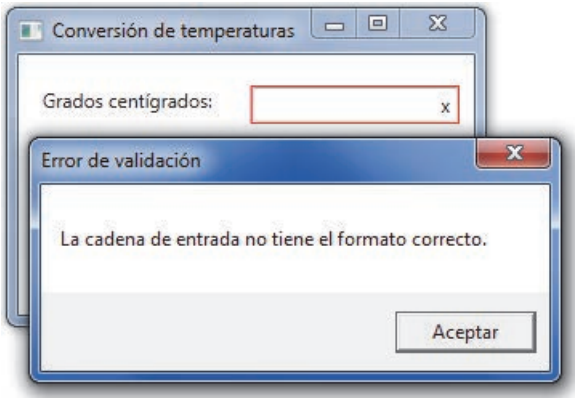
```
  If Validation.GetHasError(objTextBox) Then
    MessageBox.Show(e.Error.ErrorContent.ToString(), "Error de validación")
  End If
End Sub
```

También podemos añadir el controlador para el evento **Validation.Error** en el código XAML de la caja de texto:

```
<TextBox Name="ctGradosC" ...
         Validation.Error="ctGrados_ValidationError">
```

Observe que el controlador del evento **Error** verifica, por medio del método **Validation.GetHasError**, si el destino del enlace (caja de texto referenciada por el atributo *objTextBox*) tiene un error, en cuyo caso accede a la propiedad *e*.**Error**, que referencia a un objeto de tipo **ValidationError**, para obtener el mensaje que describe el error. Otra información que nos proporciona este objeto es **RuleInError**, el objeto **ValidationRule** correspondiente a la regla de validación (en nuestro caso **ExceptionValidationRule**) y **Exception**, excepción lanzada (en nuestro caso **FormatException**, información disponible sólo si especificamos la regla, o **Nothing** si no se lanzó ninguna excepción).

Si ahora ejecuta la aplicación, podrá comprobar que si la cadena introducida como grados centígrados no se corresponde con un valor **Double**, se mostrará un mensaje para notificar tal hecho al usuario.

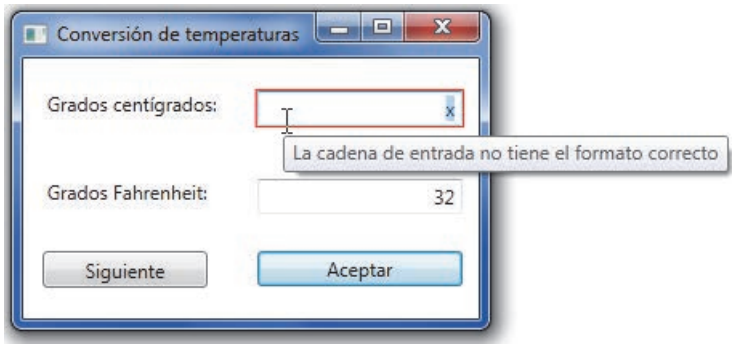


Con la otra caja de texto procederíamos de forma análoga, utilizando el mismo controlador:

```
<TextBox Name="ctGradosF" ...
         Validation.Error="ctGrados_ValidationError">
```

Si prefiere eliminar el diálogo **MessageBox** y mostrar un mensaje abreviado al pasar el ratón por encima de la caja de texto cuyo borde cambió a rojo debido a un error de entrada, modifique el método *ctGrados_ValidationError* así:

```
Private Sub ctGrados_ValidationError(sender As Object,
e As ValidationErrorEventArgs)
  If Validation.GetHasError(objTextBox) Then
    objTextBox.ToolTip = e.Error.ErrorContent.ToString()
  Else
    objTextBox.ToolTip = Nothing
  End If
End Sub
```



Una alternativa a lo anteriormente expuesto es sustituir el controlador anterior por el código XAML que se muestra a continuación:

```
<TextBox Name="ctGradosC" ...
     ToolTip="{Binding Path=(Validation.Errors)[0].ErrorContent,
                       ElementName=ctGradosC}">

<TextBox Name="ctGradosF" ...
     ToolTip="{Binding Path=(Validation.Errors)[0].ErrorContent,
                       ElementName=ctGradosF}">
```

Observe que ahora hemos vinculado la propiedad **ToolTip** de la caja de texto con la propiedad **ErrorContent** del objeto **ValidationError** que está en la posición 0 de la colección **Errors** de la propia caja de texto (**ElementName**), *ctGradosC* o *ctGradosF*. Esta manera de proceder nos permite eliminar el controlador del evento adjunto **Validation.Error**, por lo que no hay necesidad de especificar la propiedad **NotifyOnValidationError** de **Binding** con el valor **True**.

Cuando la validación produce un error, la propiedad **ErrorContent** proporciona la cadena que describe el error y si no produce un error, esta cadena es nula.

Resumiendo, el proceso de validación tiene lugar durante la transferencia de un valor del destino al origen del enlace, y antes de que se llame al conversor, en los términos que se describen a continuación:

1. Se vacía la colección **Validation.Errors** del elemento enlazado.

2. Se llama al método **Validate** en cada regla **ValidationRule** (el valor a validar viene dado por el parámetro *value* de este método, como veremos un poco más adelante) hasta que una de las reglas de validación detecte un error o hasta que se cumplan todas las reglas de validación.

3. Si una regla no pasa la validación, el motor de enlace crea un objeto **ValidationError** y lo añade a la colección **Validation.Errors** del elemento enlazado y además, si la propiedad **NotifyOnValidationError** del enlace vale **True**, el elemento genera el evento adjunto **Validation.Error**.

4. Si todas las reglas pasan la validación, el motor de enlace llama al conversor, si lo hay, y si éste pasa la validación se establece la propiedad en el origen.

5. Si se especificó la regla **ExceptionValidationRule** y se lanza una excepción durante el proceso de validación, el motor de enlace crea un objeto **ValidationError** y lo añade a la colección **Validation.Errors** del elemento enlazado, sólo si no se especificó en el enlace una propiedad **UpdateSourceExceptionFilter** con la intención de proporcionar un controlador personalizado que controle las excepciones.

Regla DataErrorValidationRule

Para utilizar esta regla para verificar la validez de los datos, el objeto origen del enlace tiene que implementar la interfaz **IDataErrorInfo**. La interfaz **IDataErrorInfo** proporciona dos propiedades: **Error** e **Item**. **Error** proporciona un mensaje de error que describe todo el objeto (por ejemplo, "datos no válidos") e **Item** acepta como argumento un nombre de propiedad y devuelve un mensaje de error relacionado con esa propiedad (por ejemplo, "ºC tiene que estar entre -273 y 1500"). Por ejemplo:

```
Class CGrados
  Implements INotifyPropertyChanged
  Implements IDataErrorInfo
  Private gradosC As Double = 0.0
  Private gradosF As Double = 32.0

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged
  ' ...
```

```
  Public ReadOnly Property [Error]()
  As String Implements IDataErrorInfo.Error
    Get
      Throw New System.NotImplementedException()
    End Get
  End Property

  Public ReadOnly Property Item(propiedad As String)
  As String Implements IDataErrorInfo.Item
    Get
      Dim mensaje As String = Nothing
      Select Case propiedad
        Case "GradosC"
          If gradosC < -273 OrElse gradosC > 1500 Then
            mensaje = "ºC tiene que estar entre -273 y 1500"
          End If
          Exit Select

        Case "GradosF"
          If gradosC < -459.4 OrElse gradosC > 2732 Then
            mensaje = "ºF tiene que estar entre -459.4 y 2732"
          End If
          Exit Select
      End Select
      Return mensaje
    End Get
  End Property
End Class
```

En el código anterior podemos observar la implementación de la interfaz **IDataErrorInfo** para establecer los límites de los valores de entrada de los grados centígrados y Fahrenheit (aunque también aparece la implementación de la interfaz **INotifyPropertyChanged**, ésta no es necesaria para utilizar **IDataErrorInfo**). El parámetro *propiedad* de *Item* tomará el nombre de la propiedad modificada del objeto *CGrados*.

La utilización de **IDataErrorInfo** tiene desventajas y es que el método *Item* normalmente incluye una sentencia **Select Case** con un caso para cada propiedad del objeto. Además, la funcionalidad de **IDataErrorInfo** se llama después de que el valor de la propiedad del objeto se ha establecido. Por lo tanto, si otros objetos también se han suscrito al evento **PropertyChanged** del objeto, ya habrán sido notificados del cambio, con lo que habrán comenzado a trabajar a partir de los datos que **IDataErrorInfo** está a punto de declarar como no válidos.

Esta regla se puede establecer implícitamente asignando el valor **True** a la propiedad **ValidatesOnDataErrors** del enlace, según se muestra a continuación:

```
<TextBox Grid.Column="1" Margin="20,20,10,0" Name="ctGradosC"
```

```
    ToolTip="{Binding
                Path=(Validation.Errors)[0].ErrorContent,
                ElementName=ctGradosC}"
    ... >
    <TextBox.Text>
      <Binding Path="GradosC" ConverterCulture="es-ES"
                UpdateSourceTrigger="LostFocus"
                ValidatesOnDataErrors="True" />
     </TextBox.Text>
</TextBox>
```

Ídem para la caja de texto *ctGradosF*.

Información del enlace

Según expusimos anteriormente, la propiedad **TextBox.Text** tiene como valor predeterminado de **UpdateSourceTrigger** *LostFocus*. Pero con este valor, en nuestro ejemplo, sólo se actualizaba la propiedad de *CGrados* vinculada con la caja de texto que perdía el foco. La solución fue cambiar el valor *Default* de **UpdateSourceTrigger** del enlace a *PropertyChanged*, ya que el hecho de introducir una cadena en una caja de texto y hacer la conversión hace que cambien las propiedades **Text** de las dos cajas de texto, por lo que también se actualizarán automáticamente las dos propiedades de *CGrados*. Pero este cambio, con el sistema de validación implementado, nos plantea el siguiente problema: cuando introducimos una cantidad negativa, al teclear el signo – se muestra el mensaje de error "la cadena de entrada no tiene el formato correcto" (con *PropertyChanged*, cada pulsación hace que se ponga en marcha el proceso de validación).

Una solución al problema planteado podría ser actualizar explícitamente el origen del enlace con los valores del destino en el momento de hacer clic en el botón *Aceptar*. Pues bien, para controlar cuándo se copian los valores de los destinos del enlace en el origen del enlace disponemos del método **UpdateSource** de la clase **BindingExpression**.

Un objeto **BindingExpression** proporciona información sobre un enlace definido por un objeto **Binding**. ¿Cómo obtenemos este objeto **BindingExpression**? Pues invocando al método **GetBindingExpression** heredado por el elemento de la clase **FrameworkElement**. Este método tiene un parámetro que se corresponde con la propiedad de dependencia en la que se establece el enlace.

Según lo expuesto, lo primero que haremos será cambiar el valor de **UpdateSourceTrigger** de los enlaces de las propiedades **Text** de las cajas de texto a *Explicit*, lo cual implica llamar al método **UpdateSource**, de lo contrario, los

cambios no se propagarán de nuevo al origen, y después modificar el método que responde al evento **Click** del botón *Aceptar*, como se puede ver a continuación:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  If objTextBox Is Nothing Then Return
  Dim be As BindingExpression =
    objTextBox.GetBindingExpression(TextBox.TextProperty)
  be.UpdateSource()
  If be.HasError Then Return

  Try
    Dim grados As Double
    ' Si se escribió en la caja de texto grados centígrados...
    If objTextBox Is ctGradosC Then
      grados = Convert.ToDouble(ctGradosC.Text) * 9.0 / 5.0 + 32.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosF.Text = String.Format("{0:F2}", grados)
      ctGradosF.GetBindingExpression(
                  TextBox.TextProperty).UpdateSource()
    End If
    ' Si se escribió en la caja de texto grados Fahrenheit...
    If objTextBox Is ctGradosF Then
      grados = (Convert.ToDouble(ctGradosF.Text) - 32.0) * 5.0 / 9.0
      ' Mostrar el resultado redondeado a dos decimales
      ctGradosC.Text = String.Format("{0:F2}", grados)
      ctGradosC.GetBindingExpression(
                  TextBox.TextProperty).UpdateSource()
    End If
  Catch exc As FormatException
    ctGradosC.Text = "0,00"
    ctGradosF.Text = "32,00"
  End Try
End Sub
```

Analizando el código del método anterior observamos que lo primero que se hace es obtener el objeto **BindingExpression** (objeto *be*) de la propiedad de dependencia (**TextBox.TextProperty**: identifica la propiedad de dependencia **Text**) en la que se ha establecido el enlace correspondiente a la caja de texto sobre la que el usuario ha escrito una cadena de caracteres y a continuación, invocando a **UpdateSource** se intenta actualizar la propiedad correspondiente en el origen del enlace, instante en el que se invoca al método **Valídate** para validar la cadena. Si la cadena no es válida, **Validate** generará el error correspondiente y la propiedad **HasError** del objeto *be* será establecida en **True**, condición que utilizamos para dar por terminado el proceso de conversión, permitiendo al usuario modificar su entrada. Si la cadena es válida, **UpdateSource** actualizará el origen del enlace (obsérvese que este método se invoca dos veces, una por cada caja).

Regla de validación personalizada

Para crear una regla de validación personalizada definiremos una clase derivada de la clase **ValidationRule** y después, redefiniremos su método **Validate**.

Por ejemplo, continuando con la aplicación anterior, supongamos que deseamos limitar la entrada de los grados centígrados entre los valores -273 y 1.500. Para ello, añada a la carpeta *Clases* del proyecto una nueva clase denominada *ReglaRangoGrados* derivada de **ValidationRule** que redefina el método **Validate**. Después, edite este método como se indica a continuación:

```
Class ReglaRangoGrados
  Inherits ValidationRule
  Private m_tempMin As Double
  Private m_tempMax As Double

  Public Property TempMin() As Double
    Get
      Return m_tempMin
    End Get
    Set(value As Double)
      m_tempMin = value
    End Set
  End Property

  Public Property TempMax() As Double
    Get
      Return m_tempMax
    End Get
    Set(value As Double)
      m_tempMax = value
    End Set
  End Property

  Public Overrides Function Validate(value As Object,
  cultureInfo As System.Globalization.CultureInfo) _
  As ValidationResult
    Dim grados As Double
    If Not Double.TryParse(value.ToString(), grados) Then
      Return New ValidationResult(False,
        "La cadena de entrada no tiene el formato correcto.")
    End If

    If grados < m_tempMin OrElse grados > m_tempMax Then
      Return New ValidationResult(False,
        String.Format("Valor fuera del rango {0} a {1}",
                      m_tempMin, m_tempMax))
    End If
```

```
    Return ValidationResult.ValidResult
  End Function
End Class
```

La clase *ReglaRangoGrados* proporciona dos propiedades públicas, que definen el rango de temperaturas válido, y el método **Validate**, que define dos condiciones para pasar la validación: una para verificar el formato del dato y otra para verificar si el dato está dentro del rango definido. Si el formato o el rango del dato *value* (dato a validar) no son los adecuados, se construye y se devuelve un objeto **ValidationResult**. El constructor de **ValidationResult** tiene un primer parámetro que indica si el dato es (**True**) o no (**False**) válido y un segundo parámetro que da información acerca de lo ocurrido. Si todas las reglas pasan, se devuelve el valor de la propiedad **ValidationResult.ValidResult** que equivale a un objeto **ValidationResult** construido con los parámetros **True** y **Nothing**.

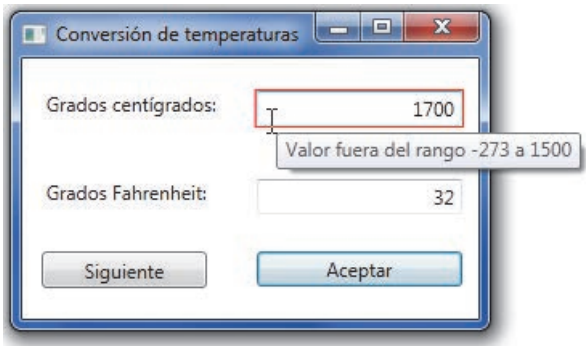
Un objeto de esta clase representa una regla de validación. Para aplicarla a cada una de las cajas de texto, tendremos que añadirla a la colección **ValidationRules** del objeto **Binding** establecido en su propiedad **Text**, según explicamos anteriormente:

```
<TextBox Name="ctGradosC" ... >
  <TextBox.Text>
    <Binding Path="GradosC" UpdateSourceTrigger="Explicit"
      ConverterCulture="es-ES" >
      <Binding.ValidationRules>
        <local:ReglaRangoGrados TempMin="-273" TempMax="1500" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>

<TextBox Name="ctGradosF" ... >
  <TextBox.Text>
    <Binding Path="GradosC" UpdateSourceTrigger="Explicit"
      ConverterCulture="es-ES" >
      <Binding.ValidationRules>
        <local:ReglaRangoGrados TempMin="-459.4" TempMax="2732" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Recuerde que el prefijo *local* hace referencia al espacio de nombres *ConverTemps* al que pertenece la clase *ReglaRangoGrados*.

Ahora, cuando el dato introducido por el usuario no se ajuste al formato y rango exigidos, después de pulsar el botón *Aceptar* la caja de texto correspondiente mostrará el borde rojo y una breve descripción indicando lo ocurrido.



Fuentes relativas

Anteriormente, para mostrar el mensaje de error ocurrido durante la entrada de datos, enlazamos la propiedad **ToolTip** de la caja de texto con la propiedad **ErrorContent** del objeto **ValidationError** así:

```
<TextBox Name="ctGradosC"...
     ToolTip="{Binding Path=(Validation.Errors)[0].ErrorContent,
                       ElementName=ctGradosC}">
```

Observe que el enlace vincula la propiedad **ToolTip** de la caja de texto, en este caso *ctGradosC*, con la propiedad **ErrorContent** del primer elemento de la colección **Errors** de la misma caja de texto. Esto es, la fuente con respecto al elemento *ctGradosC* es el mismo elemento, lo cual podríamos especificarlo también así:

```
<TextBox Name="ctGradosF"...
     ToolTip="{Binding Path=(Validation.Errors)[0].ErrorContent,
                       RelativeSource={RelativeSource Self}}">
```

La propiedad **RelativeSource** de **Binding** indica el origen del enlace especificando su ubicación con respecto a la posición del destino del enlace. Para más detalles, véase la clase **RelativeSource** en la ayuda proporcionada por MSDN.

ESTILOS Y PLANTILLAS

WPF proporciona un conjunto de características (estilos, plantillas, desencadenadores y guiones gráficos) para facilitar la personalización de la apariencia de una aplicación. Esto es, gracias al modelo de estilos y plantillas de WPF, los diseñado-

res pueden trabajar en la presentación usando XAML, y los desarrolladores pueden también hacerlo en la lógica de programación usando C# u otro lenguaje .NET.

Estilos

Un estilo, objeto de la clase **Style**, permite aplicar un conjunto de valores de propiedades a más de un elemento. La manera más común de declarar un estilo es como un recurso en la sección **Resources** de un elemento XAML. Por ejemplo:

```
<Window.Resources>
  <Style TargetType="TextBox" x:Key="EstTextBox">
    <Setter Property="Background" Value="Lightblue" />
  </Style>
</Window.Resources>
```

El estilo anterior define la propiedad **Background** para los elementos de tipo (**TargetType**) **TextBox**. Este recurso será identificado por la clave (**x:Key**) *EstTextBox*. Si no especificáramos la clave, el estilo se aplicaría automáticamente a todos los elementos del tipo especificado. Especificando la clave, el estilo sólo se aplicará a los elementos, del tipo especificado, que deseemos. Por ejemplo:

```
<TextBox Name="ctGradosC"
         Style="{StaticResource ResourceKey=EstTextBox}" ... >
</TextBox>
```

También podemos aplicar el estilo mediante programación así:

```
ctGradosC.Style = DirectCast(Me.Resources("EstTextBox"), Style)
```

Según lo expuesto, vemos que un estilo es un conjunto de objetos **Setter**, uno por cada propiedad que deseamos establecer del elemento especificado por **TargetType** al que se aplicará el estilo, cada uno de los cuales consta de, al menos, dos atributos: **Property** y **Value**; el primero es el nombre de la propiedad y el segundo, el valor que se va a aplicar a la propiedad.

Cuando se define un estilo sólo con **TargetType** (sin clave), ese estilo se aplicará automáticamente sólo a los elementos del tipo especificado y no a los tipos derivados. Por ejemplo, el siguiente estilo no se aplicará a los elementos de la clase **TextBox**.

```
<Style TargetType="Control">
  <Setter Property="Background" Value="Lightblue" />
</Style>
```

Podríamos omitir el atributo **TargetType** si especificamos explícitamente a qué clase pertenece la propiedad, según se muestra a continuación, pero esto no es una buena solución cuando son muchas las propiedades a establecer.

```
<Style x:Key="EstTextBox">
  <Setter Property="Control.Background" Value="Lightblue" />
</Style>
```

Ahora, utilizando el identificador *EstTextBox*, sí puede aplicar este estilo a un elemento de tipo **TextBox**, por ejemplo.

Una vez definido un estilo se puede aplicar a cualquier elemento que derive de **FrameworkElement** o **FrameworkContentElement**.

También podemos crear un estilo basado en otro especificado por el atributo **BasedOn** de **Style**. Por ejemplo:

```
<Style TargetType="TextBox" x:Key="EstTextBox2"
       BasedOn="{StaticResource EstTextBox}" >
  <Setter Property="ToolTip" Value="Error: valor no válido" />
</Style>
```

Cuando se utiliza la propiedad **BasedOn**, el nuevo estilo hereda los valores del estilo original, que podrían ser sobrescritos si fuera preciso, a los que se añaden otros nuevos. En el ejemplo anterior *EstTextBox2* hereda de *EstTextBox* el valor de **TextBox.Background** y añade un valor para **TextBox.ToolTip**.

Vincular controladores de eventos

Normalmente, cuando utilizamos estilos es para establecer valores específicos para determinadas propiedades de un tipo de objetos. No obstante, en un estilo también es posible que ese tipo de objetos especifique los controladores para uno o más eventos por medio de objetos **EventSetter**. De esta forma, el controlador de eventos que se especifique para responder a un determinado evento enrutado se aplicará a todos los elementos que hagan referencia al objeto **Style**. Por ejemplo, en lugar de que las cajas de texto de nuestra aplicación ejemplo especifiquen de forma independiente el controlador para los eventos **PreviewTextInput** y **GotKeyboardFocus**, podríamos incorporar esta especificación en el objeto **Style** que comparten ambas, según se indica a continuación:

```
<Style TargetType="TextBox" x:Key="EstTextBox">
  ...
  <EventSetter Event="PreviewTextInput"
               Handler="ctGradosF_PreviewTextInput" />
  <EventSetter Event="GotKeyboardFocus"
```

```
                Handler="ctGradosF_GotKeyboardFocus" />
</Style>
...
<TextBox Name="ctGradosF"
         Style="{StaticResource ResourceKey=EstTextBox}"...
```

No obstante, los objetos **EventSetter** no son la mejor elección para manipular eventos de propagación, ya que es más fácil manejarlos en un elemento de nivel superior (en nuestro caso, podríamos hacerlo en el **Grid**).

Desencadenadores

Un desencadenador establece propiedades o inicia acciones cuando cambia una propiedad o se genera un evento. **Style**, **ControlTemplate** y **DataTemplate** tienen una propiedad **Triggers** que puede contener un conjunto de desencadenadores **Trigger** o **EventTrigger**, entre otros. Por ejemplo:

```
<Window.Resources>
  <Style TargetType="TextBox" x:Key="EstTextBox">
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="True">
        <Setter Property="ToolTip" Value="Error: valor no válido"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

En el ejemplo anterior se muestra una clase **Style** con nombre, que está disponible para los controles **TextBox**, que define un elemento **Trigger** que establece la propiedad **ToolTip** de una caja de texto cuando la propiedad asociada **HasError** es **True** (se ha producido un error de validación).

Cuando se aplica un estilo a un elemento, se establecen incondicionalmente las propiedades de ese elemento especificadas por los elementos **Setter** del estilo. En cambio, utilizando un desencadenador las propiedades especificadas por los elementos **Setter** serán establecidas cuando se cumpla la condición especificada por el **Trigger** (en el ejemplo, cuando la propiedad **HasError** valga **True**); cuando la condición no se cumpla, las propiedades establecidas volverán a tomar sus valores anteriores.

Cuando el establecimiento de las propiedades o la iniciación de las acciones programadas en el desencadenador dependan de que se cumplan varias condiciones, utilizaremos un desencadenador de la clase **MultiTrigger**. Un elemento **MultiTrigger** tiene una colección denominada **Conditions** de objetos **Condition**, cada uno de los cuales define una condición del conjunto de condiciones que se

tienen que cumplir para que se desencadene el proceso programado. El ejemplo siguiente define un **MultiTrigger** que establece el valor de la propiedad **ToolTip** de una caja de texto cuando su propiedad **IsFocused** es **True** y su propiedad **Text** está vacía.

```
<Window.Resources>
  <Style TargetType="TextBox" x:Key="EstTextBox">
    <Style.Triggers>
      <MultiTrigger>
        <MultiTrigger.Conditions>
          <Condition Property="IsFocused" Value="True" />
          <Condition Property="Text" Value="" />
        </MultiTrigger.Conditions>
        <Setter Property="ToolTip" Value="Introducir datos"/>
      </MultiTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Los desencadenadores expuestos hasta ahora se activan cuando se cumple una condición establecida por una o más propiedades de un elemento visual, pero podríamos requerir que se activaran cuando se cumpla una condición establecida por una propiedad de un elemento no visual como es un objeto de datos vinculado. Para un caso como el planteado disponemos del desencadenador de datos **DataTrigger**, que permite aplicar valores de propiedad o realizar acciones cuando los datos enlazados cumplan la condición especificada. En el ejemplo siguiente se ha especificado un **DataTrigger** de modo que, si el valor de *GradosC* del elemento de datos *CGrados* es "0", se establece el valor de la propiedad **ToolTip** de una caja de texto.

```
<Window.Resources>
  <Style TargetType="TextBox" x:Key="EstTextBox">
    <Style.Triggers>
      <DataTrigger Binding="{Binding Path=GradosC}" Value="0">
        <Setter Property="ToolTip"
                Value="Punto de fusión y de congelación del agua"/>
      </DataTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Plantillas

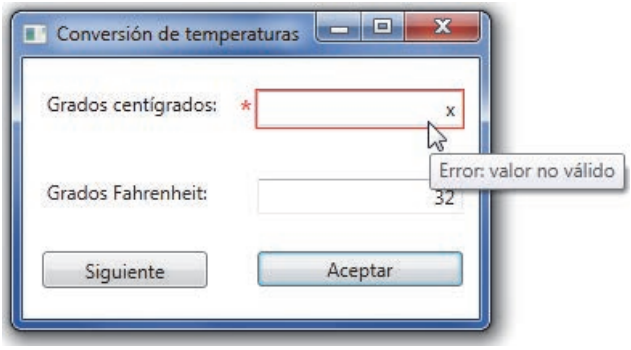
Para personalizar el aspecto de un control podemos proporcionarle una nueva plantilla **ControlTemplate** (en muchos casos, esta forma de proceder ofrece suficiente flexibilidad como para no tener que implementar controles personalizados)

y para definir cómo se presentarán los datos, WPF proporciona la plantilla **DataTemplate**.

Plantillas de control

Cuando se crea una plantilla **ControlTemplate** se combinan objetos **FrameworkElement** para compilar un único control. Su elemento raíz debe ser un objeto **FrameworkElement**, que normalmente contendrá otros objetos **FrameworkElement**.

En el ejemplo siguiente se crea una plantilla **ControlTemplate** personalizada que crea una nueva estructura visual para un control **TextBox**. En este ejemplo, la modificación de la apariencia del **TextBox** se aplica cuando durante el proceso de validación se produce un error, y consiste en añadir un panel con un control **TextBlock** para que muestre un asterisco (*) en rojo a la izquierda de la caja y un control **Border** en rojo que rodeará al control **TextBox** cuya información dio lugar al error, según muestra la figura siguiente. En realidad, esta plantilla es similar a la plantilla predeterminada, con la diferencia de que hemos añadido un asterisco.



Esto es, lo que se persigue con el código anterior es que cuando el usuario especifique en la caja de texto un valor no válido, la interfaz gráfica proporcione información del error, y la manera de hacerlo es estableciendo la propiedad asociada **Validation.ErrorTemplate** de la caja:

```
<Window.Resources>
  <Style TargetType="TextBox" x:Key="EstTextBox">
    ...

    <Setter Property="Validation.ErrorTemplate">
      <Setter.Value>
        <ControlTemplate>
          <WrapPanel>
            <TextBlock Text="*" FontSize="14pt"
                       Foreground="Red" Margin="0,0,2,0" />
```

```
            <Border BorderBrush="Red" BorderThickness="1" >
              <AdornedElementPlaceholder />
            </Border>
          </WrapPanel>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Para indicar la posición del control adornado, en nuestro caso del **TextBox**, se utiliza un elemento de la clase **AdornedElementPlaceholder** (marcador de posición del elemento adornado). Esto es, un elemento **AdornedElementPlaceholder** especifica dónde se colocará un control decorado con respecto a los otros elementos del **ControlTemplate**.

Observe que la plantilla creada es el valor que se asigna a la propiedad **ErrorTemplate**. Esto es, esta propiedad hace referencia al objeto **ControlTemplate** que se utilizará para generar la información de error de validación en la capa de adorno cuando durante el proceso de validación se produce un error. Recuerde que la plantilla predeterminada define un borde rojo alrededor del elemento.

Para poner en práctica lo explicado, puede aplicarlo a la última versión que hicimos de la aplicación "Conversión de temperaturas".

```
<Window.Resources>
  <local:ConversorDoubleBrush x:Key="converGradosCaBrush" />
  <Style TargetType="TextBox" x:Key="EstTextBox">
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="True">
        <Setter Property="ToolTip"
                Value="{Binding
                          Path=(Validation.Errors)[0].ErrorContent,
                          RelativeSource={RelativeSource Self}}" />
      </Trigger>
    </Style.Triggers>
    <Setter Property="Validation.ErrorTemplate">
      <Setter.Value>
        <ControlTemplate>
          <WrapPanel>
            <TextBlock Text="*" FontSize="14pt"
                       Foreground="Red" Margin="0,0,2,0" />
            <Border BorderBrush="Red" BorderThickness="1" >
              <AdornedElementPlaceholder />
            </Border>
          </WrapPanel>
        </ControlTemplate>
      </Setter.Value>
```

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

Observe que el estilo *EstTextBox* que acabamos de crear asigna un valor a la propiedad **ToolTip** de la caja de texto, por lo tanto no puede especificar simultáneamente esta misma propiedad en el código que define dicha caja.

Para terminar, aplique el estilo a las cajas de texto de la aplicación. Por ejemplo:

```
<TextBox Name="ctGradosC"
         Style="{StaticResource ResourceKey=EstTextBox}" ...

<TextBox Name="ctGradosF"
         Style="{StaticResource ResourceKey=EstTextBox}" ...
```

Plantillas de datos

Finalmente, apuntar que también es posible describir cómo debe mostrar un control, por ejemplo un **ListBox**, cada elemento de datos utilizando una plantilla **DataTemplate**. Por ejemplo:

```
<ListBox Name="list1" Margin="10"
      ItemsSource="{Binding Source={StaticResource fuenteDatos}}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding Path=dato1}" />
        <TextBlock Text="{Binding Path=dato2}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

El código anterior asigna un objeto **DataTemplate** a la propiedad **ItemTemplate** del control **ListBox**, que define cómo se mostrarán los datos. Como se puede observar, el objeto **DataTemplate** define los controles de datos adecuados al tipo de los datos a mostrar; por ejemplo, para mostrar los datos *dato1* y *dato2*, en este caso, hemos utilizado dos elementos **TextBlock** que hemos colocado dentro de un objeto **StackPanel** y después, hemos enlazado cada uno de ellos con la propiedad correspondiente del origen de datos. Tenga en cuenta que cada objeto **DataTemplate** se aplica a cada uno de los elementos **ListBoxItem** del control lista. En los próximos capítulos explicaremos esto con más detalle.

RECURSOS

En WPF, cualquier objeto puede ser un recurso: un dibujo o forma, un estilo, un pincel o un color utilizado en varias partes de una interfaz gráfica pueden ser definidos como un recurso. El ejemplo siguiente define dos recursos: un objeto **ImageBrush** que pinta un área con una imagen y una elipse.

```
<Window.Resources>
  <ImageBrush x:Key="PincelMosaico" TileMode="Tile"
              ViewportUnits="Absolute" Viewport="0 0 24 24"
              ImageSource="imagenes/info_24x24.png" Opacity="0.3">
  </ImageBrush>
  <Ellipse x:Key="elipse" Width="48" Height="24" />
  ...
</Window.Resources>
```

Según lo expuesto en los apartados anteriores, los estilos y las plantillas nos permiten tomar el control de los aspectos visuales de nuestra aplicación, pero para ser aplicados con facilidad allí donde se necesiten fueron definidos como recursos WPF. ¿Dónde? Pues lo hicimos en la sección de recursos de la ventana porque dichos recursos iban a ser utilizados por los elementos contenidos en la misma, pero también podíamos haberlos definido en la sección de recursos de la aplicación. No obstante, los recursos pueden ser definidos para cualquier elemento del marco de trabajo en su sección de recursos, aunque normalmente se definen en el elemento raíz.

Lo anteriormente expuesto nos puede conducir a preguntarnos si cada elemento **FrameworkElement** o **FrameworkContentElement** tiene una propiedad **Resources** que permita definir los recursos relacionados con dicho elemento. La respuesta es sí; cada elemento del marco de trabajo tiene una propiedad **Resources** que hace referencia a un objeto **ResourceDictionary** que define el diccionario de recursos local para ese elemento.

```
<ResourceDictionary>
  Recursos
</ResourceDictionary>
```

La recuperación individual de uno u otro recurso incluido en ese diccionario exige que cada uno de ellos tenga una clave única que permita acceder al mismo, la definida por el atributo **x:Key**; normalmente es una cadena, aunque podría ser de otro tipo; esto no significa que todos los recursos deban tener un atributo **x:Key** explícito; varios tipos de objetos (por ejemplo **Style**), cuando se definen como un recurso admiten una clave implícita deducida de alguna forma. En el siguiente ejemplo, el elemento **Window** utiliza su diccionario de recursos local para almacenar los recursos *PincelMosaico*, *converGradosCaBrush* y *EstTextBox*:

```
<Window.Resources>
  <ImageBrush x:Key="PincelMosaico" TileMode="Tile"
              ViewportUnits="Absolute" Viewport="0 0 24 24"
              ImageSource="imagenes/info_24x24.png" Opacity="0.3">
  </ImageBrush>
  <local:ConversorDoubleBrush x:Key="converGradosCaBrush" />
  <Style TargetType="TextBox" x:Key="EstTextBox">
    ...
  </Style>
</Window.Resources>
```

Según lo expuesto, un diccionario de recursos es un objeto de la clase **ResourceDictionary** cuya finalidad es almacenar un conjunto de objetos accesibles individualmente mediante clave que actúan como recursos para los elementos que componen la aplicación.

Una vez definido un recurso, se puede hacer referencia al mismo como un recurso estático, utilizando la extensión de marcado **StaticResource**, o como un recurso dinámico, utilizando la extensión de marcado **DynamicResource**.

```
<TextBox Name="ctGradosC"
         Style="{StaticResource ResourceKey=EstTextBox}" ...

<Button Name="btAceptar" ...
        Background="{StaticResource PincelMosaico}"/>
```

Una extensión de marcado procesa la cadena del atributo **ResourceKey** y devuelve el objeto correspondiente a un cargador XAML. La extensión de marcado **StaticResource** procesa la clave durante la carga de la aplicación buscando el valor asociado con la misma para asignárselo a la propiedad que hace uso de dicha extensión, y la extensión de marcado **DynamicResource**, por el contrario, procesa la clave creando una expresión, la cual permanece sin evaluar hasta que se necesite durante la ejecución de la aplicación, momento en el cual se evalúa y proporciona el valor. Los recursos dinámicos son adecuados cuando el valor del recurso depende de condiciones que no se conocen hasta la ejecución.

Aunque la propiedad **Resources** hace referencia sólo al diccionario de recursos declarado directamente dentro de ese elemento, en el proceso de búsqueda de recursos, un elemento secundario puede tener acceso a cualquiera de los recursos definidos en cada elemento primario; esto es, la búsqueda se hace de forma recursiva y ascendente, hasta llegar a los recursos de la aplicación y finalmente a los recursos del sistema.

En el ejemplo anterior, cuando el cargador XAML procesa el valor *{StaticResource ResourceKey=EstTextBox }* para la propiedad **Style** de **TextBox**, la lógica de búsqueda de recursos comprueba primero la existencia de *EstTextBox* en el

diccionario de recursos del elemento **TextBox**. Si **TextBox** no tiene una definición de la clave *EstTextBox* (no la tiene, porque su colección de recursos está vacía), la búsqueda continúa en el elemento primario de **TextBox**, que es el **Grid**, y así sucesivamente. Así, cuando se define un recurso en el elemento raíz **Window**, todos los elementos del árbol lógico del objeto **Window** pueden tener acceso a él y se puede reutilizar el mismo recurso para establecer el valor de cualquier propiedad de un elemento compatible con el recurso.

El objeto **ResourceDictionary** subyacente, por tratarse de una colección, admite los métodos comunes para agregar, quitar o consultar recursos desde la colección mediante código. Así mismo, la colección de recursos de un elemento puede ser reemplazada asignando a la propiedad **Resources** otro objeto **ResourceDictionary**.

Recursos creados mediante código

De lo anterior se deduce que es posible crear los recursos de una aplicación mediante código. Para ello, cree una colección **ResourceDictionary** y, a continuación, agregue todos los recursos a dicha colección invocando al método **Add**. Después, utilice la colección creada para establecer la propiedad **Resources** de un elemento de la aplicación o de la propia aplicación.

```
Dim recursos As New ResourceDictionary()
Me.Resources = recursos
recursos.Add("objGrados", New CGrados())
```

En la práctica, normalmente no creará su propia colección de recursos, sino que utilizará la proporcionada por el elemento WPF. Por ejemplo:

```
Me.Resources.Add("objGrados", New CGrados())
```

Para recuperar un recurso concreto desde el código invoque al método **FindResource** o **TryFindResource** desde el lugar de la aplicación donde el recurso sea necesario. La diferencia de comportamiento entre estos dos métodos es que **FindResource** produce una excepción si no se encuentra la clave solicitada y **TryFindResource** no producirá una excepción, sino que devolverá **Nothing**.

```
Dim objGrados As CGrados = TryCast(Me.FindResource("objGrados"), CGrados)
```

¿Por qué utilizar **FindResource** y no el indizador del diccionario? Porque **FindResource** no abandona la búsqueda si el recurso no se encuentra en el diccionario del elemento especificado, sino que continúa la búsqueda en el elemento padre del árbol lógico hasta llegar a la raíz, después busca en la aplicación, en los temas y, finalmente, en los recursos del sistema.

Recursos del sistema

WPF utiliza los recursos del sistema para definir pinceles, fuentes y métricas que el usuario puede configurar a nivel del sistema. Las claves para estos recursos se proporcionan como propiedades estáticas de las clases **SystemColors**, **SystemFonts** y **SystemParameters**. Por ejemplo, el siguiente código fija el color de fondo para el elemento *gridVentPpal*:

```
Me.gridVentPpal.Background =
  TryCast(Me.FindResource(SystemColors.ControlBrushKey), Brush)
```

La clave del recurso (en el ejemplo, el objeto devuelto por **ControlBrushKey**) se proporciona como un objeto de la clase **ResourceKey** en lugar de como una cadena; los recursos del sistema siempre se identifican por un objeto específico para evitar ambigüedades entre ellos y los recursos con nombre propios de la aplicación.

La forma seguida en el ejemplo anterior para asignar a una propiedad de un elemento el valor de un recurso está bien para algunos tipos de recursos, pero no para el del ejemplo, ya que si el usuario cambia el esquema de colores del sistema, la propiedad **Background** de *gridVentPpal* no se actualizará en el mismo sentido. Para solucionar este problema proceda como se indica a continuación:

```
Me.gridVentPpal.SetResourceReference( _
  Grid.BackgroundProperty, SystemColors.ControlBrushKey)
```

Ahora, si el valor del recurso del sistema cambia, la propiedad **Background** del **Grid** cambiará en el mismo sentido, porque el método **SetResourceReference** lo que hace es buscar el recurso especificado y establecer una referencia al mismo para la propiedad especificada.

Igualmente desde XAML. Por otra parte, si sabe que el recurso no va a cambiar nunca durante la ejecución, utilice la extensión de marcado **StaticResource**, en otro caso utilice **DynamicResource**. Por ejemplo:

```
<Grid Name="gridVentPpal"
      Background="{DynamicResource
                   {x:Static SystemColors.ControlBrushKey}}">
```

Si en lugar de la sintaxis anterior hubiéramos empleado la que se expone a continuación, aun siendo sintácticamente correcta no funcionará:

```
<Grid Name="gridVentPpal"
      Background="{DynamicResource
                    ResourceKey=SystemColors.ControlBrushKey}">
```

¿Por qué no funciona? Porque *SystemColors.ControlBrushKey* se interpreta como la cadena que nombra el recurso y no como el objeto devuelto por la propiedad estática **ControlBrushKey**, de ahí que tengamos que indicar explícitamente que se trata de una propiedad estática:

```
{x:Static SystemColors.ControlBrushKey}
```

Ahora, el código XAML anterior, hace referencia a la propiedad estática **ControlBrushKey**.

TEMAS Y MÁSCARAS

Desde una perspectiva visual, un tema define la apariencia global de un sistema operativo (por ejemplo, Windows) y de las aplicaciones que se ejecutan en su seno (Windows incluye varios temas), mientras que una máscara es un tema personalizado y específico de cada aplicación (una aplicación puede incluir varias máscaras).

Según hemos visto en el apartado anterior sobre recursos, una aplicación WPF puede tener varios recursos definidos en distintos lugares de la misma. Colectivamente, este conjunto de recursos puede considerarse el tema para la aplicación, lo cual pone de manifiesto que tanto los temas como las máscaras de WPF se pueden definir con suma facilidad mediante los diccionarios de recursos.

Estos diccionarios de recursos pueden estar embebidos en el propio código o pueden proporcionarse por medio de un ensamblado separado, como sucede con el ensamblado que contiene los controles de WPF, *PresentationFramework.dll* (observe el nodo *References* de su aplicación).

También podemos definir los diccionarios de recursos como archivos individuales con extensión *.xaml*. Esta forma de proceder permite reutilizar un tema en varias aplicaciones o crear temas/máscaras intercambiables definiendo varios diccionarios de recursos que proporcionen los mismos tipos de recursos pero con diferentes valores. Redefinir estos estilos u otros recursos en el nivel de aplicación es el enfoque recomendado para aplicar una máscara a una aplicación.

Por ejemplo, haciendo uso de los diccionarios de recursos vamos a modificar la última versión de la aplicación conversión de temperaturas para que el color de fondo de la ventana varíe en función de la temperatura mostrada. Para ello añadiremos varias máscaras, por ejemplo tres: una que muestre un fondo azul degradado para temperaturas inferiores a 10 ºC, otra que muestre un verde degradado para temperaturas entre 10 y 30 ºC y otra que muestre un amarillo degradado para temperaturas mayores de 30 ºC.

Según lo expuesto, recuperamos de nuevo la última versión de la aplicación y añadimos al proyecto una nueva carpeta denominada *Skins* para almacenar en la misma las distintas máscaras. A continuación, para añadir una máscara, haga clic con el botón secundario del ratón sobre esa carpeta y elija la opción *Agregar > Diccionario de recursos*. Denomine al diccionario *azul.xaml* y haga clic en *Agregar*. Se le mostrará una ventana de edición que le permitirá editar el fichero XAML *azul.xaml* que inicialmente almacena un diccionario de recursos vacío:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

</ResourceDictionary>
```

Después, complete la máscara como se indica a continuación:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="pincelAzulDegradado"
                         StartPoint="0,0" EndPoint="0,1">
      <GradientStop Color="LightBlue" Offset="0"/>
      <GradientStop Color="Blue"  Offset="1"/>
    </LinearGradientBrush>
    <Style TargetType="{x:Type Grid}">
      <Setter Property="Background"
              Value="{StaticResource pincelAzulDegradado}"/>
    </Style>
</ResourceDictionary>
```

A continuación, siguiendo un proceso análogo, añada otros dos diccionarios de recursos, *verde.xaml* y *amarillo.xaml*, en los que, con respecto al diccionario anterior, cambie simplemente el color.

Para utilizar estos diccionarios de recursos tendremos que combinarlos con el diccionario de recursos principal del elemento apropiado de la aplicación o con el diccionario de recursos en el nivel de aplicación. Esto se hace añadiéndolos a la colección genérica referenciada por la propiedad **MergedDictionaries** del **ResourceDictionary** principal. Por ejemplo:

```
<Application x:Class="ConverTemps.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
```

```
        <ResourceDictionary Source="Skins/amarillo.xaml"/>
        <ResourceDictionary Source="Skins/verde.xaml"/>
        <ResourceDictionary Source="Skins/azul.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Obsérvese que cada objeto **ResourceDictionary** de la colección **MergedDictionaries** utiliza el atributo **Source** para especificar el identificador de recursos uniforme (URI) que indica la ubicación del archivo de recursos XAML que se va a combinar. Según vimos anteriormente, el elemento raíz de este archivo XAML tiene que ser de tipo **ResourceDictionary**.

En la colección de diccionarios combinados, **MergedDictionaries**, son válidas las claves duplicadas. La búsqueda comienza por el diccionario principal y continúa en la colección de diccionarios combinados, y dentro de la colección **MergedDictionaries** la búsqueda se inicia por el último **ResourceDictionary** agregado; la búsqueda se detiene en cuanto se encuentra una clave solicitada. Según esto, la búsqueda en la colección **MergedDictionaries** del ejemplo anterior se realizará primero en *azul.xaml*.

El objeto **ResourceDictionary** predeterminado, inicialmente vacío, que existe para cualquier propiedad **Resources** también tiene una propiedad **MergedDictionaries** predeterminada: una colección genérica inicialmente vacía.

Los diccionarios combinados se pueden agregar también mediante código. Para ello, se obtiene una referencia al objeto **ResourceDictionary** primario deseado, se obtiene el valor de su propiedad **MergedDictionaries** y se invoca al método **Add** de la colección genérica **MergedDictionaries**. Por ejemplo:

```
Imports System.Collections.ObjectModel

Private Sub formPrincipal_Loaded1(sender As Object,
e As RoutedEventArgs)
  ' Temas/máscaras
  Dim dic_combinados As Collection(Of ResourceDictionary) =
    Application.Current.Resources.MergedDictionaries

  ' Asegurarse de que sólo se cargarán una vez
  If dic_combinados.Count = 0 Then
    Dim rd1 As New ResourceDictionary()
    Dim rd2 As New ResourceDictionary()
    Dim rd3 As New ResourceDictionary()
    Try
      rd1.Source = New Uri("Skins/amarillo.xaml", UriKind.Relative)
      rd2.Source = New Uri("Skins/verde.xaml", UriKind.Relative)
```

```
      rd3.Source = New Uri("Skins/azul.xaml", UriKind.Relative)
      dic_combinados.Add(rd1)
      dic_combinados.Add(rd2)
      dic_combinados.Add(rd3)
    Catch exc As Exception
      MessageBox.Show(exc.Message)
    End Try
  End If
End Sub
```

Una vez cargados los recursos, la aplicación los utiliza de forma automática actualizando todos los controles afectados. En nuestro ejemplo, observaremos que el color de la rejilla cambia a azul, recurso proporcionado por el último diccionario cargado.

Continuando con el problema planteado, en nuestro caso sólo necesitaríamos cargar el diccionario de recursos *azul.xaml*, ya que inicialmente la temperatura es 0º C:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Skins/azul.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

e intercambiar ese diccionario con alguno de los diccionarios de recursos que creamos con el fin de que proporcionaran los mismos tipos de recursos pero con diferentes valores. Para ello, añada a la clase *MainWindow* un método *EstablecerMascara* y edítelo como se indica a continuación:

```
Private Sub EstablecerMascara(gradosC As Double)
  Dim skin As String = ""
  If objGrados.GradosC < 10 Then
    skin = "Skins/azul.xaml"
  ElseIf objGrados.GradosC > 30 Then
    skin = "Skins/amarillo.xaml"
  Else
    skin = "Skins/verde.xaml"
  End If

  Dim rd As New ResourceDictionary()
  rd.Source = New Uri(skin, UriKind.Relative)
  ' Cambiar el último diccionario de recursos
  Dim n As Integer = Application.Current.Resources.MergedDictionaries.Count
  Application.Current.Resources.MergedDictionaries(n - 1) = rd
End Sub
```

Este método crea un diccionario de recursos *azul*, *amarillo* o *verde* en función de la temperatura, que después utilizamos para sustituir al que está en la última posición de la colección **MergedDictionaries**, posición por donde comienza la búsqueda (en nuestro caso la colección tiene un solo elemento).

Finalmente, invoque a este método desde el método *btAceptar_Click* y pruebe los resultados que se obtienen (proceda de forma análoga desde *btSiguiente_Click*).

```
Private Sub btAceptar_Click(sender As [Object], e As RoutedEventArgs)
  ' ...
  EstablecerMascara(Convert.ToDouble(ctGradosC.Text))
End Sub
```

Si los recursos fueran nombrados como se indica a continuación:

```
<Style TargetType="{x:Type Grid}" x:Key="azul">
  <Setter Property="Background"
          Value="{StaticResource pincelAzulDegradado}"/>
</Style>
```

una vez que hayan sido cargados desde el nivel de la aplicación, podríamos proceder localizando el recurso *azul*, *amarillo* o *verde* en función de la temperatura y aplicándolo a la propiedad **Style** del elemento correspondiente, en nuestro caso de la rejilla:

```
Private Sub EstablecerMascara(gradosC As Double)
  If objGrados.GradosC < 10 Then
    gridVentPpal.Style = TryCast(Me.FindResource("azul"), Style)
  ElseIf objGrados.GradosC > 30 Then
    gridVentPpal.Style = TryCast(Me.FindResource("amarillo"), Style)
  Else
    gridVentPpal.Style = TryCast(Me.FindResource("verde"), Style)
  End If
End Sub
```

RESUMEN

En este capítulo se ha hecho una introducción a la jerarquía de clases de la biblioteca WPF y se ha desarrollado una aplicación para explicar cómo utilizar los controles de uso más frecuente como son las etiquetas, las cajas de texto y los botones de pulsación. Se ha expuesto también cómo establecer la tecla de acceso y el botón predeterminado, así como el significado y uso de las propiedades más comunes de los controles.

Después se han descrito los eventos enrutados y cómo interceptarlos; concretamente nos hemos detenido en los eventos del teclado y en los eventos relacionados con el foco y el ratón, y aplicando lo estudiado se ha explicado cómo interceptar la tecla pulsada.

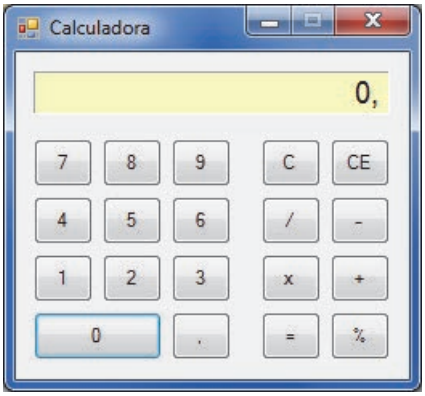
A continuación se ha expuesto cómo validar un campo de texto, primero sin utilizar el motor de enlace de datos de WPF y después, utilizándolo. Utilizar el motor de enlace de datos nos condujo a estudiar, además de la interfaz **INotifyPropertyChanged**, la clase **Binding** para poder crear un enlace entre un origen y un destino de los datos, a entender lo que es un contexto de datos, a utilizar conversores, a obtener información del enlace y a implementar reglas de validación.

También se ha expuesto cómo crear una apariencia personalizada para una aplicación utilizando estilos, plantillas y desencadenadores.

Así mismo, aprendimos sobre los recursos WPF, estilos y plantillas entre otros, y vimos que son almacenados en diccionarios de recursos para posteriormente hacer referencia a los mismos de una forma sencilla desde cualquier parte de nuestra aplicación. Colectivamente, este conjunto de recursos se considera el tema/máscara de la aplicación que los define. También vimos cómo definir los diccionarios de recursos como archivos individuales con extensión *.xaml* para combinarlos con el diccionario principal, ya que esta forma de proceder permite reutilizar un tema/máscara en varias aplicaciones o crear temas/máscaras intercambiables en una misma aplicación.

EJERCICIOS PROPUESTOS

Realizar el diseño de una calculadora como la de la figura siguiente.



CAPÍTULO 3

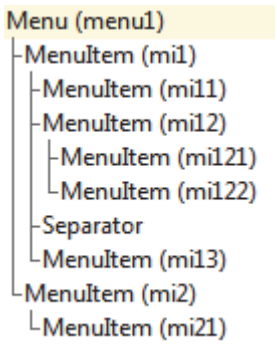
© F.J.Ceballos/RA-MA

MENÚS Y BARRAS DE HERRAMIENTAS

Muchas aplicaciones simples tienen un formulario y varios componentes, pero no cabe duda de que su aspecto puede ser mejorado utilizando menús y barras de herramientas. Ambos componentes son muy familiares para los usuarios de interfaces gráficas; de ahí el interés por introducirlos en nuestras aplicaciones. Por eso dedicaremos este capítulo a explicar cómo crear y utilizar menús en una aplicación, así como barras de herramientas.

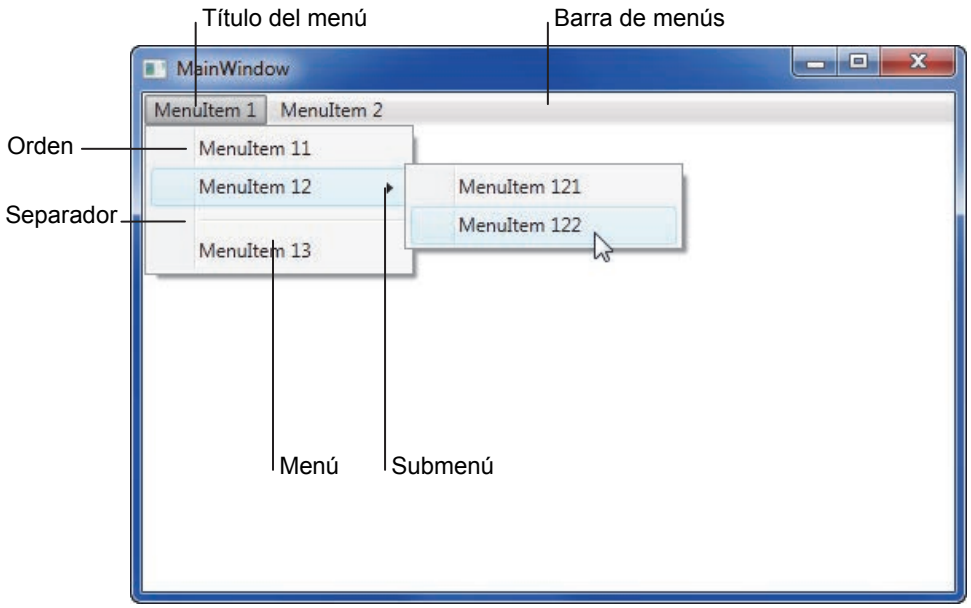
ARQUITECTURA DE UNA BARRA DE MENÚS

Para construir una barra de menús, WPF proporciona básicamente los elementos **Menu**, **MenuItem** y **Separator**, que podemos combinar según muestra la estructura jerárquica en árbol expuesta a continuación. Concretamente, esta estructura jerárquica representa una barra de menús, *menu1*, con dos menús, *mi1* y *mi2*, con sus respectivos elementos:



Un elemento **Menu** es un contenedor de elementos **MenuItem** y **Separator**. A su vez, cada elemento **MenuItem** puede contener otros elementos **MenuItem** y **Separator**, y así sucesivamente, formando una estructura jerárquica en árbol.

Según esta jerarquía, el elemento **Menú** formaría lo que generalmente denominamos barra de menús, la cual contendrá uno o más menús. El primer nivel de elementos **MenuItem** daría lugar a los menús dentro de esta barra, el segundo nivel daría lugar a los elementos de cada menú, el tercer nivel a los elementos de los submenús correspondientes y así sucesivamente.



Las clases que definen los elementos descritos pertenecen al espacio de nombres **System.Windows.Controls**.

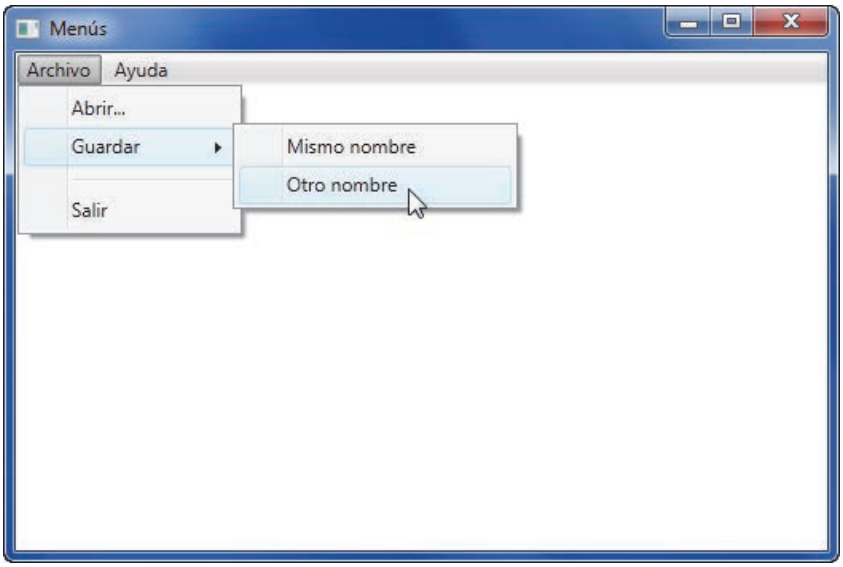
DISEÑO DE UNA BARRA DE MENÚS

Un menú es una forma de proveer al usuario de un conjunto de órdenes, lógicamente relacionadas, agrupadas bajo un mismo título. El conjunto de todos los títulos correspondientes a los menús diseñados aparecerá en la barra de menús situada normalmente debajo del título del formulario, pero se puede colocar en cualquier otra parte.

Cuando el usuario haga clic en un título de un menú, se desplegará una lista visualizando los elementos que contiene el menú. Los elementos de un menú pueden ser *órdenes*, *submenús* y *separadores*. Cuando se hace clic en una *orden* o se

selecciona y se pulsa *Entrar*, se ejecuta una acción o se despliega una caja de diálogo. Por convenio, una orden seguida de tres puntos (...) indica que se desplegará una caja de diálogo. Cuando se selecciona un *submenú*, se despliega una nueva lista de elementos. Un *separador* tiene como finalidad separar grupos de órdenes de un mismo menú, lógicamente en función de su actividad.

Para añadir una barra de menús, básicamente disponemos de las clases **Menu**, **MenuItem** y **Separator** que soportan la barra de menús, los menús de la barra y los elementos de los menús, y los separadores, respectivamente. Dicho de otra forma, **Menu** es un contenedor para un menú de un formulario, al que se le pueden añadir objetos **MenuItem** (menús de la barra, submenús o los elementos de los menús) y **Separator** (separadores).



Crear una barra de menús

Para crear una barra de menús, los pasos a ejecutar son los siguientes:

1. *Arrastrar desde la caja de herramientas un control* **Menu**. Normalmente, lo colocaremos en la parte superior de la ventana utilizando un **DockPanel** o colocándolo en la fila 0 de un **Grid**, extendiéndolo a todo lo ancho de la ventana (valor por omisión de **HorizontalAlignment**). En el siguiente ejemplo se muestra una barra de menús colocada en la fila 0 de un **Grid**:

```
<Window x:Class="MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Menús" Height="350" Width="525">
```

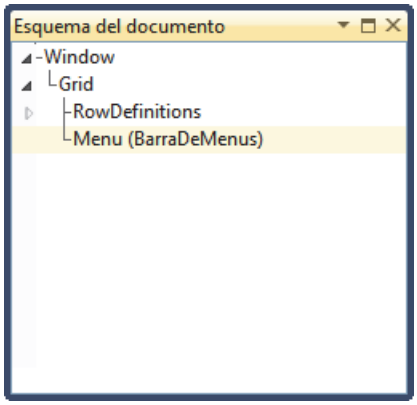
```
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>

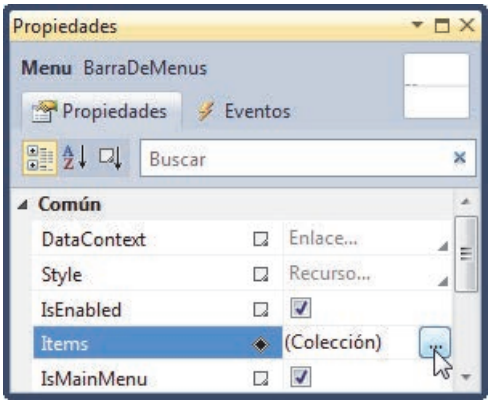
    <Menu Grid.Row="0" Name="BarraDeMenus" >
      ...
    </Menu>
  </Grid>
</Window>
```

2. *Introducir los elementos que componen la barra de menús*. Añada a la colección referenciada por **Items** del elemento **Menu** un elemento **MenuItem** por cada menú de la barra, asigne a su propiedad **Name** el nombre utilizado en el código para referirse a dicho elemento y a su propiedad **Header** el título que mostrará la barra.

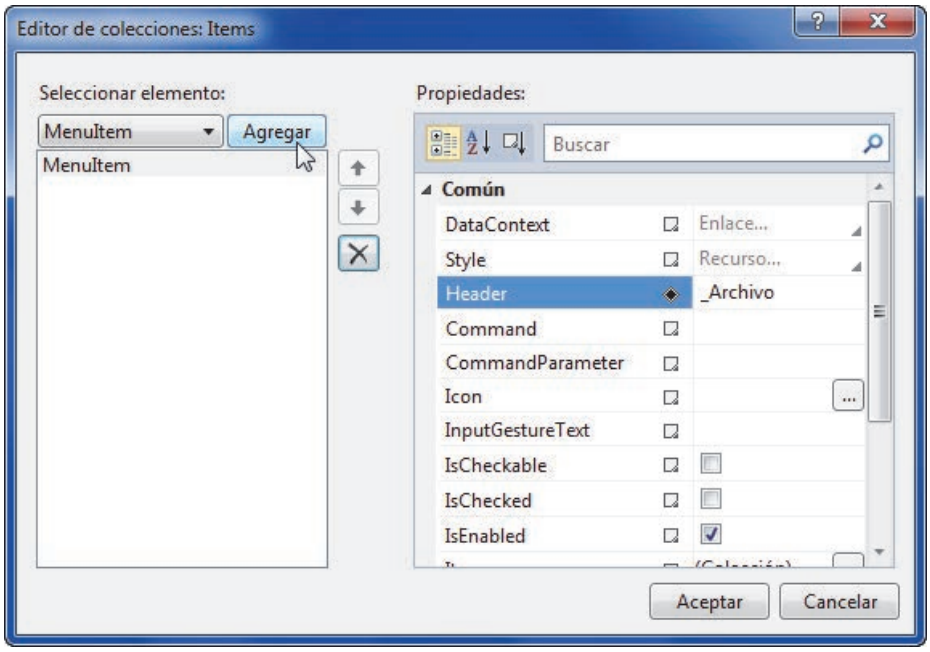
   La propiedad **Items** de **Menu** (o de **MenuItem** cuando se trate de un menú de la barra o de un submenú de un menú) hace referencia a la colección de objetos del elemento en cuestión.

   Por ejemplo, para añadir a la barra de menús los menús *Archivo* y *Ayuda* que muestra la figura anterior, diríjase al diseñador de Visual Studio, muestre el esquema del documento y seleccione el elemento *BarraDeMenus*. Después, diríjase a la ventana de propiedades del elemento seleccionado, seleccione su propiedad **Items** y abra la colección de elementos (inicialmente vacía).





   Se mostrará el editor de colecciones. Haga clic en el botón *Agregar* para añadir un nuevo objeto **MenuItem**, establezca sus propiedades y, cuando haya finalizado, desde la ventana de propiedades, asígnele un identificador para referirnos al mismo desde el código.



Inserte, cuando lo considere necesario, un *guión bajo* (_) antes de la letra que dará acceso directo al menú, para que aparezca subrayada. De esta forma, el usuario podrá seleccionar este menú, además de con el ratón, pulsando la tecla *Alt* más la tecla correspondiente a la letra que aparece subrayada.

Repita la acción anterior para cada uno de los menús que desea añadir a la barra de menús. Estas acciones añadirán en *MainWindow.xaml* el código XAML necesario para la construcción de los menús:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>

  <Menu Grid.Row="0" Name="BarraDeMenus"
        VerticalAlignment="Top" >
    <MenuItem Name="menuArchivo" Header="_Archivo">

    </MenuItem>
    <MenuItem Name="menuAyuda" Header="Ay_uda">

    </MenuItem>
  </Menu>
</Grid>
```

3. *Introducir los elementos que componen el menú*. Añada a la colección **Items** de cada menú un elemento **MenuItem** por cada orden/submenú del mismo, asigne a su propiedad **Name** el nombre utilizado en el código para referirse a dicho elemento y a su propiedad **Header** el título que se mostrará cuando se abra el menú. Por ejemplo, observe en el código XAML siguiente los elementos de los menús *Archivo* y *Ayuda* que muestra la ventana *Menús* anterior.

```
<MenuItem Name="menuArchivo" Header="_Archivo">
  <MenuItem Name="ArchivoAbrir" Header="A_brir..." />
  <MenuItem Name="ArchivoGuardar" Header="_Guardar">
    <MenuItem Name="ArchivoGuardarMismoNombre"
              Header="_Mismo nombre" />
    <MenuItem Name="ArchivoGuardarOtroNombre"
              Header="_Otro nombre" />
  </MenuItem>
  <Separator />
  <MenuItem Name="ArchivoSalir" Header="_Salir" />
</MenuItem>

<MenuItem Name="menuAyuda" Header="Ay_uda">
  <MenuItem Name="AyudaAcercaDe" Header="_Acerca de..." />
</MenuItem>
```

Inserte, para los elementos que considere, un *guión bajo* (_) antes de la letra que dará acceso directo al elemento del menú.

4. *Crear un submenú*. Un elemento puede ser una orden, o bien un submenú si a su vez despliega una lista de elementos. En ambos casos se trata de un objeto de la clase **MenuItem**. Observe en el código anterior el submenú *Guardar*.

5. *Añadir un separador*. Utilizando separadores, objetos de la clase **Separator**, puede agrupar visualmente las órdenes en función de su actividad.

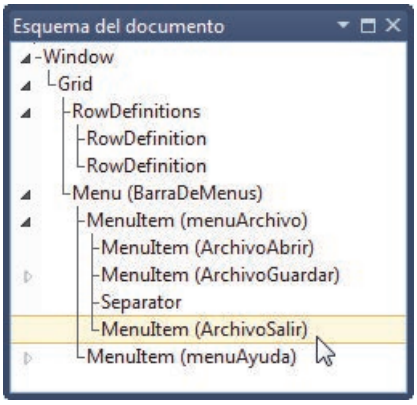
A modo de ejemplo, siguiendo los pasos descritos anteriormente, cree el esqueleto para una nueva aplicación, denominada *Menus* (véase la figura anterior), que utilice un objeto **Window** como ventana principal a la que asignaremos el título "Menús". El proceso de crear el esqueleto de una aplicación ya fue explicado en los capítulos anteriores, por lo que no nos detendremos más en estas cuestiones. Después, añada el código descrito anteriormente. Esta aplicación se encuentra en la carpeta *Cap3\Menus* del CD.

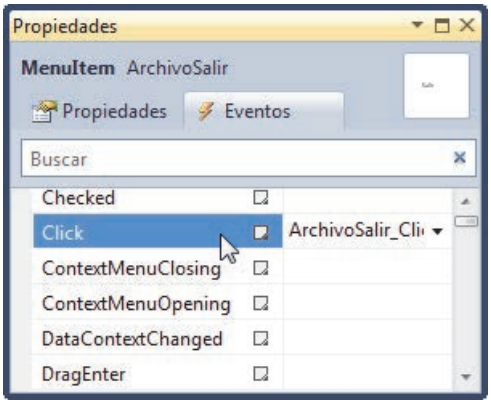
Controlador de un elemento de un menú

En general, los elementos que componen cada menú no estarán operativos hasta que añadamos el código que responda a la acción de seleccionarlo para su ejecu-

ción, acción que el usuario de la aplicación puede llevar a cabo haciendo clic sobre el elemento utilizando el ratón, seleccionándolo con las teclas de movimiento del cursor y pulsando la tecla *Entrar* o utilizando las teclas de acceso directo.

Pues bien, cuando se selecciona un elemento de un menú para su ejecución se genera el evento **Click**. Tendremos entonces que añadir a la ventana un controlador para este evento y completar el método definido por dicho controlador con el código que se debe ejecutar. Por ejemplo, para suscribir la ventana *MainWindow* de la aplicación *Menus* con el evento **Click** de la orden *Salir* del menú *Archivo*, diríjase al diseñador, muestre el esquema del documento y seleccione el elemento *ArchivoSalir* del menú. Después, diríjase a la ventana de propiedades del elemento seleccionado, haga clic en la pestaña *Eventos*, seleccione el evento **Click** y haga doble clic sobre el mismo.





La acción anterior añadirá a la ventana *MainWindow* el código siguiente:

```
<!-- En el fichero XAML -->
<MenuItem Name="ArchivoSalir" Header="_Salir"
          Click="ArchivoSalir_Click"/>

' En el fichero de código subyacente
Private Sub ArchivoSalir_Click(sender As Object, e As RoutedEventArgs)
End Sub
```

A continuación, complete el método añadido como se muestra a continuación:

```
Private Sub ArchivoSalir_Click(sender As Object, e As RoutedEventArgs)
  ' Cerrar la ventana principal;
  Close()
End Sub
```

Cuando se cierra (**Close**) una ventana como la anterior, todos los recursos creados con ese objeto se cierran y se elimina la ventana.

Para el resto de las órdenes procederemos de igual forma. De esta manera, cada orden tendrá asociado su propio controlador de eventos. Como ejercicio, puede añadir estos controladores para que simplemente muestren, en modo depuración, un mensaje indicando qué orden se ha ejecutado. Por ejemplo:

```
Private Sub ArchivoAbrir_Click(sender As Object, e As RoutedEventArgs)
  System.Diagnostics.Debug.WriteLine("Archivo > Abrir...")
End Sub
```

El método **WriteLine** de la clase **Debug** sólo será tenido en cuenta si ejecuta la aplicación en modo depuración (*F5*). Otra forma de mostrar un mensaje al usuario es utilizando el método **Show** de **MessageBox**. La ejecución de este método mostrará una ventana con el mensaje especificado:

```
Private Sub ArchivoAbrir_Click(sender As Object, e As RoutedEventArgs)
  MessageBox.Show("Archivo > Abrir")
End Sub
```

Aceleradores y nemónicos

Sabemos que el nemónico de un menú o de alguno de sus elementos proporciona acceso directo desde el teclado. Un nemónico se corresponde con el carácter que aparece subrayado en el texto del control. Sirva de ejemplo el menú *Archivo* de la aplicación *Menus* o la orden *Salir* de este menú. El acceso desde el teclado se consigue pulsando las teclas *Alt*+nemónico. Por ejemplo, cuando el menú *Archivo* de la aplicación *Menus* está desplegado, pulsar las teclas *Alt+s* es equivalente a hacer clic en el elemento *Salir* de este menú. Si el menú no está abierto (desplegado), no se podrá acceder a sus elementos a través de sus nemónicos.

Los aceleradores son muy similares a los nemónicos, excepto en que se puede especificar cualquier combinación de teclas de la forma:

```
[{Ctrl|Shift|Alt}]+{cualquier tecla}
```

(*Shift* = *Máyus* = mayúsculas) y actúan independientemente de que el menú esté abierto o no. Por ejemplo, si asociamos el acelerador *Ctrl+A* con la orden *Abrir* del menú *Archivo* de la aplicación *Menus*, siempre que se pulse esa combinación de teclas se ejecutará la orden *Abrir*, independientemente de que el menú *Archivo* esté o no abierto.

Como ejemplo, vamos a añadir a la aplicación *Menus* el código que asocie el acelerador *Ctrl+A* con la orden *Abrir* (identificada por el objeto *ArchivoAbrir*) del menú *Archivo*. Para ello, diríjase al diseñador de formularios, seleccione la orden

Abrir en el esquema del documento, diríjase a la ventana de propiedades, seleccione la propiedad **InputGestureText** y asígnele la cadena *Ctrl+A*:

```
<MenuItem Name="ArchivoAbrir" Header="A_brir..."
          InputGestureText="Ctrl+A" />
```

Esta propiedad no asocia la entrada del usuario, en nuestro caso *Ctrl+A*, con el elemento del menú; simplemente agrega ese texto a la cabecera del elemento. Quiere esto decir que la aplicación debe administrar la entrada del usuario para llevar a cabo la acción. ¿Cómo? Utilizando *órdenes enrutadas* (**RoutedCommand**), cuestión que estudiamos a continuación.

ÓRDENES ENRUTADAS

Una *orden enrutada* es una acción que la aplicación lleva a cabo a petición del usuario. El sistema de órdenes enrutadas de WPF se basa en las clases **RoutedCommand** y **RoutedEvent**.

Los eventos estudiados hasta ahora ofrecen un mecanismo para enlazar elementos WPF de la interfaz de usuario a controladores, realizando emparejamientos "elemento origen del evento – controlador", que requiere escribir código que se repite en muchas ocasiones en la aplicación; por ejemplo, cuando se trata de habilitar o deshabilitar elementos de la interfaz de usuario porque en ese instante no tiene sentido llevar a cabo la acción programada para esos elementos. Bajo este mecanismo, el origen de la acción y la lógica que ejecuta esa acción están fuertemente acoplados. Este mecanismo, en muchas ocasiones, podrá simplificarse utilizando *órdenes enrutadas*.

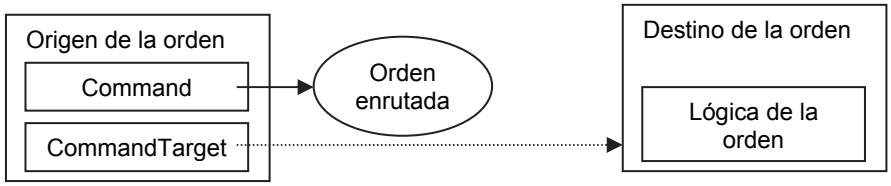
Lo que hace diferente a una *orden enrutada* de un controlador de eventos vinculado a un control, como un **Button** o un **Timer**, es que la *orden enrutada* separa la semántica y el origen de la acción de la lógica que ejecuta esa acción. Por ejemplo, piense en la orden *Copiar* (*Ctrl+C*); semánticamente no hay diferencias entre distintas aplicaciones, incluso el origen puede ser el mismo, una orden de un menú, por ejemplo, pero la lógica de la acción es específica del objeto destino sobre el que se actúa (no es lo mismo copiar un texto de un **TextBox** que una imagen de un **Image**). Incluso, el destino de una *orden enrutada* puede informar al origen de si esa acción está disponible, para si no lo está, que no se lleve a cabo; por ejemplo, la acción de *Copiar* tiene sentido sólo cuando algo está seleccionado.

El patrón en el que se basa una *orden enrutada* no es nuevo: se crea un objeto que sabe cómo hacer una tarea y, entonces, toda la aplicación se apoya en ese objeto para hacer esa tarea.

Vincular una orden enrutada con un control

Genéricamente, una *orden enrutada* es un objeto que implementa la interfaz **ICommand**. No obstante, para facilitar la creación de *órdenes enrutadas*, WPF proporciona la clase **RoutedCommand** que es la implementación WPF de **ICommand**. Desde esta definición, podemos decir que una *orden enrutada* es un objeto de la clase **RoutedCommand** o de una clase derivada.

WPF define un conjunto de *órdenes enrutadas* muy comunes, listas para ser utilizadas, distribuido en varias clases: **MediaCommands**, **ApplicationCommands**, **NavigationCommands**, **ComponentCommands** y **EditingCommands**. Estas clases definen solamente las *órdenes enrutadas*, esto es, los objetos **RoutedCommand**, no la lógica de implementación de esas órdenes, de la cual es responsable el objeto destino de la orden. Por ejemplo, los objetos **Cut**, **Copy** y **Paste** están definidos en la clase **ApplicationCommands** como propiedades **Shared** públicas, pero para el caso en el que el destino de estas órdenes sea un objeto **TextBox**, es este objeto el que proporciona la lógica de estas órdenes (así fue diseñado en WPF) y lo mismo diríamos para otros objetos destino de esas órdenes. Como estudiaremos con detalle a continuación, para ejecutar una de estas órdenes habrá que vincularla con el elemento que la tiene que invocar para su ejecución, por ejemplo un botón o una orden de un menú, y con el elemento sobre el que se va a ejecutar la acción, por ejemplo una caja de texto.



El elemento que ejecuta la orden hace referencia a la misma por medio de su propiedad **Command** y al destino sobre el que se ejecutará la acción por medio de su propiedad **CommandTarget**, que por omisión coincide con el elemento que tiene el foco del teclado:

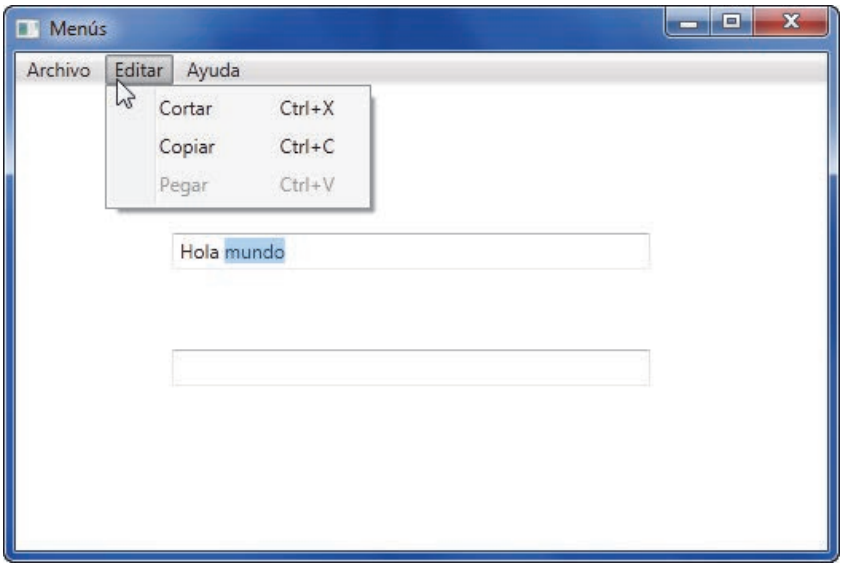
```
<Button Content="Copiar" Name="btCopiar"
        Command="ApplicationCommands.Copy"
        CommandTarget="{Binding ElementName=textBox1}"
        />
```

Como ejercicio, puede añadir a la aplicación *Menus* un menú *Editar* con los elementos *Cortar*, *Copiar* y *Pegar*; después, asígneles las órdenes **Cut**, **Copy** y **Paste**, respectivamente, según muestra el código XAML siguiente:

```
<MenuItem Name="menuEditar" Header="_Editar">
  <MenuItem Name="EditarCortar" Header="Cotar" Command="Cut" />
```

```
  <MenuItem Name="EditarCopiar" Header="Copiar" Command="Copy" />
  <MenuItem Name="EditarPegar" Header="Pegar" Command="Paste" />
</MenuItem>
```

Añada también dos cajas de texto, según muestra la figura siguiente:



Ejecute la aplicación, sitúe el punto de inserción en una de las cajas de texto, escriba algún texto, seleccione texto y haga clic en el menú *Editar*. Los elementos del menú *Cortar* y *Copiar* aparecen habilitados y *Pegar*, deshabilitado; lógico, hay texto seleccionado que se puede cortar o copiar, pero el portapapeles está vacío y no hay texto para pegar.

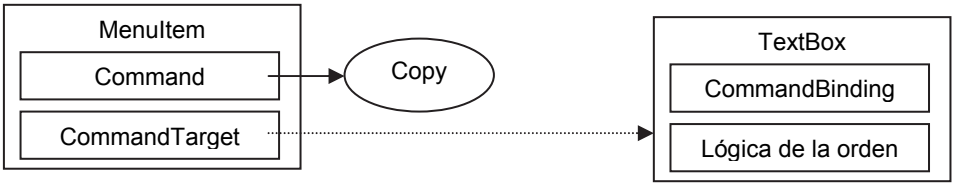
Analicemos el porqué de ese comportamiento. Pero antes, como comparativa, pensemos cómo hubiéramos procedido utilizando controladores de eventos.

Para poder conectar el origen de una de esas órdenes con el destino de la misma utilizando eventos, sería necesario definir un controlador para el evento **Click** de cada uno de los elementos del menú y ese código necesitaría hacer referencia a las dos cajas de texto, tendría que determinar qué caja de texto tiene el foco y llamar a operaciones apropiadas del portapapeles según la selección del texto en el control. También tendría que preocuparse de habilitar y deshabilitar los elementos del menú según el estado de las cajas de texto y del portapapeles. El resultado es un código muy acoplado.

Más aun, ¿qué ocurriría si esas cajas de texto estuvieran ahora en un control de usuario o en un control personalizado y el código subyacente de la ventana no tuviera acceso directo a ellos? En ese caso el control de usuario tendría que expo-

ner una interfaz programática para posibilitar el enlace de cosas desde el contenedor, o bien exponer las cajas de texto públicamente, y ninguno de estos dos métodos es el ideal.

Con *órdenes enrutadas*, lo único que hay que hacer es establecer la propiedad **Command** de los elementos del menú con los objetos correspondientes de la clase **ApplicationCommands** de WPF y todo listo. Ahora, después de seleccionar texto en una de las cajas de texto, podrá cortarlo o copiarlo, pasando automáticamente al portapapeles, con lo que podría pegarlo en la otra caja. Incluso, funcionaría para cualquier caja de texto de cualquier lugar de la interfaz gráfica. ¿Por qué? Lo que sucede aquí es que la implementación de la clase **TextBox** tiene un enlace integrado con cada orden enrutada **Cut**, **Copy** y **Paste** y encapsula automáticamente el control del portapapeles para esas órdenes. Este enlace es un objeto de la clase **CommandBinding**.



La mayoría de las órdenes que WPF pone a nuestra disposición no son objetos **RoutedCommand**, sino objetos de su clase derivada **RoutedUICommand**. La diferencia entre una clase y otra es que **RoutedUICommand** incluye una propiedad **Text** que hace referencia al texto que aparece en la cabecera de un elemento de menú o en la descripción de un botón de la barra de herramientas, por ejemplo. La ventaja de definir el texto de la orden en ella misma (en lugar de hacerlo directamente en el control) es que facilita que la localización quede resuelta. Para comprobar lo expuesto, pruebe a escribir las órdenes del menú *Editar* sin especificar su atributo **Header**:

```
<MenuItem Name="menuEditar" Header="_Editar">
  <MenuItem Name="EditarCortar" Command="Cut" />
  <MenuItem Name="EditarCopiar" Command="Copy" />
  <MenuItem Name="EditarPegar" Command="Paste" />
</MenuItem>
```

Modelo de una orden enrutada

Llegados a este punto, podemos resumir lo expuesto diciendo que el modelo de una orden enrutada se puede descomponer en cuatro conceptos básicos:

1. El *origen de la orden* que es el objeto que invoca a la misma; por ejemplo, el elemento *Copiar* del menú *Editar*. Son ejemplos de orígenes de órdenes las clases **MenuItem**, **Button** y **KeyGesture**.

2. La *orden* o acción que se va a ejecutar; por ejemplo **Copy**. Normalmente se trata de un objeto **RoutedCommand**.

3. El *enlace de la orden* que es el objeto **CommandBinding** que asigna la lógica (el código que hay que ejecutar) a la orden.

4. El *destino de la orden* que es el objeto en el que se ejecuta la misma; por ejemplo, una caja de texto.

Los orígenes de órdenes en WPF, generalmente, implementan la interfaz **ICommandSource** que expone tres propiedades: **Command**, **CommandTarget** y **CommandParameter**. Por ejemplo:

```
<MenuItem Name="EditarCopiar" Header="Copiar"
          Command="Copy"
          CommandTarget="{Binding ElementName=textBox1}"
          />
```

Command es la orden a ejecutar cuando sea invocada por el origen, **CommandTarget** es el destino de la orden: objeto en el que se ejecuta la orden y **CommandParameter** es un tipo de datos definido por el usuario utilizado para pasar información a los controladores que implementan la orden.

Si no se establece **CommandTarget**, el destino de la orden será el elemento que tenga el foco del teclado (es lo que sucedió en nuestro ejemplo).

```
<MenuItem Name="EditarCopiar" Header="Copiar"
          Command="Copy"
          />
```

Esto es así porque cuando un elemento contenedor es un ámbito de foco permite realizar un seguimiento de quién tiene el foco (realiza un seguimiento de la propiedad asociada **FocusManager.FocusedElement**). Un elemento es un ámbito de foco cuando el valor de la propiedad asociada **FocusManager.IsFocusScope** es **True**; por ejemplo, las clases **Window**, **Menu**, **ContextMenu** y **ToolBar** son ámbitos de foco. **FocusManager** proporciona un conjunto de métodos compartidos, propiedades asociadas y eventos que nos permiten determinar y establecer los ámbitos de foco, así como establecer el elemento enfocado dentro del ámbito.

Una de las ventajas de utilizar como destino el elemento que tenga el foco del teclado es que así podemos utilizar el mismo origen para invocar una orden en varios destinos, sin tener que hacer un seguimiento del destino de la orden.

Al principio del apartado anterior vinculamos la orden **Copy** con un botón así:

```
<Button Content="Copiar" Name="btCopiar"
        Command="ApplicationCommands.Copy"
        CommandTarget="{Binding ElementName=textBox1}"
        />
```

¿Por qué especificamos el atributo **CommandTarget**? Porque **Button** no es un ámbito de foco. Para saber si un elemento es un ámbito de foco invoque al método **FocusManager.GetIsFocusScope**. Por ejemplo:

```
Dim esAmbitoDeFoco As Boolean = FocusManager.GetIsFocusScope(btCopiar)
```

La línea de código anterior devolverá un valor **False**, porque el botón *btCopiar* no es un ámbito de foco. ¿Qué hacemos para que lo sea con el fin de poderlo utilizar con diferentes cajas de texto? Pues asociar el botón con la propiedad **FocusManager.IsFocusScope** establecida a **True**:

```
<Button Content="Copiar" Name="btCopiar"
        FocusManager.IsFocusScope="True"
        Command="ApplicationCommands.Copy"
        />
```

Ahora, podrá utilizar este botón para ejecutar la orden **Copy** sobre el elemento que tenga el foco.

Cómo se ejecuta una orden enrutada

Siguiendo con el ejemplo, ¿cómo llega el mensaje a la caja de texto para decirle que controle la orden, y cómo se invoca la orden seleccionada (**Cut**, **Copy** o **Paste**) en la caja de texto enfocada? Aquí es donde entra en juego el enrutamiento de órdenes: utilizar eventos enrutados para llevar a cabo una orden.

Las clases WPF que implementan **ICommandSource** son **ButtonBase**, **MenuItem**, **Hyperlink** e **InputBinding**. Los objetos derivados de las tres primeras clases invocan una orden cuando se hace clic en ellos y un objeto **InputBinding** invoca una orden cuando se realiza la entrada **InputGesture** asociada, como veremos más adelante.

Una orden **RoutedCommand** se ejecuta en el objeto destino de la orden. Antes de ejecutarse, la lógica que implementa la orden tiene que verificar que ésta se puede ejecutar; si no se puede, el origen de la orden se inhabilitará y si se puede, estará habilitado y cuando sea invocada, se ejecutará.

El acceso a la lógica real que determina si una orden puede ejecutarse en el destino actual es proporcionado por el método **CanExecute** de la orden enrutada (objeto **RoutedCommand**). Este método no incluye la lógica, sino que genera los eventos **PreviewCanExecute** y **CanExecute** que recorren el árbol de elementos, siguiendo la ruta que hay entre la raíz y el elemento sobre el que se ejecuta la orden, en busca de un elemento que incluya un objeto **CommandBinding** que vincule la orden con la lógica de implementación de la misma. El acceso a la lógica real que ejecuta la orden es análogo; en este caso, es el método **Executed** de la orden enrutada el que genera los eventos **PreviewExecuted** y **Executed**, que recorren el árbol de elementos en busca del objeto **CommandBinding**. Estos eventos se generan en el elemento especificado por **CommandTarget** o, por omisión, en el elemento que tiene el foco de teclado.

Un objeto **CommandBinding** asocia una orden enrutada con los controladores de eventos que implementan dicha orden. Este objeto puede ser suministrado por el objeto destino de la orden enrutada, como sucede en el caso del control **TextBox**, o por cualquier otro elemento entre la raíz del árbol de elementos que componen la interfaz gráfica y el elemento destino de la orden. La estructura de este objeto, para el ejemplo de la caja de texto que estamos analizando, tiene un aspecto análogo siguiente:

```
<TextBox... >
  <TextBox.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Copy"
                    CanExecute="CanExecuteHandler"
                    Executed="CopyCommandHandler" />
  </TextBox.CommandBindings>
</TextBox>
```

donde la propiedad **Command** hace referencia a la orden a ejecutar, el evento **CanExecute** hace referencia al controlador que da acceso a la lógica real que determina si una orden puede ejecutarse en el destino actual y el evento **Executed** hace referencia al controlador que da acceso a la lógica real que ejecuta la orden. Obsérvese que la lógica de implementación de la orden se vincula a la orden a través de la colección **CommandBindings**.

Según lo expuesto, si el destino actual de la orden enrutada tiene un **CommandBinding** para dicha orden, se llama a los controladores especificados y si no lo tiene, los eventos a los que hemos hecho referencia anteriormente se tuneli-

zan/propagan a través del árbol de elementos hasta encontrar un elemento que tenga un **CommandBinding**.

En la aplicación ejemplo que hemos implementado, lo que sucede es esto: debido a que las órdenes **Cut**, **Copy** y **Paste** están asociadas con elementos de un menú, a **CanExecute** y **Execute** los controla el objeto **TextBox** que tiene el foco. Si las cajas de texto estuvieran contenidas en un control de usuario, entonces tendríamos la oportunidad de establecer el enlace de la orden en la ventana, en el control de usuario que contiene el **Grid**, en el **Grid** que contiene las cajas de texto o en las cajas de texto individuales. La caja de texto que tenga el foco determinará el final de la ruta de búsqueda del enlace iniciada en la raíz. El siguiente ejemplo muestra un enlace establecido en una ventana (objeto **Window**):

```
<Window.CommandBindings>
  <CommandBinding Command="{x:Static local:MainWindow.OrdenPersonalizada}"
                  Executed="ExecutedOrdenPersonalizada"
                  CanExecute="CanExecuteOrdenPersonalizada" />
</Window.CommandBindings>
```

Una cosa importante es saber que una vez que se invoca a un controlador para una orden determinada, ya no se llamará a ningún otro controlador. Así que, si en el ejemplo expuesto en el párrafo anterior, el control de usuario administra el método **CanExecute**, ya no se llamará a la implementación **CanExecute** de **TextBox**.

La interfaz **ICommand**, además de los métodos **CanExecute** y **Execute**, tiene un evento **CanExecuteChanged**. Como hemos dicho, **CanExecute** determina si el estado de la aplicación es, actualmente, compatible con la ejecución de la orden que se quiere ejecutar; si este método devuelve **True**, entonces el método **Execute** puede ser invocado, que es el que indirectamente lleva a cabo la ejecución de la orden; ahora bien, si devuelve **False**, entonces todos los controles que a través de su propiedad **Command** hacen referencia a esa orden son deshabilitados para que el usuario no pueda intentar ejecutarla en ese momento.

Lógicamente, se espera que el origen de la orden llame a **CanExecute** para determinar si habilitar o deshabilitar cualquier código de invocación de la orden asociada. En el objeto **RoutedCommand** se generará **CanExecuteChanged** de acuerdo con los cambios de estado o de foco en la interfaz gráfica de usuario. Normalmente, el origen de la orden escuchará este evento y consultará a **RoutedCommand** a través del método **CanExecute**.

Resumiendo, las órdenes enrutadas de WPF ofrecen un mecanismo específico para enlazar controles de la interfaz de usuario, como los botones de una barra de herramientas o los elementos de un menú, a controladores de una forma más sim-

plificada que cuando utilizamos el control normal de eventos. Resaltamos algunas cosas importantes de este mecanismo con respecto al control normal de eventos:

- Los elementos origen de las órdenes enrutadas pueden estar desconectados de los controladores que realmente implementan la orden, no es necesario que haya referencias directas entre ellos, como ocurriría si estuvieran vinculados por un controlador de eventos.

- Las órdenes enrutadas habilitarán o deshabilitarán todos los controles de la interfaz de usuario asociados con ellas cuando el controlador indique que la orden no se puede ejecutar.

- Las órdenes enrutadas permiten asociar métodos abreviados de teclado y otras formas de entrada (lápiz óptico, por ejemplo) como otra forma de invocar una orden.

- Las órdenes enrutadas pueden presentar problemas cuando tratemos con interfaces de usuario complejas, donde la lógica del control de las órdenes está en el código de soporte para las definiciones de vista y los orígenes de las órdenes no siempre están dentro de una barra de herramientas o de un menú. Por ejemplo, piense que la habilitación y la lógica de control para una orden puede no formar parte directamente del árbol visual, simplemente porque pueden estar situadas en un presentador o en un modelo de presentación, lo que haría que la orden enrutada no funcionara. Para evitar el problema comentado y otros, necesitaremos simplificar las cosas, asegurándonos de que los controladores de las órdenes están en el mismo elemento destino de la orden o en niveles superiores del árbol visual.

Órdenes enrutadas personalizadas

Si las órdenes enrutadas de las clases de la biblioteca WPF no satisfacen nuestras necesidades, podemos crear nuestras propias órdenes personalizadas. La forma más común de hacer esto consiste en crear un objeto **RoutedCommand Shared** y **Public**.

Como ejemplo, vamos añadir una orden enrutada personalizada para que se ejecute al hacer clic en la orden *Abrir* del menú *Archivo* de nuestra aplicación. Para ello, añada a la clase *MainWindow* un atributo *RCAbrir* que sea un objeto de la clase **RoutedUICommand** (observe que *RCAbrir* es **Shared** y **Public**):

```
Public Shared RCAbrir As New RoutedUICommand()
```

A continuación añada los controladores para los eventos **Execute** y **CanExecute** del objeto **CommandBinding**. Estos controladores, según se muestra a continuación, implementarán la lógica de la orden *RCAbrir*:

```
Private Sub ExecutedRCAbrir(sender As Object,
e As ExecutedRoutedEventArgs)
  ' Implemente aquí la lógica de la orden RCAbrir
  MessageBox.Show("Orden Archivo > Abrir... personalizada ejecutada")
End Sub

Private Sub CanExecuteRCAbrir(sender As Object,
e As CanExecuteRoutedEventArgs)
  ' Retorna True sólo si el destino de la orden es un control
  Dim destino As Control = TryCast(e.Source, Control)

  If destino IsNot Nothing Then
    e.CanExecute = True
  Else
    e.CanExecute = False
  End If
End Sub
```

Ya tenemos implementada la orden enrutada *RCAbrir*. El paso siguiente consiste en crear un objeto **CommandBinding** que vincule la orden *RCAbrir* con los controladores *ExecutedRCAbrir* y *CanExecuteRCAbrir* que proporcionan la lógica para dicha orden. Para una mayor flexibilidad, añadiremos el objeto de enlace de la orden a la colección de enlaces de órdenes de la ventana (objeto **Window**):

```
<Window x:Class="MainWindow"
        ...
        xmlns:local="clr-namespace:Menus"
        Title="Menús" Height="350" Width="525">
  <Window.CommandBindings>
    <CommandBinding Command="{x:Static local:MainWindow.RCAbrir}"
                    Executed="ExecutedRCAbrir"
                    CanExecute="CanExecuteRCAbrir" />
  </Window.CommandBindings>
  ...
</Window>
```

Este paso también lo podríamos ejecutar desde el código subyacente así:

```
Dim cb As New CommandBinding(RCAbrir,
                             AddressOf ExecutedRCAbrir,
                             AddressOf CanExecuteRCAbrir)
Me.CommandBindings.Add(cb)
```

Finalmente, invocamos a la orden *RCAbrir* desde el elemento *Abrir* del menú *Archivo*:

```
<MenuItem Name="ArchivoAbrir" Header="A_brir..."
          Command="{x:Static local:MainWindow.RCAbrir}" />
```

Ejecute ahora la aplicación y observe los resultados.

Aceleradores de teclado

Anteriormente, en este mismo capítulo, aprendimos cómo añadir a un elemento de un menú el texto correspondiente a las teclas que definen el acelerador, propiedad **InputGestureText**, y pospusimos la lógica del acelerador (también llamado método abreviado o atajo) al estudio de las órdenes enrutadas. Pues bien, continuemos con el ejemplo que allí iniciamos y hagamos operativo el acelerador *Ctrl+A* asociado con la orden enrutada *RCAbrir* que se invoca desde el elemento *Abrir* (identificado *ArchivoAbrir*) del menú *Archivo*:

```
<MenuItem Name="ArchivoAbrir" Header="A_brir..."
          Command="{x:Static local:MainWindow.RCAbrir}"
          InputGestureText="Ctrl+A" />
```

Una entrada desde el teclado o una entrada del ratón se pueden asociar con un controlador de una orden enrutada para que se ejecute cuando el usuario realice dicha entrada. Para vincular una combinación de teclas con una orden hay que crear un objeto de la clase **KeyBinding** derivada de **InputBinding** que encapsule esta información: "orden + acelerador de teclado", y después añadir dicho objeto a la colección **InputBindings** del elemento apropiado del árbol visual. Por ejemplo, el siguiente código añade a la colección **InputBindings** de la ventana (en este ejemplo, **Me** referencia a *MainWindow*) el acelerador *Ctrl+A* vinculado con la orden *RCAbrir* implementada anteriormente:

```
Dim kb As New KeyBinding(RCAbrir,
                         Key.A,
                         ModifierKeys.Control)
Me.InputBindings.Add(kb)
```

Este mismo código escrito en XAML sería así:

```
<Window.InputBindings>
  <KeyBinding Key="A"
              Modifiers="Control"
              Command="{x:Static local:MainWindow.RCAbrir}" />
</Window.InputBindings>
```

Una combinación de teclas se define mediante un objeto **KeyGesture**. Este objeto encapsula una sola tecla normal (**Key**) y una o varias teclas modificadoras separadas por el signo + (**ModifierKeys**). Tal información puede ser proporcionada a **KeyBinding** por medio de las propiedades **Key** y **Modifiers** o por medio de la propiedad **Gesture**, según se muestra a continuación:

```
<Window.InputBindings>
  <KeyBinding Gesture="Ctrl+A"
              Command="{x:Static local:MainWindow.RCAbrir}" />
</Window.InputBindings>
```

La propiedad **Gesture** hace referencia a un objeto de la clase **KeyGesture** derivada de **InputGesture** que encapsula el acelerador de teclado. El código XAML anterior puede escribirse en Visual Basic así:

```
Dim kb As New KeyBinding(
                RCAbrir,
                New KeyGesture(Key.A, ModifierKeys.Control))
Me.InputBindings.Add(kb)
```

En la mayoría de los casos, **KeyGesture** debe ser una combinación de una tecla normal y una o más teclas modificadoras. La excepción a esta regla son las teclas de función y las del teclado numérico, que se pueden utilizar solas.

En general, un enlace **InputBinding** representa un enlace entre **InputGesture** y una orden y se puede definir para un objeto concreto, por ejemplo para una ventana o una caja de texto o, utilizando el método **RegisterClassInputBinding** de **CommandManager**, asociarlo a una clase de objetos.

Según lo expuesto hasta ahora, ¿cómo añadiríamos un acelerador de teclado *Ctrl+S* para el elemento *Salir* del menú *Archivo*? Para salir de la aplicación podemos invocar al método **Close** desde el controlador del evento **Click** del elemento *Salir* o utilizar la orden enrutada **ApplicationCommands.Close**. La primera opción ya la vimos cuando iniciamos el desarrollo de la aplicación *Menus*; en ese caso, el controlador para el evento **Click** de *Salir* lo implementamos así:

```
Private Sub ArchivoSalir_Click(sender As Object, e As RoutedEventArgs)
  ' Cerrar la ventana principal;
  Close()
End Sub
```

La segunda opción es utilizar este mismo controlador como lógica para la orden **ApplicationCommands.Close**. Para ello, creamos un objeto **CommandBinding** que vincule la orden enrutada **Close** con el controlador *ArchivoSalir_Click* (en este caso, eliminaremos en el elemento *ArchivoSalir* la asociación *Click="ArchivoSalir_Click"*):

```
<Window.CommandBindings>
  ...
  <CommandBinding Command="ApplicationCommands.Close"
                  Executed="ArchivoSalir_Click" />
</Window.CommandBindings>
```

Y a continuación, vinculamos el acelerador *Ctrl+S* con la orden **Close**:

```
<Window.InputBindings>
  ...
  <KeyBinding Key="S"
              Modifiers="Control"
              Command="ApplicationCommands.Close" />
</Window.InputBindings>
```

Finalmente, invocamos a la orden **Close** desde el elemento *Salir* del menú *Archivo*:

```
<MenuItem Name="ArchivoSalir" Header="_Salir"
          InputGestureText="Ctrl+S"
          Command="Close" />
```

Ejecute ahora la aplicación y observe los resultados.

Y, ¿si no dispusiéramos de **ApplicationCommands.Close**? Pues, aplicando lo estudiado anteriormente, podríamos crear una nueva orden enrutada *RCSalir* y vincularla con el controlador *ArchivoSalir_Click*, procediendo de forma análoga a como acabamos de explicar.

Para crear *RCSalir* añada a la clase *MainWindow* la siguiente línea de código:

```
Public Shared RCSalir As New RoutedUICommand()
```

Una vez implementada la orden enrutada *RCSalir*, creamos un objeto **CommandBinding** que vincule la orden *RCSalir* con el controlador que proporciona la lógica para dicha orden, en nuestro caso *ArchivoSalir_Click*:

```
<Window.CommandBindings>
  ...
  <CommandBinding Command="{x:Static local:MainWindow.RCSalir}"
                  Executed="ArchivoSalir_Click" />
</Window.CommandBindings>
```

Y a continuación, vinculamos el acelerador *Ctrl+S* con la orden *RCSalir*:

```
<Window.InputBindings>
  ...
```

```
  <KeyBinding Key="S"
              Modifiers="Control"
              Command="{x:Static local:MainWindow.RCSalir}"  />
</Window.InputBindings>
```

Finalmente, invocamos a la orden *RCSalir* desde el elemento *Salir* del menú *Archivo*:

```
<MenuItem Name="ArchivoSalir" Header="_Salir"
          InputGestureText="Ctrl+S"
          Command="{x:Static local:MainWindow.RCSalir}" />
```

Y, ¿cómo añadiríamos un acelerador de ratón *Ctrl+MiddleClick* para la orden *Salir* del menú *Archivo*? Suponiendo que ya disponemos de un enlace para la orden **ApplicationCommands.Close**,

```
<Window.CommandBindings>
  ...
  <CommandBinding Command="ApplicationCommands.Close"
                  Executed="ArchivoSalir_Click" />
</Window.CommandBindings>
```

el paso siguiente sería vincular el acelerador de ratón *Ctrl+MiddleClick* con la orden **Close**. Para ello, creamos un objeto de la clase **MouseBinding** derivada de **InputBinding** que encapsule esta información: "orden + acelerador de ratón", y después añadimos dicho objeto a la colección **InputBindings** del elemento adecuado del árbol visual:

```
<Window.InputBindings>
  ...
  <MouseBinding Gesture="Control+MiddleClick"
                Command="Close"/>
</Window.InputBindings>
```

La propiedad **Gesture** hace referencia a un objeto de la clase **MouseGesture**, derivada de **InputGesture**, que encapsula el acelerador de ratón. El código XAML anterior puede escribirse en Visual Basic así:

```
Dim cb As New CommandBinding(ApplicationCommands.Close,
                             AddressOf ArchivoSalir_Click)
Me.CommandBindings.Add(cb)

Dim mbArchivoSalir As New MouseBinding(ApplicationCommands.Close,
New MouseGesture(MouseAction.MiddleClick, ModifierKeys.Control))
Me.InputBindings.Add(mbArchivoSalir)
```

Información adicional en las órdenes enrutadas

Anteriormente vimos que la mayoría de las órdenes que WPF pone a nuestra disposición son objetos **RoutedUICommand**, los cuales incluyen el texto descriptivo de la orden (propiedad **Text**), el nombre declarado para esa orden (propiedad **Name**), el tipo que registra la orden (propiedad **OwnerType**) y la colección de aceleradores de teclado y/o de ratón especificados (propiedad **InputGestures**). Siguiendo estas directrices, como ejemplo, vamos a implementar la orden personalizada *Guardar* para que incluya la información adicional citada. Esta orden guardaría el contenido de la caja de texto que tenga el foco. Para ello, vamos a partir de la aplicación *Menus* anterior, pero suponiendo que el menú *Archivo* contiene una orden *Guardar* (no un submenú *Guardar*; haga esta modificación).

```
<MenuItem Name="ArchivoGuardar" />
```

Empecemos por añadir una carpeta *Ordenes* al proyecto con la intención de guardar ahí las órdenes personalizadas. A continuación, agregue a esa carpeta una clase *OrdenesPersonalizadas* y, según lo explicado en los apartados anteriores, complétela como se indica a continuación:

```
Public Class OrdenesPersonalizadas
  Private Shared m_guardar As RoutedUICommand

  Shared Sub New()
    ' Iniciar la orden
    Dim acelerador As New InputGestureCollection()
    acelerador.Add(
      New KeyGesture(Key.G, ModifierKeys.Control, "Ctrl+G"))
    m_guardar = New RoutedUICommand("_Guardar",
                                    "Guardar",
                                    GetType(OrdenesPersonalizadas),
                                    acelerador)
  End Sub

  Public Shared ReadOnly Property Guardar() As RoutedUICommand
    Get
      Return m_guardar
    End Get
  End Property
End Class
```

Analice el código sombreado. En él puede observar la información adicional añadida a la orden *Guardar*.

A continuación añada los controladores para los eventos **Execute** y **CanExecute** del objeto **CommandBinding**. Estos controladores, al igual que hacemos pa-

ra las órdenes predefinidas, los añadiremos en el fichero de código subyacente de la ventana, esto es, en la clase *MainWindow*, que es lo que haría el usuario de la orden, colocarlos en el elemento sobre el que se ejecuta la orden:

```
Private Sub ExecutedGuardar(sender As Object,
e As ExecutedRoutedEventArgs)
    ' Implemente aquí la lógica de la orden Guardar
End Sub

Private Sub CanExecuteGuardar(sender As Object,
e As CanExecuteRoutedEventArgs)
    ' Implemente aquí la lógica de la orden Guardar
End Sub
```

Una vez definida la orden, ya se puede utilizar en los enlaces de órdenes de la misma forma que cualquiera de las órdenes predefinidas. Sin embargo, para poder utilizarla en el código XAML, hay que especificar el espacio de nombres .NET como un espacio de nombres XML. En nuestro caso esto se hace así:

```
xmlns:local="clr-namespace:Menus"
```

El paso siguiente consiste en crear un objeto **CommandBinding** que vincule la orden *Guardar* con los controladores *ExecutedGuardar* y *CanExecuteGuardar* que proporcionan la lógica para dicha orden. En principio, añadiremos el objeto de enlace de la orden a la colección de enlaces de órdenes de la ventana:

```
<Window x:Class="MainWindow"
        ...
        xmlns:local="clr-namespace:Menus"
        Title="Menús" Height="350" Width="525">
  <Window.CommandBindings>
    ...
    <CommandBinding
          Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
          Executed="ExecutedGuardar"
          CanExecute="CanExecuteGuardar" />
  </Window.CommandBindings>
  ...
</Window>
```

Finalmente, invocamos a la orden *Guardar* desde el elemento *Guardar* del menú *Archivo*:

```
<MenuItem Name="ArchivoGuardar"
          Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
/>
```

Ejecute ahora la aplicación y analice los resultados. Observará que el título del elemento del menú, *Guardar*, es tomado de la información adicional que proporcionamos al implementar la orden; lo mismo ocurre con los aceleradores.

¿Dónde se aplica la orden?

El resultado de ejecutar una orden varía dependiendo de dónde se desencadenó. Por ejemplo, si tiene dos cajas de texto, ambas soportan las órdenes *Cortar*, *Copiar* y *Pegar*, pero, como hemos visto anteriormente, la operación seleccionada ocurre sólo en la caja de texto que actualmente tiene el foco.

Y, ¿qué ocurrirá con la orden *Guardar* implementada en el apartado anterior? Veamos. Implemente los controladores que responderán a los eventos **Executed** y **CanExecute** del enlace de la orden, por ejemplo, así:

```
Private Sub ExecutedGuardar(sender As Object,
e As ExecutedRoutedEventArgs)
  ' Implemente aquí la lógica de la orden Guardar
  MessageBox.Show("Se guarda el texto: " +
                  DirectCast(e.Source, TextBox).Text)
End Sub

Private Sub CanExecuteGuardar(sender As Object,
e As CanExecuteRoutedEventArgs)
  ' Retorna True sólo si el destino de la orden es un TextBox
  If TypeOf e.Source Is TextBox Then
    e.CanExecute = True
  Else
    e.CanExecute = False
  End If
End Sub
```

Cuando ahora ejecute la aplicación, observará que la orden *Guardar* permanece inhabilitada y se habilita en cuanto coloque el punto de inserción en una caja de texto. Esto es así porque, al no establecer la propiedad **CommandTarget** del elemento *ArchivoGuardar*, el destino de la orden es el elemento que tiene el foco del teclado (véase el apartado *Modelo de una orden enrutada* expuesto anteriormente) y, además, según especifica la condición impuesta en el controlador *CanExecuteGuardar*, ese elemento tiene que ser un **TextBox**. *CanExecuteGuardar* es invocado en el instante en el que se intenta ejecutar la orden *Guardar*; en este caso, cuando se va a desplegar el menú para mostrar la orden *Guardar* (en el caso de un **Button** sería invocado en el instante de crear el botón).

Pero tenemos un pequeño problema y es que el acelerador *Ctrl+G* no funciona. ¿Qué sucede? Al pulsar las teclas *Ctrl+G* se invoca al controlador *Can-*

ExecuteGuardar que devuelve **False** debido a que *e.Source* hace referencia al objeto *MainWindow*, la ventana. ¿Por qué? Porque hemos asociado el enlace de la orden (el objeto **CommandBinding**) con el objeto **Window**; entonces, el acelerador incluido en la orden queda también asociado con ese objeto **Window**. La solución pasa por establecer el enlace de la orden en cada una de las cajas de texto:

```
<TextBox ... >
  <TextBox.CommandBindings>
    <CommandBinding
       Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
       Executed="ExecutedGuardar"
       CanExecute="CanExecuteGuardar" />
  </TextBox.CommandBindings>
</TextBox>
```

Lo que sucede ahora es que esta implementación utiliza dos **CommandBinding**, uno para cada caja de texto, cuando realmente sólo necesitamos uno, según vimos anteriormente. Piense en la cantidad de código que tendríamos que añadir si tuviéramos más cajas de texto y más órdenes compartidas por esas cajas: el mantenimiento sería más costoso. La solución a este problema es crear sólo un objeto **CommandBinding** y añadirlo a la colección de recursos de la ventana:

```
<Window.Resources>
  <CommandBinding x:Key="EnlaceGuardar"
          Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
          Executed="ExecutedGuardar"
          CanExecute="CanExecuteGuardar" />
</Window.Resources>
```

Una vez creado el recurso, en nuestro caso *EnlaceGuardar*, añádalo a la colección **CommandBindings** de cada una de las cajas de texto:

```
<TextBox ... >
  <TextBox.CommandBindings>
    <StaticResource ResourceKey="EnlaceGuardar" />
  </TextBox.CommandBindings>
</TextBox>
```

Utilizar parámetros

En ocasiones una orden podrá requerir información adicional proporcionada por el origen de la orden. Por ejemplo, la orden *Guardar* de la aplicación *Menus* anterior tiene como fin guardar el contenido de la caja de texto que tenga el foco. Ahora bien, si necesitamos guardar el contenido de cada una de las cajas en un fichero diferente, el controlador *ExecutedGuardar* tendrá que conocer el nombre

del fichero a utilizar en cada caso. Tal información puede ser proporcionada a través de la propiedad **Parameter** del objeto **ExecutedRoutedEventArgs**:

```
Private Sub ExecutedGuardar(sender As Object,
e As ExecutedRoutedEventArgs)
  ' Implemente aquí la lógica de la orden Guardar
  MessageBox.Show("Se guarda el texto """ +
                  DirectCast(e.Source, TextBox).Text & """ en " +
                  e.Parameter.ToString())
End Sub
```

¿De dónde procede el valor de la propiedad **Parameter**? Según expusimos anteriormente, de la propiedad **CommandParameter** del origen de la orden (objeto **ICommandSource**).

Supongamos que el nombre del fichero a utilizar en cada caso es proporcionado por un objeto de una clase *DestinoDatos* y que esta clase proporciona una propiedad *Destino* que almacena tal nombre. Según esto, el objeto origen de la orden podría estar configurado así:

```
<MenuItem Name="ArchivoGuardar"
          Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
          CommandParameter="{Binding Path=Destino}" />
```

En este código observamos que la propiedad **CommandParameter** está vinculada con la propiedad *Destino* de un objeto *DestinoDatos* que puede ser proporcionado a través del contexto de datos de la ventana *MainWindow*, por ejemplo:

```
Class MainWindow
  Public Property FicheroGuardar() As DestinoDatos

  Sub New()
    ' Llamada necesaria para el diseñador.
    InitializeComponent()
    Me.FicheroGuardar = New DestinoDatos()
    Me.DataContext = FicheroGuardar
  End Sub
  ...

End Class
```

La propiedad *Destino* del objeto *DestinoDatos* referenciado por *FicheroGuardar* tendrá que ser actualizada, con el nombre del fichero en el que hay que escribir, en función de la caja de texto que tenga el foco. De este trabajo se encargarán los controladores del evento **GotFocus** de las cajas de texto (evidentemente, el nombre del fichero será obtenido de la fuente que nos interese en cada caso):

```
Private Sub textBox1_GotFocus(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  FicheroGuardar.Destino = "datos1.txt"
End Sub

Private Sub textBox2_GotFocus(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  FicheroGuardar.Destino = "datos2.txt"
End Sub
```

Cada vez que se modifique la propiedad *Destino* del objeto *DestinoDatos* referenciado por *FicheroGuardar* se modificará la propiedad **CommandParameter** del elemento *ArchivoGuardar* del menú *Archivo*. Éste es el trabajo que hace automáticamente el objeto **Binding**. Para ello, la clase *DestinoDatos* debe implementar la interfaz **INotifyPropertyChanged**, ya que es la que utiliza el motor de enlace a datos de WPF para mantener la interfaz gráfica sincronizada con el objeto origen del enlace. Según esto, añada la clase *DestinoDatos* a la carpeta *Ordenes* del proyecto y complétela como se indica a continuación:

```
Imports System.ComponentModel

Public Class DestinoDatos
  Implements INotifyPropertyChanged
  Private m_destino As String

  Public Sub New()
    Destino = "default"
  End Sub

  Public Property Destino() As String
    Get
      Return m_destino
    End Get
    Set(value As String)
      m_destino = value
      NotificarCambio("Destino")
    End Set
  End Property

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged

  Private Sub NotificarCambio(nombreProp As String)
    RaiseEvent PropertyChanged(Me,
                New PropertyChangedEventArgs(nombreProp))
  End Sub
End Class
```

El proceso descrito hasta ahora funciona bien cuando el usuario para ejecutar la orden *Guardar* la selecciona utilizando el teclado o el ratón, pero no funciona cuando para ejecutar dicha orden utiliza el acelerador *Ctrl+G*. ¿Por qué? Porque el objeto **KeyBinding** vinculado por omisión con las cajas de texto tiene conocimiento del acelerador y de la orden vinculada, información que fue proporcionada al implementar dicha orden, pero no del parámetro que hay que pasar. Para solucionar este problema, añada este objeto a la colección **InputBindings** de cada una de las cajas de texto, especificando el parámetro a pasar:

```
<TextBox.InputBindings>
  <KeyBinding Key="G" Modifiers="Control"
        Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
        CommandParameter="{Binding Path=Destino}" />
</TextBox.InputBindings>
```

Finalmente, si su aplicación requiere que la propiedad *FicheroGuardar* sea una propiedad de dependencia puede escribirla así:

```
Public Shared ReadOnly FicheroGuardarProperty As DependencyProperty =
  DependencyProperty.Register("FicheroGuardar",
                              GetType(DestinoDatos),
                              GetType(MainWindow),
                              New UIPropertyMetadata(Nothing))

Public Property FicheroGuardar() As DestinoDatos
  Get
    Return DirectCast(GetValue(FicheroGuardarProperty), Desti-
noDatos)
  End Get
  Set(value As DestinoDatos)
    SetValue(FicheroGuardarProperty, value)
  End Set
End Property
```

ICommand versus RoutedCommand

Anteriormente, en este mismo capítulo, al definir una *orden enrutada* dijimos que era un objeto que implementaba la interfaz **ICommand**, y que para facilitar la creación de *órdenes enrutadas*, WPF proporcionaba la clase **RoutedCommand** que es la implementación WPF de **ICommand**. Desde esta definición, también podemos decir que una *orden enrutada* es un objeto que implementa la interfaz **ICommand**. Esta interfaz está definida así:

```
Public Interface ICommand
  ' Evento CanExecuteChanged:
  '      Se produce cuando hay cambios que influyen en si
```

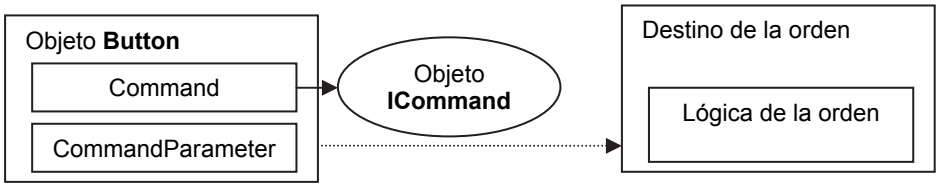
```
    '     la orden debería ejecutarse o no.
  Event CanExecuteChanged As EventHandler

    ' Método CanExecute:
    '     Define el método que determina si la orden se puede
    '     ejecutar en su estado actual.
    '
    ' Parámetros:
    '   parámetro:
    '     Datos que usa la orden. Si la orden no exige pasar
    '     los datos, se puede establecer este objeto en Nothing.
    '
    ' Devuelve:
    '     True si esta orden se puede ejecutar; de lo contrario,
    '     devuelve false.
  Function CanExecute(parámetro As Object) As Boolean

    ' Método Execute:
    '     Define el método al que se va a llamar cuando se invoca
    '     la orden.
    '
    ' Parámetros:
    '   parámetro:
    '     Datos que usa la orden. Si la orden no exige pasar
    '     los datos, se puede establecer este objeto en Nothing.
  Sub Execute(parámetro As Object)
End Interface
```

La interfaz **ICommand** permite implementar una abstracción de una llamada parametrizada a un método a través de su método **Execute**. Además, el usuario determinará si se puede o no llamar a ese método escribiendo el código adecuado en el método **CanExecute**.

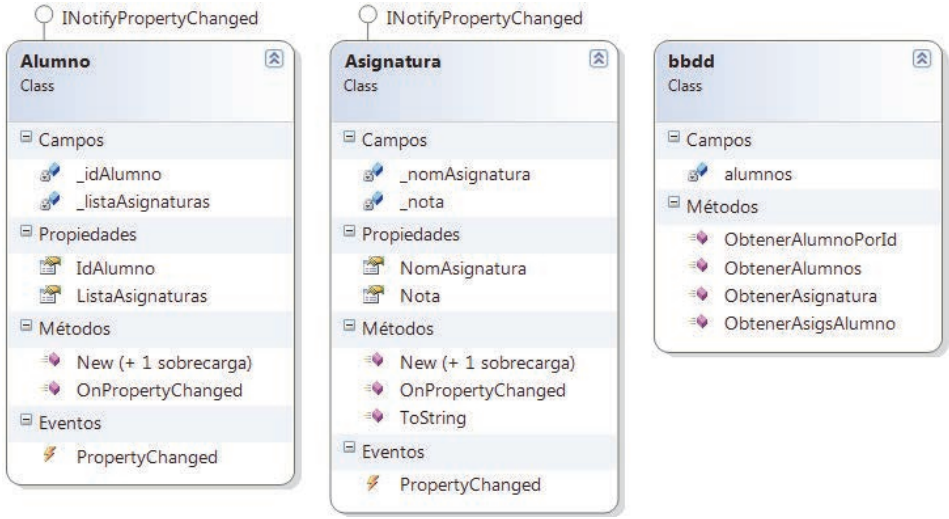
Normalmente, los objetos implementan esta interfaz para habilitar llamadas a métodos a través de enlaces XAML (objetos **Binding**). Esto resulta sobre todo útil bajo el modelo de programación *Model-View-ViewModel* (MVVM: modelo-vista-vista modelo) ya que permite al modelo exponer las órdenes a las que se pueden enlazar controles, como botones, según muestra la figura siguiente, sin que sea necesario ningún código adicional.



Según indica esta figura, puede enlazar la propiedad **Command** de algunos objetos y de sus clases derivadas, como **Button**, a objetos que implementen la interfaz **ICommand**.

Como ejemplo vamos a realizar una aplicación que permita a un usuario ver la nota que ha obtenido en una determinada asignatura elegida del conjunto de asignaturas de las que se matriculó para realizar determinados estudios. Este ejemplo pondrá en práctica cómo enlazar un botón, objeto de la clase **Button**, con una orden, objeto que implementa la interfaz **ICommand**. Lógicamente, la llamada a la orden será habilitada a través de la propiedad **Command** del botón.

Los datos serán tomados de una base de datos modelada por medio de las clases siguientes:



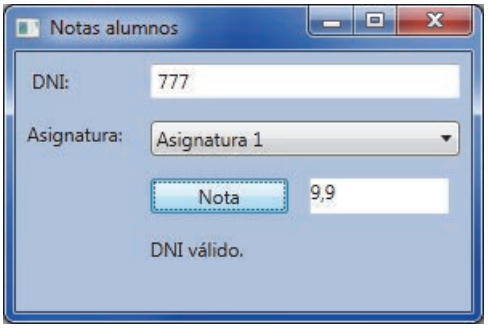
Realizamos una breve descripción de este modelo de datos, puesto que el interés de la aplicación está en ver cómo implementar la orden que denominaremos *ordenMostrarNota* y su enlace con el botón que ejecutará la misma. El modelo que hemos realizado de la base de datos es simplemente para presentar una aplicación trabajando. El estudio de este modelo y de las bases de datos lo realizaremos en próximos capítulos.

La clase *Asignatura*, que representa una asignatura, básicamente presenta dos propiedades: el nombre de la asignatura y la nota que se obtenga en la misma. La clase *Alumno* representa un alumno; también presenta dos propiedades: el identificador del alumno y la lista de asignaturas (objeto **List (Of** *Asignatura***)**) de las que se ha matriculado el alumno. La clase *bbdd* representa la base de datos; proporciona una colección **List (Of** *Alumno***)** con los datos de todos los alumnos matriculados; esta colección será devuelta por el método *ObtenerAlumnos*. También

proporciona los métodos *ObtenerAlumnoPorId* que devuelve el objeto *Alumno* correspondiente al identificador pasado como argumento, *ObtenerAsigsAlumno* que devuelve la colección **List (Of** *Asignatura***)** correspondiente a la lista de las asignaturas de las que se ha matriculado el alumno cuyo identificador se pasa como argumento, y *ObtenerAsignatura* que devuelve el objeto *Asignatura* correspondiente al identificador del alumno y al nombre de la asignatura pasados como argumentos. Puede ver el código de la aplicación en el CD del libro.

Es importante resaltar que las clases *Alumno* y *Asignatura* implementan la interfaz **INotifyPropertyChanged** para que los objetos de estas clases puedan notificar a la interfaz gráfica un cambio en el valor de cualquiera de sus propiedades, con el fin de que los elementos de dicha interfaz que estén enlazados con estos objetos puedan ser actualizados automáticamente (véase el apartado *Notificar cuándo cambia una propiedad* del capítulo *Introducción a WPF*). La notificación la realizan los objetos de esas clases generando el evento **PropertyChanged** que lleva asociado como dato el nombre de la propiedad cuyo valor ha cambiado. Recuerde que el motor de enlace a datos de WPF utiliza esta interfaz para mantener la interfaz gráfica sincronizada con el objeto origen del enlace.

Una vez que disponemos del modelo de datos, el paso siguiente puede ser diseñar la interfaz gráfica. La aplicación mostrará una ventana como la siguiente:



Entonces, como siguiente paso, vamos a crear una aplicación WPF y a diseñar la interfaz que muestra la figura anterior. Dicha interfaz estará formada, además de por las etiquetas que se muestran (objetos **TextBlock**), por un **TextBox** (identificador del alumno), un **ComboBox** (lista de asignaturas de las que se ha matriculado el alumno), un **Button** (botón que ejecutará la orden para obtener la nota de la asignatura seleccionada), un **TextBlock** (para mostrar la nota obtenida) y otro **TextBlock** para especificar si el identificador introducido (DNI, por ejemplo) se corresponde o no con uno de los registrados en la base de datos. El código XAML que define la interfaz descrita puede ser el siguiente:

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

```
        xmlns:local="clr-namespace:Ap_ICommand"
        Title="Notas alumnos" Height="200" Width="300">

  <Grid Background="LightSteelBlue">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

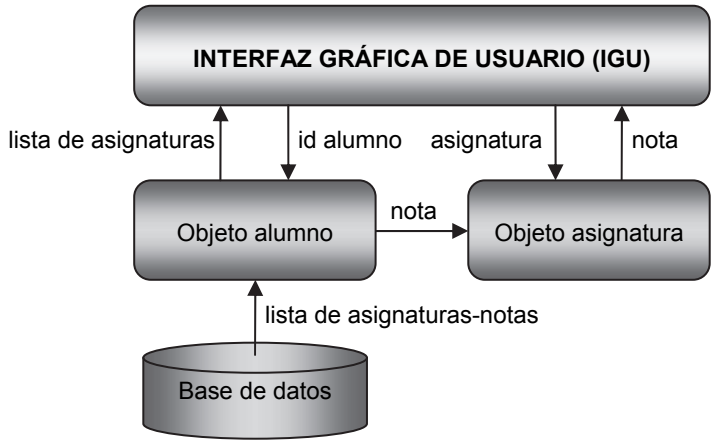
    <StackPanel>
      <TextBlock Text="DNI: " Margin="10"/>
      <TextBlock Text="Asignatura: " Margin="7"/>
    </StackPanel>

    <StackPanel Grid.Column="1">
      <TextBox Name="ctDni" Margin="7"
               Text="{Binding Path=IdAlumno,
                              UpdateSourceTrigger=PropertyChanged}"
               TextChanged="ctDni_TextChanged" />
      <ComboBox Name="lsdAsigs" Margin="7"
                SelectedItem="{Binding Path=NomAsignatura}" />
      <WrapPanel ItemWidth="100">
        <Button Name="btNota" Margin="7" Content="Nota"
             Command="{x:Static local:MainWindow.ordenMostrarNota}"
             CommandParameter="{Binding ElementName=lsdAsigs,
                                        Path=SelectedIndex}"/>
        <TextBlock Name="tbNota" Margin="7" Background="White"
                   Text="{Binding Path=Nota,
                                  ConverterCulture=es-ES}" />
      </WrapPanel>
      <TextBlock Name="tbError" Text="Error: " Margin="7" />
    </StackPanel>
  </Grid>
</Window>
```

Vamos a analizar las partes importantes que hemos resaltado de este código. El objeto **Window** define el espacio de nombres *Ap_ICommand* al que pertenecen las clases de la aplicación; dicho espacio de nombres será identificado por el prefijo *local:*. Esta declaración es necesaria para acceder a la orden *ordenMostrarNota* definida en la clase *MainWindow* así:

```
Class MainWindow
  ' Command: OrdenMostrarNota
  Private Shared alumno As New Alumno()
  Private Shared asignatura As New Asignatura()
  Public Shared ordenMostrarNota As New OrdenMostrarNota( _
                                    alumno, asignatura, 111, 999)
  ...
```

Dicha orden es un objeto de la clase *OrdenMostrarNota* que tiene cuatro atributos (observe los argumentos pasados en la llamada al constructor): los objetos *alumno* (alumno que se identifica por medio de la caja de texto *ctDni*) y *asignatura* (asignatura de la cual el alumno desea conocer su nota), y los valores mínimo y máximo entre los que se encuentran los identificadores de los alumnos. Estos objetos se crean inicialmente vacíos y se llenarán con los datos proporcionados por el usuario a través de la interfaz gráfica (identificador del alumno y nombre de la asignatura) y los proporcionados, a partir de éstos, por el código subyacente y por la orden cuando se ejecute.



Siguiendo con el análisis del código XAML observamos que la propiedad **Text** de la caja de texto *ctDni* (donde el alumno introducirá su identificador) está enlazada con la propiedad *IdAlumno* del objeto *alumno*; este enlace hará que esas propiedades de los objetos enlazados reflejen los cambios automáticamente cuando el valor de una u otra cambie (para una caja de texto el enlace trabaja en las dos direcciones; obsérvese que el atributo **UpdateSourceTrigger** del enlace lo hemos fijado al valor **PropertyChanged**). También, la propiedad **SelectedItem** del **ComboBox** está enlazada con la propiedad *NomAsignatura* del objeto *asignatura* (en este caso el enlace trabaja en una sola dirección, hacia *NomAsignatura*). Observamos también que la propiedad **Text** del **TextBlock** *tbNota* está enlazada con la propiedad *Nota* del objeto *asignatura* (en este caso el enlace también trabaja en una sola dirección, hacia la propiedad **Text**).

Finalmente, analizamos el botón *btNota*. Su propiedad **Command** hace referencia a la orden, *ordenMostrarNota*, que se ejecutará cuando el usuario haga clic en dicho botón y su propiedad **CommandParameter** especifica el parámetro que se pasará a los métodos **CanExecute** y **Execute** de la orden; para nuestro ejemplo, este parámetro será el índice de la asignatura seleccionada en la lista desplegable (propiedad **SelectedIndex** de *lsdAsigs*).

Respecto a los enlaces (objetos **Binding**), no hay duda de los objetos destino de los mismos, ya que están definidos en los propios destinos (objetos de la IGU), pero, ¿cómo indicamos a los objetos **Binding** que las propiedades origen del enlace especificadas pertenecen a uno u otro objeto? Pues, mediante los contextos de datos. El aspecto más importante de un contexto de datos es hacer referencia al origen de datos que se utiliza para el enlace. Está claro que las propiedades origen del enlace pertenecen a los objetos *alumno* y *asignatura*. Según vimos anteriormente, estos objetos fueron construidos en la clase *MainWindow* para definir la orden *ordenMostrarNota*. Entonces, podemos utilizar el constructor de *MainWindow* para establecer los contextos de datos de los elementos de la IGU implicados:

```
Public Sub New()
  InitializeComponent()
  ' Iniciar contextos de datos
  ctDni.DataContext = alumno
  lsdAsigs.DataContext = asignatura
  tbNota.DataContext = asignatura
End Sub
```

La interfaz está lista, pero para que pueda trabajar tenemos que escribir el código subyacente necesario y la orden *ordenMostrarNota*. Para ello, pensemos cómo se sucederán las acciones cuando el usuario interactúe con la misma. Observe las figuras anteriores. Una vez que el usuario ponga en marcha la aplicación:

1. Se crean los objetos *alumno* y *asignatura* y la orden *ordenMostrarNota*.

2. Se ejecuta el constructor de la ventana, se crea la interfaz gráfica (IGU) y se definen los contextos de datos con los que interactuará la IGU.

3. El usuario se identifica (escribe su DNI en la caja de texto correspondiente). Este valor se reflejará automáticamente en la propiedad *IdAlumno* del objeto *alumno*.

4. Si el identificador no corresponde a ningún alumno, el **TextBlock** *tbError* lo indicará, para que el usuario rectifique.

5. Si el identificador introducido corresponde a un alumno, se asignará al objeto *alumno* la lista de asignaturas en las que está matriculado el alumno identificado, con sus notas correspondientes, y se poblará la lista desplegable con los nombres de esas asignaturas. También, el **TextBlock** *tbError* indicará que el identificador es válido.

El proceso descrito, concretamente los puntos 4 y 5, será ejecutado desde el controlador del evento **TextChanged** de la caja de texto *ctDni*. De esta forma ga-

rantizamos que, cuando el usuario haya escrito un identificador válido, la lista de asignaturas estará cargada.

```
Private Sub ctDni_TextChanged(sender As Object,
e As System.Windows.Controls.TextChangedEventArgs)
  lsdAsigs.ItemsSource = Nothing
  alumno.ListaAsignaturas.Clear()
  asignatura.Nota = -1.0F
  Dim id As Integer
  If Not Integer.TryParse(ctDni.Text, id) Then
    Return
  End If
  Dim listaAsigs As List(Of Asignatura) =
        bbdd.ObtenerAsigsAlumno(id)
  If listaAsigs Is Nothing Then
    tbError.Text = "Error: el DNI no existe."
    Return
  End If
  tbError.Text = "DNI válido."
  lsdAsigs.ItemsSource = listaAsigs
  alumno.ListaAsignaturas = listaAsigs
End Sub
```

Una vez ejecutado el código anterior:

6. El usuario seleccionará, de la lista desplegable, la asignatura de la cual desea conocer la nota. Este valor se reflejará automáticamente en la propiedad *NomAsignatura* del objeto *asignatura*.

7. Si todo el proceso anterior fue satisfactorio, el botón *btNota*, que hasta ahora estaba deshabilitado, se habilitará.

8. El usuario hará clic en el botón, se ejecutará la orden *ordenMostrarNota*, y se visualizará la nota para la asignatura y alumno seleccionados, en el control destinado a tal fin.

Sólo queda implementar la orden *ordenMostrarNota*. Para ello, añadimos al proyecto una nueva clase *OrdenMostrarNota* que implemente la interfaz **ICommand**, lo que nos obliga a implementar los elementos definidos por esa interfaz.

```
Imports System.ComponentModel

Public Class OrdenMostrarNota
  Implements ICommand
  Private alumno As Alumno
  Private asignatura As Asignatura
  Private idMin As Integer, idMax As Integer
```

```
  Public Sub New(alum As Alumno, asig As Asignatura,
  min As Integer, max As Integer)
    idMin = min
    idMax = max
    Me.alumno = alum
    AddHandler Me.alumno.PropertyChanged,
               AddressOf OnPropertyChanged
    Me.asignatura = asig
    AddHandler Me.asignatura.PropertyChanged,
               AddressOf OnPropertyChanged
  End Sub

  Public Function CanExecute(paramIndex As Object) _
  As Boolean Implements ICommand.CanExecute

  End Function

  Public Sub Execute(paramIndex As Object) _
  Implements ICommand.Execute

  End Sub

  Public Event CanExecuteChanged As EventHandler _
    Implements ICommand.CanExecuteChanged

  Private Sub OnPropertyChanged(sender As Object,
  e As PropertyChangedEventArgs)
    ' Generar evento
    RaiseEvent CanExecuteChanged(Me, EventArgs.Empty)
  End Sub
End Class
```

La clase define cuatro atributos, *alumno*, *asignatura*, *idMin* e *idMax*, a los que el constructor de la clase asignará los argumentos pasados cuando se crea un objeto de esta clase: objeto *alumno* para almacenar los datos del alumno que se identificará por medio de la caja de texto *ctDni*, objeto *asignatura* para almacenar los datos de la asignatura de la cual el alumno desea conocer su nota, y los valores mínimo y máximo entre los que se encuentran los identificadores de los alumnos.

Así mismo, la orden se suscribe, por medio del constructor, al evento **PropertyChanged** de los objetos *alumno* y *asignatura* para controlar los cambios que sucedan en las propiedades de los mismos. ¿Cómo responderá la orden a tal evento? Pues generando el evento **CanExecuteChanged** (observe el método **OnPropertyChanged**).

Podemos decir entonces que la orden genera el evento **CanExecuteChanged** cuando hay cambios que pueden influir en que la orden se pueda o no ejecutar. A parte de que cualquier otra clase puede suscribirse a este evento, cuando se genera

el evento citado se ejecuta el método **CanExecute** y es este método el que define si la orden se puede ejecutar en su estado actual. Por ejemplo, el código siguiente especifica que una orden *OrdenMostrarNota* no se podrá ejecutar si el identificador del alumno está fuera de los límites establecidos, si la lista de asignaturas no existe o si el nombre de la asignatura es nulo o está vacío.

```
Public Function CanExecute(paramIndex As Object) _
As Boolean Implements ICommand.CanExecute
  If alumno.IdAlumno >= idMin AndAlso alumno.IdAlumno <= idMax Then
    If alumno.ListaAsignaturas.Count <> 0 Then
      If Not String.IsNullOrEmpty(asignatura.NomAsignatura) Then
        Return True
      End If
    End If
  End If
  Return False
End Function
```

Si este método devuelve **False**, el control que esté vinculado con esta orden permanecerá inhabilitado de forma automática, por lo que no se podrá ejecutar la orden. Sólo se habilitará cuando este método devuelva **True**.

Finalmente, el método **Execute** define la acción a ejecutar cuando se hace clic en el control vinculado con la orden. Para nuestro ejemplo, este método, utilizando el índice correspondiente a la asignatura seleccionada de la lista, obtiene la nota de la lista de asignaturas del objeto *alumno* (recuerde que dicha lista fue cargada desde el controlador *ctDni_TextChanged*) y se la asigna a la propiedad *Nota* de la asignatura. El resto del trabajo (visualizar la nota) lo hace el objeto **Binding** del que es origen esta propiedad.

```
Public Sub Execute(paramIndex As Object) _
Implements ICommand.Execute
  Dim ind As Integer = Convert.ToInt32(paramIndex)
  asignatura.Nota = alumno.ListaAsignaturas(ind).Nota
End Sub
```

DETALLES DE UN ELEMENTO DE UN MENÚ

Según lo estudiado hasta ahora, un elemento de un menú normalmente muestra un texto por medio de su propiedad **Header** y en muchas ocasiones muestra también a la derecha del texto el acelerador o teclas que permiten el acceso directo a dicho elemento, información que se muestra por medio de la propiedad **InputGestureText**.

Pues bien, además de esta información, un elemento de un menú también puede mostrar una imagen a la izquierda del mismo, o bien una marca de verificación. Para asignar una imagen (objeto **Image**) a un elemento de un menú utilizaremos su propiedad **Icon**. Esta propiedad admite cualquier objeto, lo cual nos da la flexibilidad de asignar cualquier dibujo vectorial en miniatura.

Como ejemplo, vamos a asignar una imagen a la orden *Guardar* del menú *Archivo* de la aplicación *Menus* anterior. Para ello, en el esquema del documento seleccione el elemento *ArchivoGuardar*, después, en la ventana de propiedades, seleccione su propiedad **Icon** y elija la imagen que desea asignar. Esta acción dará lugar al código XAML que se muestra a continuación:

```
<MenuItem Name="ArchivoGuardar"
          Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
          CommandParameter="{Binding Path=Destino}">
  <MenuItem.Icon>
    <Image Source="/Menus;component/Imagenes/saveHS.png" />
  </MenuItem.Icon>
</MenuItem>
```

Para mostrar una marca de verificación junto a un elemento de menú, objeto de tipo **MenuItem**, sólo tendrá que asignar a su propiedad **IsChecked** el valor **True**. Además, si su propiedad **IsCheckable** vale **True**, al hacer clic sobre este elemento alternará su estado entre marcado y no marcado. Por ejemplo:

```
<MenuItem Name="OpsMostrar" Header="Mostrar" IsChecked="True" />
```

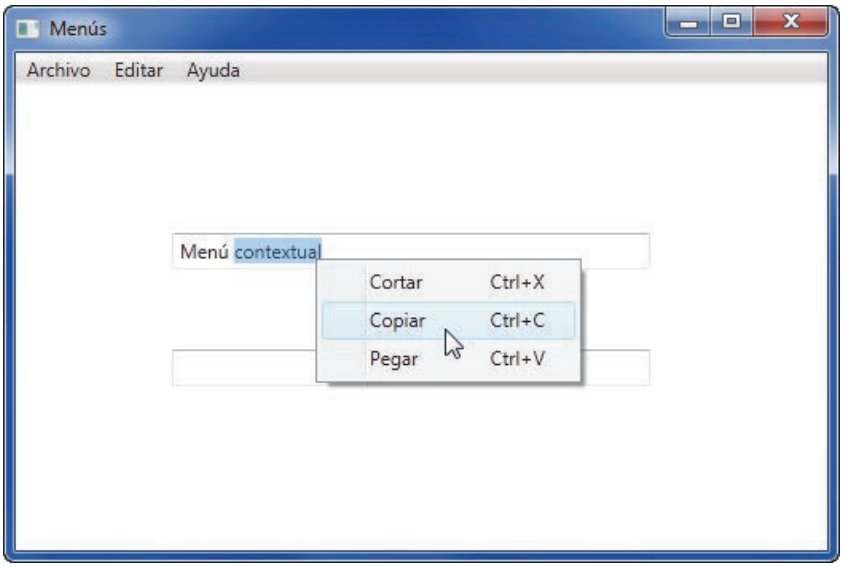
La propiedad **IsChecked** también puede ser establecida desde el código Visual Basic subyacente:

```
OpsMostrar.IsChecked = True;
```

Si estableciéramos conjuntamente las propiedades **Icon** e **IsChecked** sólo aparecería esta última.

MENÚS CONTEXTUALES

Un menú contextual, también llamado menú emergente, es un menú que se muestra normalmente al hacer clic con el botón secundario del ratón sobre un control, indicando las operaciones específicas que se pueden realizar sobre el mismo.



Un menú contextual es un objeto de la clase **ContextMenu** y, al igual que la clase **Menu**, contiene una colección de objetos **MenuItem**. La diferencia es que un **ContextMenu** no se puede colocar en una ventana, sino que hay que asignarlo a la propiedad **ContextMenu** de otro elemento **FrameworkElement** o **FrameworkContentElement**. Por ejemplo:

```
<TextBox ...>
  <TextBox.ContextMenu>
    <ContextMenu>
      <MenuItem Name="mcEditarCortar" Command="Cut" />
      <MenuItem Name="mcEditarCopiar" Command="Copy" />
      <MenuItem Name="mcEditarPegar" Command="Paste" />
    </ContextMenu>
  </TextBox.ContextMenu>
</ TextBox>
```

Puede también definir el menú contextual como un recurso y aplicarlo a varios elementos, por ejemplo a todas las cajas de texto de la ventana:

```
<TextBox ContextMenu="{StaticResource menuContextualEditar}" ...
```

Si un control tiene predefinido un menú contextual y desea reemplazarlo, asigne a su propiedad **ContextMenu** el menú contextual nuevo o asigne el valor **Nothing** si desea eliminarlo.

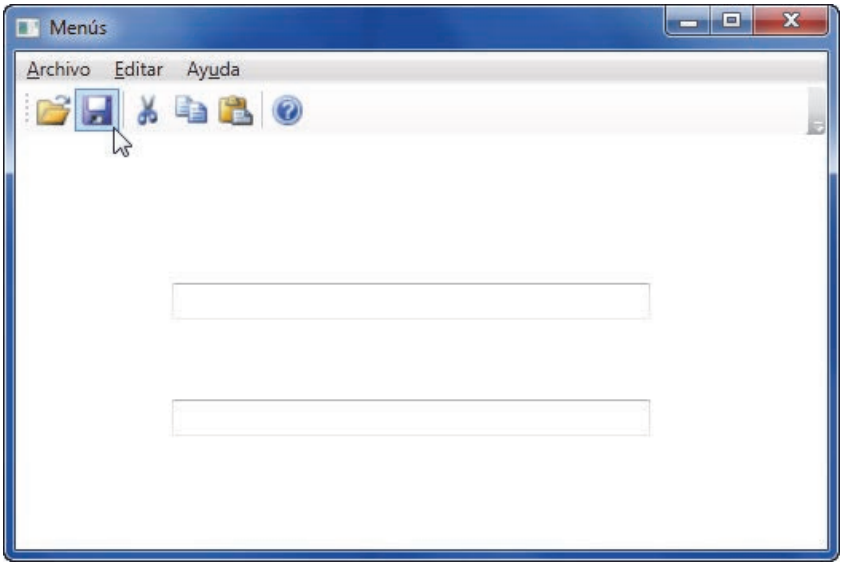
Cuando un usuario hace clic con el botón secundario del ratón sobre un control que tiene asignado un menú contextual, éste se muestra automáticamente siempre que el control esté habilitado: propiedad **IsEnabled** igual a **True**. No

obstante, estando el control deshabilitado, puede permitir que el menú contextual se muestre, asignando a la propiedad adjunta **ContextMenuService.ShowOnDisabled** del control el valor **True**.

```
<TextBox ContextMenuService.ShowOnDisabled="True" ...
```

BARRA DE HERRAMIENTAS

Una barra de herramientas es un contenedor que normalmente contiene botones gráficos, aunque también puede contener otros controles como, por ejemplo, cajas de texto o listas desplegables.



Un clic en un botón de la barra de herramientas produce los mismos eventos que un clic en la orden de un menú que realice su misma función o que el acelerador asociado con dicha orden. De hecho, las barras de herramientas y los menús están íntimamente relacionados. Normalmente se añadirá un botón a una barra de herramientas para evitar al usuario la molestia de abrir un menú para ejecutar una determinada orden. Esto es, las barras de herramientas proporcionan acceso más rápido para aquellas operaciones que se ejecutan frecuentemente, simplemente porque el usuario no tiene que navegar por las barras de menús.

Diseño de una barra de herramientas

Desde el punto de vista de la jerarquía de clases de la biblioteca WPF, una barra de herramientas es un objeto **ToolBar** que actúa como contenedor para objetos

como **Button**, **CheckBox**, **RadioButton**, **ComboBox**, **Separator**, etc. A su vez, un objeto **ToolBarTray** puede contener varias barras de herramientas.

Como ejemplo, vamos a añadir a la aplicación *Menus* anterior una barra de herramientas con los botones: *btbarAbrir*, *btbarGuardar*, separador, *btbarCortar*, *btbarCopiar*, *btbarPegar*, separador y *btbarAyuda* según muestra la figura anterior.

Tanto la barra de herramientas como la barra de menús las incluiremos en un control **StackPanel** colocado en la fila 0 de la rejilla:

```
<Window ...>
  ...
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <StackPanel Grid.Row="0" >
      <Menu Name="BarraDeMenus" >
        ...
      </Menu>
      <ToolBar Name="BarraDeHerramientas" >
        ...
      </ToolBar>
    </StackPanel>
    ...
  </Grid>
</Window>
```

Para añadir la barra de herramientas, arrastre desde la caja de herramientas de Visual Studio un control **ToolBar** sobre el **StackPanel** y póngale como nombre *BarraDeHerramientas*. El resto de las propiedades déjelas con sus valores por omisión. Después de estas acciones podrá comprobar que el asistente para diseño de formularios añadió a la clase *MainWindow* el siguiente código XAML:

```
<ToolBar Name="BarraDeHerramientas" >
  ...
</ToolBar>
```

Para añadir los botones y los separadores a la barra de herramientas, arrastre desde la caja de herramientas de Visual Studio sobre el control **ToolBar** los objetos **Button** y **Separator**, y configúrelos para que tengan como nombre el identificador indicado anteriormente, como contenido una imagen (objeto **Image**) adecuada a la función que tiene que desempeñar almacenada en la carpeta *Imágenes* del proyecto y, para el resto de las propiedades, deje los valores asignados por

omisión. Una vez realizado este proceso el asistente para diseño habrá añadido al contenedor **ToolBar** el siguiente código:

```
<ToolBar Name="BarraDeHerramientas" >
  <Button Name="btbarAbrir">
    <Image Source="Imagenes/openHS.png"/>
  </Button>
  <Button Name="btbarGuardar">
    <Image Source="Imagenes/saveHS.png"/>
  </Button>

  <Separator></Separator>

  <Button Name="btbarCortar">
    <Image Source="Imagenes/CutHS.png"/>
  </Button>
  <Button Name="btbarCopiar">
    <Image Source="Imagenes/CopyHS.png"/>
  </Button>
  <Button Name="btbarPegar">
    <Image Source="Imagenes/PasteHS.png"/>
  </Button>

  <Separator></Separator>

  <Button Name="btbarAyuda">
    <Image Source="Imagenes/Help.png"/>
  </Button>
</ToolBar>
```

Finalmente, asocie con cada botón el mismo controlador de eventos que tienen asociado sus órdenes respectivas. Para ello, proceda de forma análoga a como hizo con los elementos de los menús. Por ejemplo:

```
<Button Name="btbarGuardar"
        Command="{x:Static local:OrdenesPersonalizadas.Guardar}"
        CommandParameter="{Binding Path=Destino}">
  <Image Source="Imagenes/saveHS.png"/>
</Button>

<Button Name="btbarPegar" Command="Paste">
  <Image Source="Imagenes/PasteHS.png"/>
</Button>

<Button Name="btbarAyuda" Click="AyudaAcercaDe_Click">
  <Image Source="Imagenes/Help.png"/>
</Button>
```

Compile la aplicación, ejecútela y pruebe los resultados.

Si observa el aspecto de los botones de la barra de herramientas, se dará cuenta de que son bastante diferentes a cuando se colocan en la ventana. Por ejemplo, los botones en la barra de herramientas se muestran con una apariencia plana y optimizada que elimina el borde y el fondo sombreado, y se iluminan en azul cuando el puntero del ratón se coloca encima de ellos y no están inhabilitados (por ejemplo, porque la operación que especifican no se puede ejecutar). Lo único que sucede es que la barra de herramientas sobrescribe el estilo de sus elementos hijo, pero desde el punto de vista de ejecutar una acción, no hay diferencias en la forma de hacerlo, independientemente de que el elemento en cuestión se encuentre dentro o fuera de la barra de herramientas.

Si una barra de herramientas tiene más elementos de los que pueden caber en el ancho de la ventana, automáticamente elimina los elementos sobrantes colocándolos en un menú de desbordamiento, que se puede ver haciendo clic en la flecha que hay al final de la barra de herramientas.



Sin embargo, puede configurar este comportamiento en los elementos de la barra de herramientas a través de su propiedad asociada **ToolBar.OverflowMode**. Por ejemplo, utilice el valor *Never* para asegurar que un elemento nunca se pone en el menú de desbordamiento:

```
<Button Name="btbarAyuda" ToolBar.OverflowMode="Never" ...
```

Contenedor de barras de herramientas

La clase **ToolBarTray** permite administrar la colocación, el ajuste de tamaño, las operaciones de arrastrar y colocar, así como la reorganización de los controles **ToolBar**, y también es responsable de administrar las filas (o bandas) en las que aparecen dichos controles.

Por ejemplo, la figura siguiente muestra los botones de la barra de herramientas anterior, pero ahora distribuidos en dos barras de herramientas.

```
<ToolBarTray>
  <ToolBar>
    ...
```

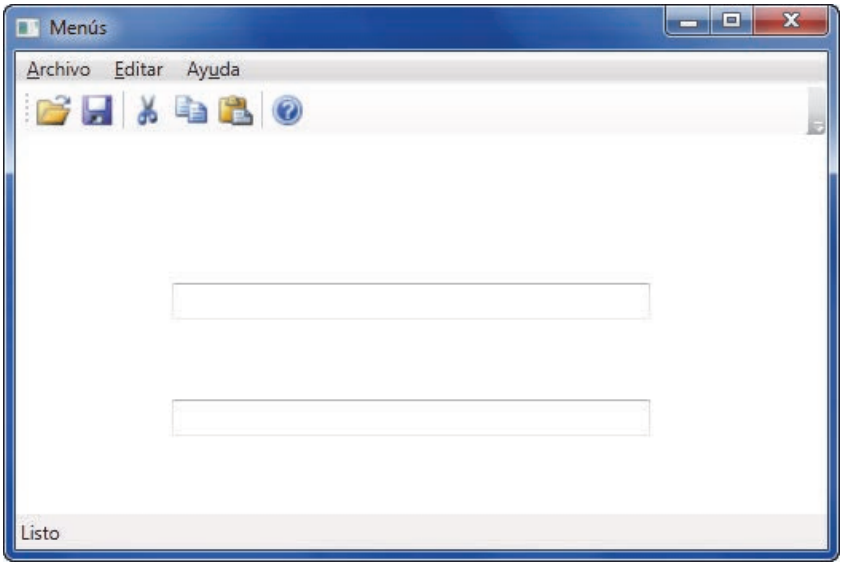
```
  </ToolBar>
  <ToolBar>
    ...
  </ToolBar>
</ToolBarTray>
```



El control **ToolBarTray** tiene una propiedad **Orientation** que permite establecer la orientación vertical (se acopla al lado izquierdo de la ventana) u horizontal (se acopla arriba) del contenedor.

BARRA DE ESTADO

Una barra de estado es un objeto que se visualiza normalmente en la parte inferior de un formulario. Contiene normalmente etiquetas o bloques de texto que muestran diferentes tipos de información al usuario, aunque también puede contener otros controles como, por ejemplo, botones o barras de progreso; en general, puede contener cualquier clase de objetos. Por ejemplo, la ventana de la figura siguiente muestra una barra de estado al fondo de la misma.



Diseño de una barra de estado

Desde el punto de vista de la jerarquía de clases de la biblioteca WPF, una barra de estado es un objeto **StatusBar** que actúa como contenedor para los objetos de las clases **Label**, **TextBox**, **TextBlock**, **ProgressBar**, etc.

En comparación con la barra de herramientas, la barra de estado puede albergar cualquier contenido (que lo envuelve implícitamente en objetos **StatusBarItem**), y anula los estilos predeterminados de algunos elementos para proporcionar una representación más adecuada, pero no tiene soporte para reorganizar arrastrando o para un menú de desbordamiento. Se utiliza sobre todo para mostrar indicadores de texto y de imagen.

Como ejemplo, vamos a añadir a la aplicación *Menus* anterior una barra de estado con un control de tipo **TextBlock**, según muestra la figura anterior, con la intención de mostrar en ella mensajes acerca de la función que realiza la orden que se seleccione de un menú.

Para colocar la barra de estado en el fondo de la ventana vamos a añadir una nueva fila a la rejilla (la fila 2) en la que colocaremos dicha barra:

```
<Window ...>
  ...
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <StackPanel Grid.Row="0" >
      ...
    </StackPanel>
    ...
    <!-- Barra de estado -->
  </Grid>
</Window>
```

Para añadir la barra de estado, arrastre desde la caja de herramientas de Visual Studio un control **StatusBar** sobre el **Grid** y póngale como nombre *BarraDeEstado*. A continuación, asigne a su propiedad asociada **Grid.Row** el valor 2. El resto de las propiedades déjelas con sus valores por omisión. Después de estas acciones podrá comprobar que el asistente para diseño de formularios añadió a la clase *MainWindow* el siguiente código XAML:

```
<StatusBar Grid.Row="2" Name="BarraDeEstado">
</StatusBar>
```

Finalmente, coloque sobre la barra de estado un control **TextBlock** con el contenido "Listo" denominado *btexbarestPpal*. Para ello, arrástrelo desde la caja de herramientas de Visual Studio o escriba directamente el código XAML:

```
<StatusBar Grid.Row="2" Name="BarraDeEstado">
  <TextBlock Name="btexbarestPpal">Listo</TextBlock>
</StatusBar>
```

Obsérvese que inicialmente el bloque de texto *btexbarestPpal* muestra el mensaje "Listo". Ahora, para que muestre información acerca de la función que realiza la orden que se seleccione de un menú, habrá que controlar cuándo el ratón entra en el elemento del menú (evento **MouseEnter**) y cuándo sale (evento **MouseLeave**). Por ejemplo, supongamos que cuando el ratón entra en la orden *Salir* deseamos que la barra de estado muestre el mensaje "Cierra la aplicación" y cuando sale, "Listo". Siguiendo lo enunciado, añada los controladores para los eventos indicados de *Salir* y complételos como se indica a continuación:

```
Private Sub ArchivoSalir_MouseEnter(sender As System.Object,
e As System.Windows.Input.MouseEventArgs)
  Me.btexbarestPpal.Text = "Cierra la aplicación"
End Sub

Private Sub ArchivoSalir_MouseLeave(sender As System.Object,
e As System.Windows.Input.MouseEventArgs)
  Me.btexbarestPpal.Text = "Listo"
End Sub
```

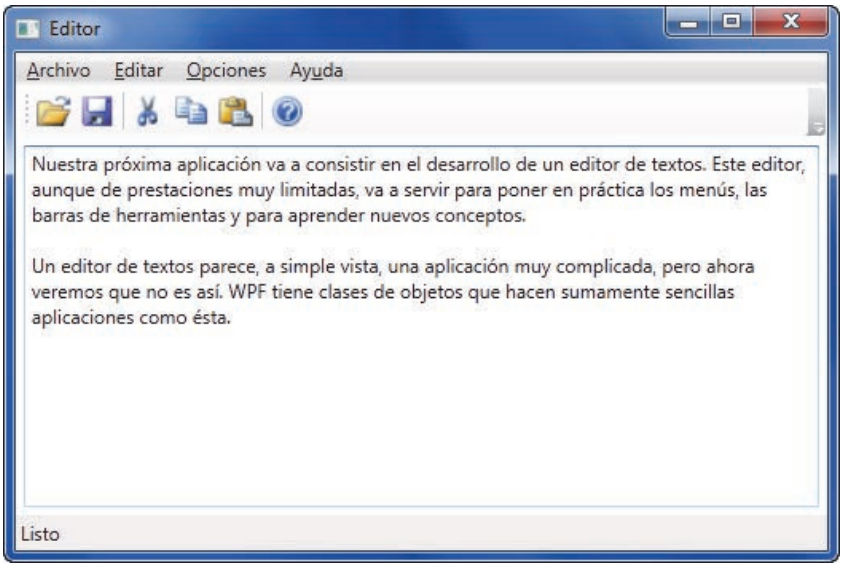
Compile la aplicación, ejecútela y pruebe los resultados. Como ejercicio, puede proceder de forma análoga con el resto de las órdenes.

DESARROLLO DE UN EDITOR DE TEXTOS

Nuestra próxima aplicación va a consistir en el desarrollo de un editor de textos. Este editor, aunque de prestaciones muy limitadas, va a servir para poner en práctica los menús, las barras de herramientas y para aprender nuevos conceptos.

Un editor de textos parece, a simple vista, una aplicación muy complicada, pero ahora veremos que no es así. WPF tiene clases de objetos que hacen sumamente sencillas aplicaciones como ésta.

Utilizando una definición sencilla, un editor de textos es una caja de texto con múltiples líneas, y WPF soporta este tipo de cajas.



Caja de texto multilínea

De forma predeterminada, el control **TextBox** muestra una sola línea de texto y no muestra barras de desplazamiento. Entonces, si la longitud del texto es mayor que el espacio disponible, sólo estará visible parte del texto. Este comportamiento predeterminado se puede cambiar estableciendo algunas de las propiedades **AcceptsReturn**, **AcceptsTab**, **TextWrapping** y **VerticalScrollBarVisibility** con los valores apropiados.

La propiedad **AcceptsReturn** tiene como valor predeterminado **False**. En este caso, el control sólo puede contener una línea de texto; en caso contrario, cuando vale **True**, el control puede contener varias líneas de texto, ya que un valor **True** para esta propiedad permite la utilización de la tecla *Entrar*.

La propiedad **AcceptsTab** tiene como valor predeterminado **False**. En este caso, el control no permite la inserción del carácter de tabulación horizontal (tecla *Tab*); en caso contrario, cuando vale **True**, el control de edición de texto admite el carácter de tabulación.

La propiedad **TextWrapping** tiene como valor predeterminado *NoWrap*. En este caso, el texto del control aparecerá en uno o más párrafos si el control admite la tecla *Entrar*, pero las líneas no se partirán automáticamente, quedando oculta la parte de las mismas que quede fuera de los bordes de la caja de texto, con lo que es recomendable utilizar una barra de desplazamiento horizontal. En cambio, si esta propiedad vale *Wrap*, las líneas de texto se partirán automáticamente al alcanzar el borde derecho de la caja.

Las propiedades **VerticalScrollBarVisibility** y **HorizontalScrollBarVisibility** permiten mostrar en la caja de texto una barra de desplazamiento vertical y/u horizontal, respectivamente. De forma predeterminada no se muestra ninguna.

Diseño del editor

Inicie Visual Basic y cree un nuevo proyecto para implementar el esqueleto para una nueva aplicación WPF que utilice un objeto de la clase *MainWindow* derivada de **Window** como ventana principal. Asigne a esta ventana el título *Editor*. Esta aplicación se localiza en la carpeta *Cap03\Editor* del CD.

Esta ventana, según la figura anterior, contendrá una barra de menús, otra de herramientas, una caja de texto multilínea y una barra de estado. Según esto, vamos a configurar la rejilla con tres filas. La primera fila contendrá un **StackPanel** para albergar las barras de menús y de herramientas, la segunda fila contendrá la caja de texto y la tercera, la barra de estado.

```
<!-- En MainWindow.xaml -->
<Window x:Class="MainWindow"
        xmlns="..."
        xmlns:x="..."
        Title="Editor" Height="350" Width="525">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

  </Grid>
</Window>
```

Observe que la altura de las filas 0 y 2 de la rejilla se ajustará automáticamente al contenido. La altura de la fila 1 (por omisión *) será el espacio restante disponible (para más detalles sobre la rejilla, véase el capítulo *Aplicación WPF*).

Añada a la rejilla de la ventana una barra de menús, *BarraDeMenus*, una barra de herramientas, *BarraDeHerramientas*, y una barra de estado, *BarraDeEstado*. Después añada una caja de texto, objeto de la clase **TextBox**, denominada *ctEditor*, que admita múltiples líneas y que muestre una barra de desplazamiento vertical cuando sea necesaria. Distribuya estos controles en la rejilla según se indicó en el párrafo anterior.

```
<Grid>
  <Grid.RowDefinitions>
```

```
    <RowDefinition Height="Auto" />
    <RowDefinition />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>

  <StackPanel Grid.Row="0" >
    <Menu Name="BarraDeMenus" >
    </Menu>
    <ToolBar Name="BarraDeHerramientas" >
    </ToolBar>
  </StackPanel>

  <TextBox Grid.Row="1" Name="ctEditor" AcceptsReturn="True"
           TextWrapping="Wrap" AcceptsTab="True"
           VerticalScrollBarVisibility="Auto" Margin="5">
  </TextBox>

  <StatusBar Grid.Row="2" Name="BarraDeEstado">
  </StatusBar>
</Grid>
```

Obsérvese que la caja de texto ocupará toda el área de trabajo que quede libre después de colocar las barras.

Guarde la aplicación, compílela y ejecútela. Pruebe a escribir texto, actúe sobre la barra de desplazamiento, modifique el texto, inserte texto, seleccione texto y borre texto. Como podrá comprobar, todas estas operaciones están implícitas sin escribir nada de código. Esto es así porque, según expusimos anteriormente, la clase **TextBox** tiene un enlace integrado con cada orden enrutada **Cut**, **Copy** y **Paste** y encapsula automáticamente el control del portapapeles para esas órdenes.

El portapapeles

El portapapeles permite transferir datos entre componentes, o bien entre aplicaciones. Por ejemplo, muchas aplicaciones que manejan texto incluyen un menú *Editar* con las órdenes *Cortar*, *Copiar* y *Pegar*. Cuando el usuario selecciona texto y ejecuta *Cortar* o *Copiar*, la aplicación transfiere los datos seleccionados en el documento al portapapeles y cuando ejecuta la orden *Pegar*, los transfiere del portapapeles al documento. Los datos seguirán en el portapapeles hasta que se ejecute la orden *Cortar* o *Copiar* nuevamente, instante en el que serán reemplazados por los nuevos seleccionados.

La opción más sencilla para llevar a cabo esas operaciones es utilizar el portapapeles estándar del sistema que en .NET está representado por un objeto de la clase **Clipboard** del espacio de nombres **System.Windows**.

Clase Clipboard

La clase **Clipboard** proporciona varios métodos estáticos para facilitar la transferencia de datos entre el portapapeles del sistema y un control. Así, para colocar datos en el objeto **Clipboard** utilizaremos alguno de los métodos **SetText**, **SetData**, **SetFileDropList**, **SetImage** o **SetAudio**, dependiendo del formato de los mismos. Por ejemplo:

```
Clipboard.SetText("Esto es una cadena de texto")
```

Para obtener los datos colocados en el objeto **Clipboard** utilizaremos alguno de los métodos **GetText**, **GetData**, **GetFileDropList**, **GetImage** o **GetAudioStream**, dependiendo del formato de los mismos. Por ejemplo:

```
textBox1.Text = Clipboard.GetText()
```

Para limpiar el portapapeles utilizaremos el método **Clear**. Por ejemplo:

```
Clipboard.Clear()
```

Los datos que podemos almacenar en el objeto **Clipboard** pueden estar en cualquier formato. Para determinar este formato podemos utilizar alguno de los métodos **ContainsText**, **ContainsData**, **ContainsFileDropList**, **ContainsImage** o **ContainsAudio**. Por ejemplo:

```
If Clipboard.ContainsImage() = True Then
  Debug.WriteLine("El portapapeles contiene una imagen")
End If
```

Finalmente, el siguiente ejemplo muestra cómo guardar el contenido de un fichero de audio en el portapapeles:

```
Dim audio As Byte()
audio = System.IO.File.ReadAllBytes("miFichero.wav")
Clipboard.SetAudio(audio)
```

Manipular la selección del texto

En cualquier editor de texto el usuario puede seleccionar texto utilizando el ratón o el teclado. Para hacerlo con el ratón, hay que apuntar al comienzo del texto a seleccionar y arrastrar con el botón izquierdo pulsado hasta haber seleccionado el texto deseado. Con el teclado, sitúe el punto de inserción donde desea iniciar la selección y, manteniendo pulsada la tecla *Shift* (mayúsculas), desplace el punto de inserción utilizando las teclas de desplazamiento. Los controles de texto de .NET

derivados de la clase **System.Windows.Controls.TextBoxBase**, como **TextBox**, ya incorporan esta característica.

Además de la capacidad de selección, la clase **TextBoxBase** proporciona una serie de propiedades y métodos que permiten trabajar con el texto seleccionado a través del portapapeles, lo cual evita tener que recurrir a la clase **Clipboard** expuesta en el apartado anterior. Algunas de estas propiedades y métodos se exponen a continuación.

Propiedades:

SelectedText. Permite obtener el texto seleccionado, o bien reemplazar el texto seleccionado (puede ser nulo) por otro.

SelectionStart. Permite obtener o establecer el punto de inicio de la selección actual.

SelectionLength. Permite obtener o establecer el número de caracteres seleccionados en la caja de texto.

CanUndo. Permite saber si el usuario puede deshacer la operación anterior realizada en un control de texto.

Métodos:

Sub Copy(). Copia el texto actualmente seleccionado en el control de texto en el portapapeles.

Sub Cut(). Mueve la selección actual del control de texto al portapapeles.

Sub Paste(). Reemplaza la selección actual del control de texto con el contenido del portapapeles.

Sub Clear(). Borra todo el texto del control de texto.

Sub SelectAll(). Selecciona todo el texto del control de texto.

Sub Select(*pos_inicial* **As Integer**, *pos_final* **As Integer)**. Selecciona el texto que se encuentra entre las posiciones especificadas.

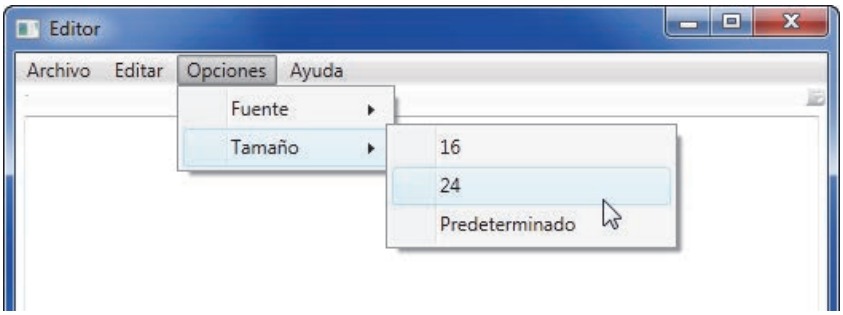
Sub AppendText(*texto* **As String)**. Añade texto al contenido actual del control de texto.

Sub Undo(). Permite deshacer las operaciones de edición.

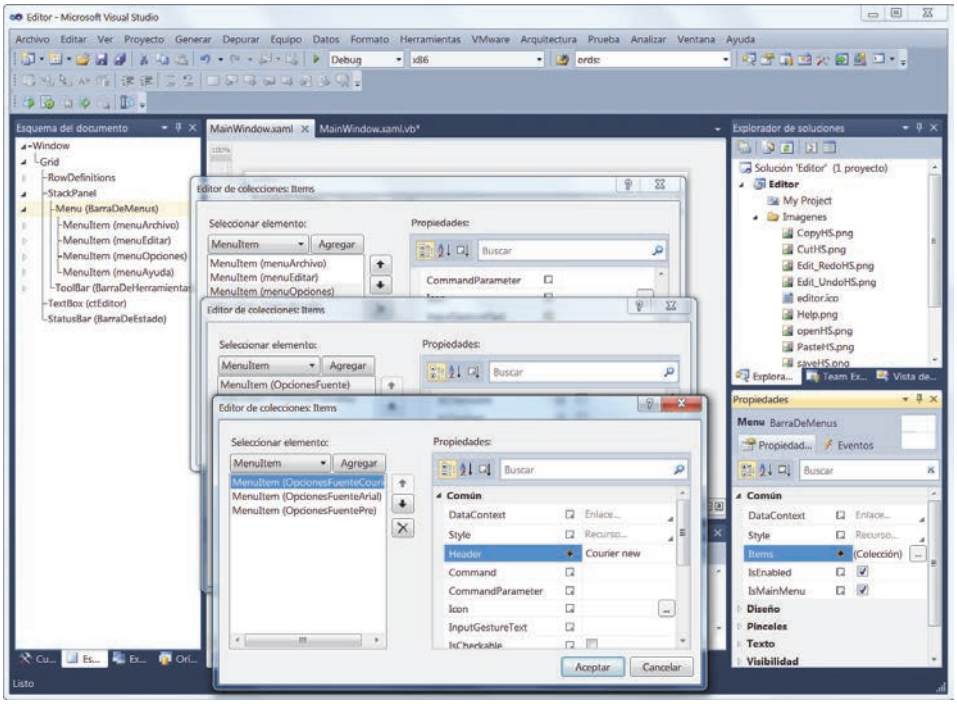
Sub Redo(). Permite rehacer las operaciones de edición.

Diseño de la barra de menús

Siguiendo con el diseño de la aplicación *Editor*, y aplicando la teoría expuesta anteriormente en este mismo capítulo, cree los menús *Archivo*, *Editar*, *Opciones* y *Ayuda*. Para este planteamiento inicial, suponga que el menú *Archivo* está formado por las órdenes *Abrir*, *Guardar* y *Salir*, el menú *Editar*, por las órdenes *Cortar*, *Copiar* y *Pegar*, el menú *Opciones*, por los submenús *Fuente* y *Tamaño*, y el menú *Ayuda*, por la orden *Acerca de*. A su vez, el submenú *Fuente* mostrará las órdenes *Courier New*, *Arial* y *Predeterminada* y el submenú *Tamaño*, las órdenes *16*, *24* y *Predeterminado*. Especifique también los nemónicos, los aceleradores y asigne las imágenes para aquellas órdenes que considere de interés. Las imágenes estarán almacenadas en una carpeta, *Imágenes*, que añadirá al proyecto.



La siguiente figura muestra cómo se utiliza la ventana de propiedades y el diseñador en su conjunto, para diseñar el menú *Opciones* partiendo del objeto barra de menús seleccionado en la ventana *Esquema del documento*:



Como ejemplo, analizaremos parte del código que el diseñador de formularios añadió para implementar la barra de menús, los menús de la barra y las órdenes de los menús:

```
<Menu Name="BarraDeMenus" >
  <MenuItem Name="menuArchivo" Header="_Archivo">
    <MenuItem Name="ArchivoAbrir" Header="A_brir...">
      <MenuItem.Icon>
        <Image Source="Imagenes/openHS.png" />
      </MenuItem.Icon>
    </MenuItem>
    ...

  <MenuItem Name="menuEditar" Header="_Editar">
    ...
  </MenuItem>

  <MenuItem Name="menuOpciones" Header="_Opciones">
    <MenuItem Name="OpcionesFuente" Header="_Fuente">
      <MenuItem Name="OpcionesFuenteCourier" Header="Courier New"/>
      <MenuItem Name="OpcionesFuenteArial" Header="Arial"/>
      <MenuItem Name="OpcionesFuentePre" Header="Predeterminada"/>
    </MenuItem>

    <MenuItem Name="OpcionesTamaño" Header="_Tamaño">
```

```
      <MenuItem Name="OpcionesTamaño16" Header="16"/>
      <MenuItem Name="OpcionesTamaño24" Header="24"/>
      <MenuItem Name="OpcionesTamañoPre" Header="Predeterminado"/>
    </MenuItem>
  </MenuItem>

  <MenuItem Name="menuAyuda" Header="Ay_uda">
    <MenuItem Name="AyudaAcercaDe" Header="_Acerca de..." />
  </MenuItem>
</Menu>
```

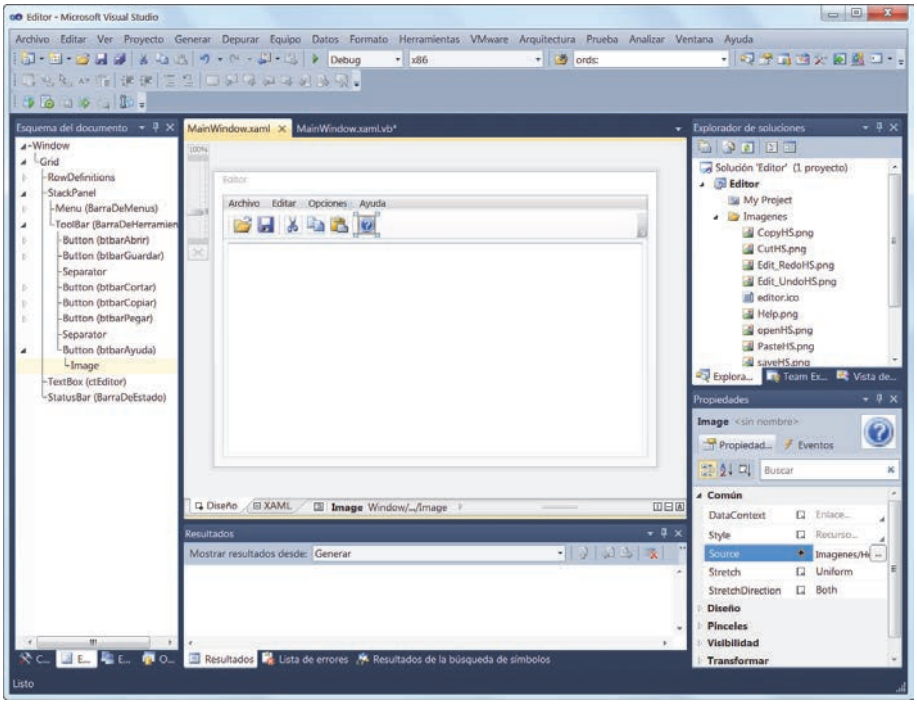
Analizando el código anterior, observamos que:

1. Inicialmente se definen los objetos *BarraDeMenus*, *menuAbrir*, *menuEditar*, *menuOpciones* y *menuAyuda* que darán lugar a la barra de menús.
2. Después, se completa el menú *menuArchivo* con sus órdenes correspondientes. Observando la orden *Abrir* vemos que la propiedad **Name** define el identificador *ArchivoAbrir* que utilizaremos en el código para referirnos a esta orden y que la propiedad **Header** define el texto que se mostrará cuando se abra el menú. También se ha asignado a su propiedad **Icon** un objeto **Image** que establece la imagen para dicha orden. El resto de las propiedades tiene el valor asignado por omisión; por ejemplo, las propiedades **Width** y **Height** tienen asignado el valor *Auto* para que el ajuste de tamaño sea automático.
3. El resto de los menús y sus órdenes se construyen de manera análoga.

Diseño de la barra de herramientas

Continuando con el diseño de la aplicación *Editor*, añada a la barra de herramientas los botones: *Abrir*, *Guardar*, *Cortar*, *Copiar*, *Pegar* y *Ayuda*.

Cada uno de los botones es un objeto de la clase **Button** que puede arrastrar desde la caja de herramientas de Visual Studio a la barra de herramientas de la aplicación. Las propiedades para cada uno de los botones puede establecerlas a través de la ventana de propiedades; la mayoría de ellas permanecerán con los valores asignados por omisión; por ejemplo, las propiedades **Width** y **Height** tienen asignado por omisión el valor *Auto* para que el ajuste de tamaño sea automático. En nuestro caso, como se puede ver un poco más adelante, hemos establecido la propiedad **Name** para especificar el identificador del botón y la propiedad **Content** para que el contenido a mostrar sea el objeto **Image** especificado.



Finalizado el diseño de la barra de herramientas, podemos observar el código XAML generado que dará lugar a la construcción de la misma:

```
<ToolBar Name="BarraDeHerramientas" >
  <Button Name="btbarAbrir" >
    <Image Source="Imagenes/openHS.png"/>
  </Button>
  <Button Name="btbarGuardar" >
    <Image Source="Imagenes/saveHS.png"/>
  </Button>
  <Separator></Separator>
  <Button Name="btbarCortar" >
    <Image Source="Imagenes/CutHS.png"/>
  </Button>
  <Button Name="btbarCopiar" >
    <Image Source="Imagenes/CopyHS.png"/>
  </Button>
  <Button Name="btbarPegar" >
    <Image Source="Imagenes/PasteHS.png"/>
  </Button>
  <Separator></Separator>
  <Button Name="btbarAyuda" >
    <Image Source="Imagenes/Help.png"/>
  </Button>
</ToolBar>
```

Asignar a un elemento de la interfaz la tarea a realizar

Una vez diseñada la interfaz, vincularemos con cada una de las órdenes de los menús el código que se tiene que ejecutar cuando sean seleccionadas. Anteriormente, en este mismo capítulo, ya expusimos cómo implementar un controlador para responder al evento **Click** generado por una orden de un menú después de haber hecho clic sobre ella. Otra opción, que también vimos, es vincular una determinada orden con una orden enrutada, predeterminada o personalizada.

Con respecto a los botones de la barra de herramientas, el proceso a seguir para implementar un controlador que responda a su evento **Click** ya fue explicado en el capítulo *Introducción a WPF*, y coincide con el seguido para una orden de un menú.

Archivo - Salir

Para asociar un controlador de eventos **Click** con la orden *Salir* del menú *Archivo*, seleccione el elemento *ArchivoSalir* en la ventana *Esquema del documento*, haga clic en el botón *Eventos* de la ventana *Propiedades* correspondiente al elemento seleccionado, seleccione el evento **Click** y haga doble clic sobre el mismo. Esta acción vinculará el evento **Click** del elemento seleccionado con el método *ArchivoSalir_Click* que será añadido en el fichero de código Visual Basic subyacente.

```
<MenuItem Name="ArchivoSalir" Header="_Salir"
          Click="ArchivoSalir_Click" />
```

Después, complete el método añadido como se muestra a continuación. El resultado será el siguiente:

```
Private Sub ArchivoSalir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Close()
End Sub
```

El resto de las órdenes que expondremos a continuación, se escriben de forma análoga o, como veremos a continuación, utilizando órdenes enrutadas de la biblioteca WPF.

Editar - Cortar

En este caso, cuando el usuario haga clic en la orden *Cortar*, queremos que se ejecute la orden **ApplicationCommands.Cut**. Esta orden corta el texto que haya seleccionado en el objeto *ctEditor* y lo envía al portapapeles.

Para vincular la orden **Cut** con el elemento *EditarCortar*, selecciónelo en la ventana *Esquema del documento*, seleccione su propiedad **Command** en la ventana *Propiedades* y elija en la lista de órdenes adjunta la orden **Cut**. El resultado de esta operación puede verse en el código mostrado a continuación:

```
<MenuItem Name="EditarCortar" Command="ApplicationCommands.Cut">
  <MenuItem.Icon>
    <Image Source="Imagenes/CutHS.png" />
  </MenuItem.Icon>
</MenuItem>
```

La propiedad **Command** especifica la orden que se ejecutará cuando el usuario haga clic en el elemento *EditarCortar*. Como dicho elemento no establece su propiedad **CommandTarget**, el destino de la orden será el elemento que tenga el foco, que en nuestra aplicación será la caja de texto.

El valor para las propiedades **Header** (texto "Copiar") e **InputGestureText** (texto del acelerador "Ctrl+X") es proporcionado automáticamente por **Cut**.

En este caso no es necesario implementar el objeto **CommandBinding**, enlace de la orden, porque también está integrado en la implementación de la clase **TextBox**, lo mismo que está integrado el control del portapapeles.

El botón de la barra de herramientas identificado por *btbarCortar* debe realizar la misma acción. Por lo tanto, debemos vincularlo también con la orden **Cut**. Para ello, proceda de forma análoga a como hizo con el elemento *EditarCortar*:

```
<Button Name="btbarCortar" Command="ApplicationCommands.Cut">
  <Image Source="Imagenes/CutHS.png"/>
</Button>
```

Editar - Copiar

La orden *Copiar* se diferencia de la orden *Cortar* en que no borra el texto seleccionado. Análogamente a como hicimos con la orden *Cortar*, vincule el elemento *EditarCopiar* con la orden **ApplicationCommands.Copy**. Esta orden copia el texto que haya seleccionado en el objeto *ctEditor* y lo envía al portapapeles.

```
<MenuItem Name="EditarCopiar" Command="ApplicationCommands.Copy">
  <MenuItem.Icon>
    <Image Source="Imagenes/CopyHS.png" />
  </MenuItem.Icon>
</MenuItem>
```

Las explicaciones para esta orden son análogas a las dadas en la orden *Cortar*.

El botón de la barra de herramientas identificado por *btbarCopiar* debe realizar la misma acción. Por lo tanto, debemos vincularlo también con la orden **Copy**. Para ello, proceda de forma análoga a como hizo con el botón *btbarCortar*:

```
<Button Name="btbarCopiar" Command="ApplicationCommands.Copy">
  <Image Source="Imagenes/CopyHS.png"/>
</Button>
```

Editar - Pegar

Cuando el usuario haga clic sobre la orden *Pegar*, se colocará el texto del portapapeles sobre el texto seleccionado en *ctEditor* o en su defecto, a partir de la posición del punto de inserción. Para implementar esta orden, análogamente a como hicimos con las órdenes *Cortar* o *Copiar*, vincule el elemento *EditarPegar* con la orden **ApplicationCommands.Paste**.

```
<MenuItem Name="EditarPegar" Command="ApplicationCommands.Paste">
  <MenuItem.Icon>
    <Image Source="Imagenes/PasteHS.png" />
  </MenuItem.Icon>
</MenuItem>
```

Las explicaciones para esta orden son análogas a las dadas en la orden *Cortar*.

El botón de la barra de herramientas identificado por *btbarPegar* debe realizar la misma acción. Por lo tanto, debemos vincularlo también con la orden **Paste**. Para ello, proceda de forma análoga a como hizo con los botones anteriores:

```
<Button Name="btbarPegar" Command="ApplicationCommands.Paste">
  <Image Source="Imagenes/PasteHS.png"/>
</Button>
```

Opciones - Fuente

A continuación, vamos a escribir el código para los elementos del menú *Opciones*. Este menú visualiza dos submenús: *Fuente*, que presenta tres elementos que permiten cambiar el tipo de letra, y *Tamaño*, que presenta otros tres elementos que permiten cambiar el tamaño de la letra.

El número de fuentes disponibles en Visual Basic varía en función de la configuración de su sistema operativo. En nuestro editor sólo vamos a incluir tres tipos: *Courier New*, *Arial* y el tipo de letra establecido de forma predeterminada.

En una caja de texto de la clase **TextBox** solamente puede utilizarse una fuente en todo el documento, lo cual significa que cuando se modifique el tipo de letra

para la caja de texto, todo el texto cambiará a ese tipo de letra. Por lo tanto, para cambiar la fuente en el editor de textos, simplemente tenemos que modificar las características de la fuente actual de la caja de texto por medio de sus propiedades **Font…** (por ejemplo **FontFamily** o **FontSize**).

Según lo expuesto, añada en *MainWindow* un controlador de eventos que responda al evento **Click** de la orden *Courier New* estableciendo la nueva fuente:

```
Private Sub OpcionesFuenteCourier_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontFamily = New FontFamily("Courier New")
End Sub
```

Este método crea la nueva familia de fuente *Courier New* y se la aplica a la caja de texto *ctEditor* por medio de su propiedad **FontFamily**.

Para el objeto *OpcionesFuenteArial* proceda de forma análoga; esto es, añada su controlador de eventos **Click** y escríbalo así:

```
Private Sub OpcionesFuenteArial_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontFamily = New FontFamily("Arial")
End Sub
```

A continuación, para poder restablecer la fuente con la que se inició la aplicación, vamos a almacenar en una variable *familiaPre* de tipo **FontFamily** la familia correspondiente a la fuente predeterminada de la caja de texto y en otra *tamañoPre* de tipo **Double**, el tamaño predeterminado. Para ello, añada estas variables como atributos de la clase *MainWindow*:

```
Class MainWindow
  Private familiaPre As FontFamily
  Private tamañoPre As Double
  ...
End Class
```

Después, en el instante en el que se vaya a presentar la ventana que da lugar al editor (evento **Loaded**), obtenemos la familia y el tamaño de la fuente predeterminada con las que se ha creado la caja de texto y las almacenamos en esos atributos. Por lo tanto, añada el controlador para el evento **Loaded** de la ventana *MainWindow* y complételo como se indica a continuación:

```
Private Sub Window_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  familiaPre = ctEditor.FontFamily
  tamañoPre = ctEditor.FontSize
```

```
End Sub
```

Finalmente, añada a la clase *MainWindow* un controlador de eventos que responda al evento **Click** de la orden *Predeterminada* del menú *Fuente* para que aplique la fuente predeterminada a la caja de texto:

```
Private Sub OpcionesFuentePre_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontFamily = familiaPre
End Sub
```

Además del control **TextBox**, la biblioteca WPF proporciona otro denominado **RichTextBox** que permite establecer el tipo de fuente a nivel de carácter.

Opciones - Tamaño

Análogamente a como sucede con la familia de la fuente, ocurre con el tamaño de la misma. Cuando se modifique el tamaño de la fuente de la caja de texto, todo el texto cambiará a ese tamaño. Por lo tanto, para modificar el tamaño del texto que muestra el editor de textos, simplemente tenemos que asignar a la propiedad **FontSize** de la caja de texto *ctEditor* el nuevo tamaño.

Según lo expuesto, añada a la clase *MainWindow* un controlador de eventos que responda al evento **Click** del objeto *OpcionesTamaño16*, estableciendo el tamaño 16:

```
Private Sub OpcionesTamaño16_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontSize = 16.0
End Sub
```

Para el objeto *OpcionesTamaño24* proceda de forma análoga:

```
Private Sub OpcionesTamaño24_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontSize = 24.0
End Sub
```

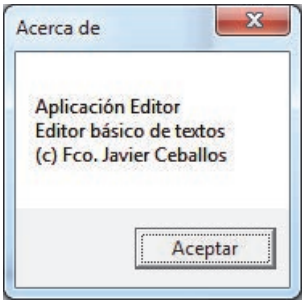
Finalmente, para poder restablecer el tamaño de la fuente con la que se inició la aplicación, utilizaremos el valor almacenado en el atributo *tamañoPr* de la clase *MainWindow*. Para ello, añada a esta clase un controlador de eventos que responda al evento **Click** del objeto *OpcionesTamañoPre* y complételo como se indica a continuación:

```
Private Sub OpcionesTamañoPre_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
```

```
    ctEditor.FontSize = tamañoPre
  End Sub
```

Ayuda – Acerca de

La orden *Acerca de* del menú *Ayuda* mostrará una ventana con los créditos de la aplicación.



En principio, esta ventana será mostrada invocando al método **MessageBox.Show** con dos argumentos: el mensaje a mostrar y el título de la ventana; más adelante, en otro capítulo aprenderemos a personalizar estas cajas de diálogo.

La ventana se mostrará cuando el usuario haga clic en el elemento *AyudaAcercaDe* o en el botón correspondiente de la barra de herramientas. Por lo tanto, añada a la clase *MainWindow* un controlador de eventos que maneje el evento **Click** del objeto *AyudaAcercaDe* y complételo con el código siguiente:

```
Private Sub AyudaAcercaDe_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim créditos As String = "Aplicación Editor" & vbLf &
                           "Editor básico de textos" & vbLf &
                           "(c) Fco. Javier Ceballos"
  MessageBox.Show(créditos, "Acerca de")
End Sub
```

El botón de la barra de herramientas identificado por *btbarAyuda* debe realizar la misma acción que *AyudaAcercaDe*. Por lo tanto, debemos vincularlo también con el controlador *AyudaAcercaDe_Click*. Para ello, selecciónelo en la ventana *Esquema del documento*, haga clic en el botón *Eventos* de la ventana *Propiedades*, seleccione el evento **Click** y elija en la lista adjunta el método *AyudaAcercaDe_Click*:

```
<Button Name="btbarAyuda" Click="AyudaAcercaDe_Click">
  <Image Source="Imagenes/Help.png"/>
</Button>
```

Eventos comunes a todos los elementos WPF

Cualquier elemento WPF derivado de **FrameworkElement** o **FrameworkContentElement** durante su ciclo de vida pasa por tres fases: creación, uso y destrucción, en el que se suceden estos tres eventos: **Initialized**, **Loaded** y **Unloaded**.

El evento **Initialized** es el primero que se genera después de la llamada al constructor. Por lo tanto, es seguro que se establecen todas las propiedades del objeto. En esta fase no se evalúan las expresiones relativas a los recursos dinámicos o a los enlaces y tampoco existe ninguna garantía de que se hayan creado todos los demás elementos del árbol de elementos.

A continuación se genera el evento **Loaded**. Este evento tiene lugar antes de la presentación final del elemento, pero después de que se hayan calculado todos los valores necesarios para la presentación. A diferencia de **Initialized**, el evento **Loaded** se genera en primer lugar en el elemento raíz, cuando todos los elementos del árbol están en un estado en que se consideran cargados, y a continuación se genera en cada uno de los elementos secundarios, por lo tanto, cada elemento siempre tiene la oportunidad de controlar su evento **Loaded**.

El evento **Unloaded** es el último que se genera. En esta fase hay que tener en cuenta que el elemento primario origen del evento o cualquier otro elemento que se encuentre más arriba en el árbol de elementos ya no esté establecido, lo que significa que los enlaces de datos, las referencias a recursos y los estilos podrían no tener ya su valor normal de ejecución.

Habilitar o inhabilitar los elementos de un menú

Una orden de un menú puede ejecutarse sólo si está habilitada. Esto quiere decir que una aplicación puede habilitar o inhabilitar los elementos de un menú, en función de que tengan o no actividad en el instante en el que el menú es desplegado. Cada vez que se selecciona un menú y se abre se producen los eventos **Initialized**, **Loaded** y **Unloaded**. Por lo tanto, podemos utilizar el evento **Loaded** (se va a presentar el menú) para mostrar habilitada cada orden del menú que pueda desempeñar en este instante su función o, en caso contrario, mostrarla inhabilitada.

Siguiendo con nuestra aplicación, el menú *Editar* tiene tres órdenes: *Cortar*, *Copiar* y *Pegar*. Un pequeño análisis nos puede conducir a la conclusión de que las órdenes *Cortar* y *Copiar* tiene sentido que estén activas si hay un bloque de texto seleccionado; en cambio, la orden *Pegar*, para el ejemplo que nos ocupa, tiene sentido que esté activa sólo si el portapapeles contiene texto. Estas características ya las cumple el menú *Editar* de nuestra aplicación, porque van implícitas

en las órdenes **Cut**, **Copy** y **Paste** definidas en la clase **ApplicationCommands**. Pero supongamos que, en lugar de estas órdenes predeterminadas, hubiéramos implementado los controladores para el evento **Click** de las órdenes *Cortar*, *Copiar* y *Pegar* así (*Cap03\Editor2* del CD):

```
Private Sub EditarCortar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.Cut()
End Sub

Private Sub EditarCopiar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.Copy()
End Sub

Private Sub EditarPegar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.Paste()
End Sub
```

¿Cómo implementaríamos la funcionalidad esperada? Según lo expuesto, la condición para activar las órdenes *Cortar* y *Copiar* es que el usuario haya seleccionado un bloque de texto, lo que puede saberse interrogando el valor de la propiedad **SelectedText** del objeto *ctEditor*:

```
ctEditor.SelectedText.Length != 0
```

Esta propiedad hace referencia al objeto **String** que almacena el texto seleccionado. Si no hay texto seleccionado, su longitud será cero, o lo que es lo mismo, la expresión anterior dará como resultado **False**.

Y la condición para activar la orden *Pegar* es que el portapapeles contenga texto:

```
Clipboard.ContainsText()
```

El método **ContainsText** devolverá **True** si la papelera contiene texto. Dicho contenido puede ser borrado al iniciarse la aplicación (evento **Loaded** de *MainWindow*) ejecutando el método **Clear**:

```
Clipboard.Clear();
```

Para realizar el proceso descrito, añada a *MainWindow* un controlador de eventos que responda al evento **Loaded** generado por cada uno de los elementos del menú *menuEditar*. ¿Qué tiene que hacer este método? Simplemente asignar a la propiedad **IsEnabled** de las órdenes *Cortar* y *Copiar* un valor **True** si hay tex-

to seleccionado (habilitar órdenes), o **False** en caso contrario (inhabilitar órdenes), y asignar a la propiedad **IsEnabled** de la orden *Pegar* un valor **True** si el portapapeles contiene texto, o **False** en caso contrario.

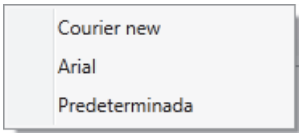
```
Private Sub menuEditar_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs) Handles menuEditar.Loaded
  EditarCortar.IsEnabled = (ctEditor.SelectedText.Length <> 0)
  EditarCopiar.IsEnabled = (ctEditor.SelectedText.Length <> 0)
  EditarPegar.IsEnabled = Clipboard.ContainsText()
End Sub
```

Ídem para los botones correspondientes a estas órdenes en la barra de herramientas. En este caso, las acciones las llevaremos a cabo desde el método que responda al evento **MouseEnter** de la barra de herramientas:

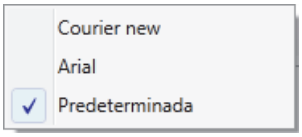
```
Private Sub BarraDeHerramientas_MouseEnter(sender As System.Object,
e As System.Windows.Input.MouseEventArgs)
  btbarCortar.IsEnabled = (ctEditor.SelectedText.Length <> 0)
  btbarCopiar.IsEnabled = (ctEditor.SelectedText.Length <> 0)
  btbarPegar.IsEnabled = Clipboard.ContainsText()
End Sub
```

Marcar el elemento seleccionado de un menú

Los elementos de un menú pueden ser marcados cuando son seleccionados, bien automáticamente poniendo su propiedad **IsCheckable** a **True**, o bien bajo programación por medio de la propiedad **IsChecked**. Esto permite identificar de forma visual el elemento del menú actualmente seleccionado.



No marcado



Marcado

Como ejemplo, vamos a dejar señalada en el menú *Fuente* la fuente que está actualmente seleccionada. Quiere esto decir que cuando el usuario haga clic en un elemento de este menú, éste debe quedar marcado y el que estaba marcado debe dejar de estarlo; esto descarta el uso de la propiedad **IsCheckable**. Por otra parte, cuando arranque la aplicación el elemento *Predeterminada* debe aparecer señalado. Esto puede hacerse durante el diseño, poniendo su propiedad **IsChecked** a valor **True**.

Según lo expuesto, proceda paso a paso como se indica a continuación:

1. Inicie en la ventana de propiedades del elemento *OpcionesFuentePre* su propiedad **IsChecked** con el valor **True**.

```
<MenuItem Name="OpcionesFuentePre" Header="Predeterminada"
          Click="OpcionesFuentePre_Click" IsChecked="True" />
```

2. Durante la ejecución, cuando el usuario seleccione otro tipo de letra, hay que quitar la señal de donde esté y ponerla en el tipo de letra seleccionado. Para realizar esta operación, modifique los controladores del evento **Click** de cada uno de los elementos del menú *Fuente* de la forma que se indica a continuación. Por ejemplo, el controlador de la orden *Courier New* quedaría así:

```
Private Sub OpcionesFuenteCourier_Click(sender As Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.FontFamily = New FontFamily("Courier New")
  OpcionesFuenteCourier.IsChecked = True
  OpcionesFuenteArial.IsChecked = False
  OpcionesFuentePre.IsChecked = False
End Sub
```

Este método se ejecutará cuando se seleccione el elemento *OpcionesFuenteCourier*, el cual quedará señalado porque pone su propiedad **IsChecked** a valor **True**. Así mismo, pone la propiedad **IsChecked** del resto de las órdenes a **False** para quitar la marca que pudiera tener cualquiera de ellas.

Como ejercicio, proceda de forma análoga con los elementos *OpcionesFuenteArial* y *OpcionesFuentePre* y con los elementos del menú *Tamaño*.

Deshacer y rehacer

Implementar la operación de *Deshacer* tiene dos partes: una, recordar las operaciones de edición realizadas para poderlas deshacer; y dos, añadir a la interfaz del usuario las órdenes que realicen ese trabajo. Como ejemplo, vamos a añadir a nuestra aplicación *Editor* estas operaciones.

Recordar las ediciones reversibles

Para que un componente de texto pueda soportar las operaciones de *Deshacer* y *Rehacer* debe recordar cada operación de edición que ha tenido lugar, el orden en el que se han sucedido y lo que supone el deshacerlas o rehacerlas. Para realizar este trabajo, la clase **ApplicationCommands** proporciona las órdenes **Undo** (deshacer) y **Redo** (rehacer), y también la clase **TextBoxBase** proporciona los métodos **Undo** (deshacer) y **Redo** (rehacer) y las propiedades **CanUndo** (¿se puede deshacer?) y **CanRedo** (¿se puede rehacer?). Por lo tanto, puede elegir la opción

que considere. No obstante, ya ha visto las ventajas de utilizar las órdenes predefinidas. A continuación veremos cómo proceder desde ambos puntos de vista.

Añadir a la interfaz la orden Deshacer

El paso siguiente es añadir al menú *Editar* la orden *Deshacer* con su controlador de eventos correspondiente. Para ello, siga los pasos indicados a continuación:

1. Añada al menú *Editar* la orden *Deshacer*. Para ello, seleccione el menú *menuEditar*, su propiedad **Items** y añada a la colección referenciada por esta propiedad un nuevo componente de la clase **MenuItem**; establezca su propiedad **Icon** y su propiedad **Name** (denomínelo *EditarDeshacer*).

2. Asigne a su propiedad **Command** la orden **ApplicationCommands.Undo**.

   Después de realizar los dos pasos anteriores, el código añadido por el asistente será similar al siguiente:

```
<MenuItem Name="EditarDeshacer"
          Command="ApplicationCommands.Undo">
  <MenuItem.Icon>
    <Image Source="Imagenes/Edit_UndoHS.png"/>
  </MenuItem.Icon>
</MenuItem>
```

   La orden **Undo** deshace las operaciones de edición realizadas, empezando por la última.

Estas mismas operaciones se pueden programar utilizando controladores en lugar de órdenes predefinidas. Para ello, siga los pasos indicados a continuación:

1. Añada al menú *Editar* la orden *Deshacer*.

2. Añada a la clase *MainWindow* el controlador que maneje el evento **Click** de la orden *Deshacer* y complételo como se indica a continuación:

```
Private Sub EditarDeshacer_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctEditor.Undo()
End Sub
```

   El método **Undo** deshace las operaciones de edición realizadas, empezando por la última.

Así mismo, para que dicha orden esté habilitada sólo cuando haya operaciones de edición para deshacer, añada el código indicado a continuación al método *menuEditar_Loaded* explicado anteriormente:

```
Private Sub menuEditar_Loaded(sender As System.Object, e As
System.Windows.RoutedEventArgs) Handles menuEditar.Loaded
  EditarDeshacer.IsEnabled = ctEditor.CanUndo
  ...
End Sub
```

El método **CanUndo** devuelve **True** si hay una operación de edición que se puede deshacer y **False** en caso contrario.

Añadir a la interfaz la orden Rehacer

Siguiendo un razonamiento análogo al seguido para añadir la orden *Deshacer*, añada al menú *Editar* la orden *Rehacer*. Para implementar este elemento, utilice la orden **ApplicationCommands.Redo** o los métodos **CanRedo** y **Redo** heredados de **TextBoxBase**.

Menú contextual

Los menús contextuales fueron expuestos anteriormente en el apartado *Menús contextuales* de este mismo capítulo. El control **TextBox** implementa de forma predeterminada un menú contextual con las órdenes *Cortar*, *Copiar* y *Pegar*.

Asociar un icono a la aplicación

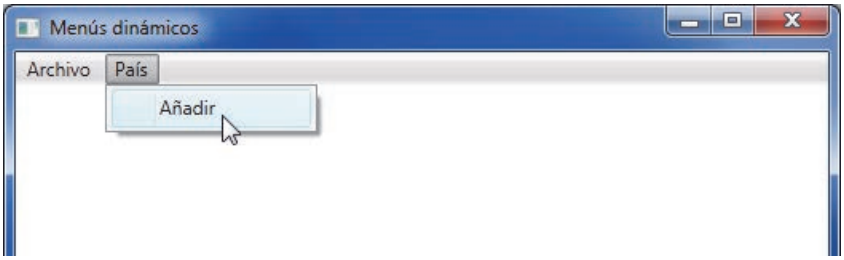
Para asociar un icono a la ventana principal de la aplicación, la clase **Window** proporciona la propiedad **Icon**. Por lo tanto, seleccione la ventana, diríjase a la ventana de propiedades, seleccione la propiedad **Icon** y asígnele el fichero que contiene la imagen. El código XAML que se añadirá será el siguiente:

```
<Window x:Class="MainWindow" ...
        ...
        Icon="/Editor;component/Imagenes/editor.ico">
```

MENÚS DINÁMICOS

Se denomina menú dinámico a aquél cuya colección de elementos puede variar durante la ejecución de la aplicación.

Para saber el código que hay que escribir para añadir un elemento a un menú durante la ejecución, vamos a analizar el código que el diseñador de formularios escribe cuando hacemos esa operación durante el diseño. Para ello, vamos a partir de que tenemos construida una aplicación, *MenusDinamicos*, que muestra la siguiente interfaz gráfica:



La figura anterior muestra una ventana con una barra de menús con dos menús: "Archivo" y "País". Supongamos que el identificador de este último menú es *menuPais*. Añadir a este menú el elemento "Añadir" implica:

1. Crear el elemento de tipo **MenuItem**:

```
Dim PaisAnadir As MenuItem
PaisAnadir = New MenuItem()
```

2. Establecer sus propiedades, o bien aceptar los valores predeterminados.

```
PaisAnadir.Header = "Añadir"
```

3. Añadir el elemento a la colección referenciada por la propiedad **Items** de *menuPais*:

```
menuPais.Items.Add(PaisAnadir)
```

La propiedad **Items** hace referencia, en este caso, a la colección de elementos del objeto **MenuItem** a la que podremos añadir elementos (**Add** para añadir un elemento), o bien de la que podremos eliminar elementos (**Remove** o **RemoveAt**), operaciones necesarias para trabajar con un menú dinámico. En general, la propiedad **Items** es una propiedad heredada de **ItemsControl** (véase el capítulo *Introducción a WPF*). Un **ItemsControl** es un tipo de control (por ejemplo, **MenuItem**, **ToolBar**, **ListBox**, etc.) que puede contener varios elementos, tales como cadenas u otros objetos (como **DateTime**, **UIElement**, etc.).

En el código anterior se observa que el método **Add** añade a la colección de elementos de *menuPais* un nuevo elemento referenciado por *PaisAnadir*. Ahora bien, cuando trabajemos con menús dinámicos los elementos a añadir hay que

crearlos dinámicamente (durante la ejecución), además de asociarlos con el controlador que responda a su evento **Click**.

Quiere esto decir que también hay que escribir el controlador que se vaya a utilizar para responder al evento **Click** de los elementos que añadamos durante la ejecución. El controlador será de la forma que se indica a continuación. Vamos a añadirlo a la clase *MainWindow*:

```
Private Sub ElementoMenuPais_Click(sender As Object, e As RoutedEventArgs)
  ' ...
End Sub
```

Una vez que sabemos cómo crear un nuevo elemento y añadirlo a un menú, vamos a escribir el controlador del evento **Click** del elemento "Añadir" para que permita añadir dinámicamente elementos al menú "País". Los títulos de los elementos se corresponderán con nombres de países (en este ejemplo, estos nombres los simularemos de la forma "País #").

Para probar cómo funciona la aplicación, vamos a hacer que *ElementoMenuPais_Click* muestre el título del elemento sobre el que hagamos clic. Finalmente, añadiremos el controlador de "Salir" para salir de la aplicación. El resultado, una vez realizado todo el proceso descrito, se muestra a continuación:

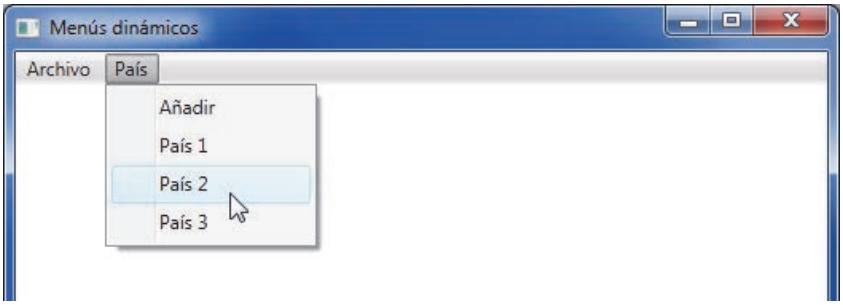
```
Class MainWindow
  Private Sub ElementoMenuPais_Click(sender As Object,
  e As RoutedEventArgs)
    MessageBox.Show(DirectCast(sender, MenuItem).Header.ToString())
  End Sub

  Private Sub PaisAñadir_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ' Construir un título para el menú
    Dim titulo As String = "País " & (menuPais.Items.Count - 2)
    ' Crear un elemento con el título construido
    Dim elemento As New MenuItem()
    elemento.Header = titulo
    ' Especificar cuál será su controlador de eventos Click
    AddHandler elemento.Click, AddressOf ElementoMenuPais_Click
    ' Añadir el elemento al menú País
    menuPais.Items.Add(elemento)
  End Sub

  Private Sub ArchivoSalir_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    Close()
  End Sub
End Class
```

En este código se observa que tanto el menú "País" (identificado por *menuPais*) como el controlador *ElementoMenuPais_Click*, para los elementos que serán añadidos dinámicamente, fueron añadidos durante el diseño. Recuerde que la sentencia *AddHandler elemento.Click, AddressOf ElementoMenuPais_Click* permite vincular el evento **Click** de *elemento* con un controlador de eventos.

El resultado de ejecutar el código anterior será el siguiente:



A continuación, vamos a añadir otra orden, "Eliminar", que permita quitar el último elemento añadido. Ahora, el menú "País" tendrá inicialmente dos elementos, "Añadir" y "Eliminar", y un separador.

Añadimos los elementos "Eliminar" (*PaisEliminar*) y un separador (*Separador1*) al menú "País", y asignamos a su propiedad **Visibility** el valor **Collapsed** para que inicialmente, y cuando no haya elementos añadidos, no sean mostrados:

```
<Menu Name="BarraDeMenus">
  <MenuItem Header="Archivo" Name="menuArchivo">
    <MenuItem Header="Salir" Name="ArchivoSalir"
              Click="ArchivoSalir_Click" />
  </MenuItem>
  <MenuItem Header="País" Name="menuPais">
    <MenuItem Header="Añadir" Name="PaisAñadir"
              Click="PaisAñadir_Click" />
    <MenuItem Header="Eliminar" Name="PaisEliminar"
              Visibility="Collapsed" />
    <Separator Name="Separador1" Visibility="Collapsed" />
  </MenuItem>
</Menu>
```

Después, modificamos el controlador *PaisAnadir_Click* como se indica a continuación:

```
Private Sub PaisAñadir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Inicialmente hay 3 elementos (dos ocultos)
```

```
  If menuPais.Items.Count = 3 Then
    ' Hacer visibles los elementos Eliminar y el Separador1
    PaisEliminar.Visibility = Visibility.Visible
    Separador1.Visibility = Visibility.Visible
  End If
  ' Construir un título para el menú
  Dim titulo As String = "País " & (menuPais.Items.Count - 2)
  ' Crear un elemento con el título construido
  Dim elemento As New MenuItem()
  elemento.Header = titulo
  ' Especificar cuál será su controlador de eventos Click
  AddHandler elemento.Click, AddressOf ElementoMenuPais_Click
  ' Añadir el elemento al menú País
  menuPais.Items.Add(elemento)
End Sub
```

Para finalizar, añadimos el controlador del elemento "Eliminar". Cuando no haya elementos "país" los elementos *PaisEliminar* y *Separador1* permanecerán ocultos, y cuando se añada el primer elemento "país" se harán visibles. Según esto, edite el controlador *PaisEliminar_Click* como se indica a continuación:

```
Private Sub PaisEliminar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Eliminar el último elemento
  Dim indUltimo As Integer = menuPais.Items.Count - 1
  menuPais.Items.RemoveAt(indUltimo)
  If menuPais.Items.Count = 3 Then
    ' No quedan más países. Ocultar los elementos
    ' Eliminar y el separador
    PaisEliminar.Visibility = Visibility.Collapsed
    Separador1.Visibility = Visibility.Collapsed
  End If
End Sub
```

El método **RemoveAt** permite eliminar el elemento de la colección que está en la posición especificada (la primera posición es la cero).

RESUMEN

En este capítulo se ha expuesto cómo añadir una barra de menús a una ventana WPF. Cuando creamos una barra de menús, vimos que las acciones llevadas a cabo por los elementos que componen cada menú de la misma podían ser programadas utilizando controladores de eventos, o bien órdenes enrutadas. Por eso, estudiamos también cómo utilizar las órdenes enrutadas predefinidas en WPF y cómo crear órdenes enrutadas personalizadas.

Una vez expuestas las órdenes enrutadas, estudiamos cómo añadir un acelerador de teclado o de ratón a un elemento de un menú, ya que esta programación requiere órdenes enrutadas.

A continuación, expusimos cómo vincular un menú contextual con un elemento WPF. Después, aprendimos a diseñar barras de herramientas y de estado, ya que son objetos muy comunes en el diseño de ventanas WPF.

Finalmente, pusimos en práctica todo lo expuesto. Para ello, planteamos el diseño de un editor de textos básico. Esta aplicación posibilitó aprender otras cosas como trabajar con cajas de texto multilínea, utilizar el portapapeles, manipular la selección del texto, habilitar/inhabilitar los elementos de un menú, marcar el elemento seleccionado de un menú, implementar las operaciones de deshacer y rehacer, asociar un icono a la aplicación e implementar menús dinámicos.

CAPÍTULO 4

© F.J.Ceballos/RA-MA

CONTROLES Y CAJAS DE DIÁLOGO

Un *control* es un componente de la interfaz de usuario que proporciona un comportamiento interactivo determinado. Existen muchos controles en WPF. Algunos, como las etiquetas, las cajas de texto, los botones de pulsación, los menús y los elementos de los menús, ya los hemos estudiado. Otros, como las casillas de verificación, los botones de opción, las listas, los controles de rango definido, el control de pestañas, los controles para la gestión de fechas, las tablas o los árboles, serán estudiados en este capítulo.

Una *caja de diálogo* o *cuadro de diálogo* es una ventana destinada a mantener una conversación corta con el usuario para mostrar u obtener algún dato específico necesario para que la aplicación pueda continuar. Para mostrar u obtener los datos a los que nos referimos, la ventana incluirá los controles apropiados que permitan esas operaciones. De una forma sencilla, una caja de diálogo es un objeto derivado de la clase **Window**. Las cajas de diálogo las podemos clasificar en:

- *Cajas de mensaje*. Son cajas de diálogo creadas por medio del método **Show** de la clase **System.Windows.MessageBox**.
- *Cajas de diálogo personalizadas*. Son cajas de diálogo hechas a medida, para lo cual la biblioteca WPF proporciona la clase **Window**.
- *Cajas de diálogo estándar*. Son cajas de diálogo muy comunes proporcionadas por clases del espacio de nombres **System.Windows.Controls**; por ejemplo, las cajas de diálogo *Abrir* o *Guardar* proporcionadas por las clases **OpenFileDialog** o **SaveFileDialog**, respectivamente, o el diálogo *Imprimir* proporcionado por la clase **PrintDialog**.

A diferencia de una ventana principal (ventana primaria), una caja de diálogo es una ventana secundaria que depende de otra.

CAJAS DE DIÁLOGO MODALES Y NO MODALES

Una caja de diálogo puede crearse *modal* o *no modal*. Cuando una aplicación necesita datos adicionales de los usuarios para continuar visualizará una *caja de diálogo modal*, esto es, una caja de diálogo que tiene que ser cerrada para poder continuar (normalmente pulsando el botón *Aceptar* o el botón *Cancelar*). Si no es así, estaremos en el caso de una *caja de diálogo no modal*, esto es, una caja de diálogo que no impide que el usuario active otras ventanas mientras está abierta.

La clase **MessageBox** crea diálogos modales y la clase **Window** permite crear diálogos modales (utilizando **ShowDialog**) y no modales (utilizando **Show**).

CAJAS DE MENSAJE

La forma más fácil de mostrar un resultado o un mensaje es utilizando las cajas de diálogo que WPF provee para estos propósitos y que se visualizan utilizando el método **Show** de la clase **MessageBox**. Este método, como veremos a continuación, tiene varias sobrecargas.

El método **Show** de la clase **MessageBox** visualiza un mensaje en una caja de diálogo semejante al de la figura mostrada un poco más adelante. Su sintaxis es la siguiente (los argumentos entre corchetes son opcionales):

```
Public Shared Function Show(
   [propietaria As Window,]
   mensaje As String
   [, título As String
   [, botones As MessageBoxButton
   [, icono As MessageBoxImage
   [, botonPredeterminado As MessageBoxResult
   [, opciones As MessageBoxOptions]]]]]
) As MessageBoxResult
```

El segundo argumento es obligatorio; el resto son opcionales. El significado de cada uno de ellos es el siguiente:

- *propietaria* – ventana propietaria de la caja de mensaje.
- *mensaje* – texto a visualizar.
- *título* – título del diálogo.

- *botones* – uno de los valores de la enumeración **MessageBoxButton** que especifica los botones que se mostrarán en el diálogo:

  OK. El diálogo contiene un botón *Aceptar*.
  OKCancel. El diálogo contiene un botón *Aceptar* y otro *Cancelar*.
  YesNo. El diálogo contiene un botón *Sí* y otro *No*.
  YesNoCancel. El diálogo contiene un botón *Sí*, otro *No* y otro *Cancelar*.

- *icono* – uno de los valores de la enumeración **MessageBoxImage** que especifica el icono que se va a mostrar:

  Information | Asterisk. El diálogo contiene un símbolo que consiste en una letra 'i' minúscula dentro de un círculo.
  Error | Hand | Stop. El diálogo contiene un símbolo que consiste en una X blanca dentro de un círculo con fondo rojo.
  Exclamation | Warning. El diálogo contiene un símbolo que consiste en un signo de exclamación dentro de un triángulo con fondo amarillo.
  None. El diálogo no contiene ningún símbolo.
  Question. El diálogo contiene un signo de interrogación dentro de un círculo.

- *botonPredeterminado* – uno de los valores de **MessageBoxResult** que especifica cuál es el botón predeterminado de la caja de mensaje (véanse más adelante los valores devueltos por la caja de mensaje).

- *opciones* – uno de los valores de **MessageBoxOptions** que especifica las opciones de pantalla y asociación que se utilizarán para el diálogo:

  RightAlign. El texto del diálogo está alineado a la derecha.
  RtlReading. El texto del diálogo se muestra con orden de lectura de derecha a izquierda.
  ServiceNotification. El diálogo se muestra en el escritorio activo.
  DefaultDesktopOnly. Igual que **ServiceNotification** excepto en que el sistema muestra el diálogo sólo en el escritorio predeterminado.

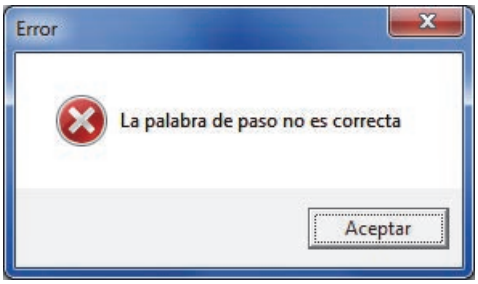
  El método **Show** devuelve uno de los valores de **MessageBoxResult**:

- **Cancel**. Se pulsó el botón *Cancelar*.
- **No**. Se pulsó el botón *No*.
- **None**. Significa que el cuadro de diálogo modal continúa ejecutándose.
- **OK**. Se pulsó el botón *Aceptar*.
- **Yes**. Se pulsó el botón *Sí*.

Por ejemplo, la sentencia

```
MessageBox.Show("La palabra de paso no es correcta",
                "Error",
                MessageBoxButton.OK, MessageBoxImage.Error)
```

visualiza la caja de mensaje siguiente:



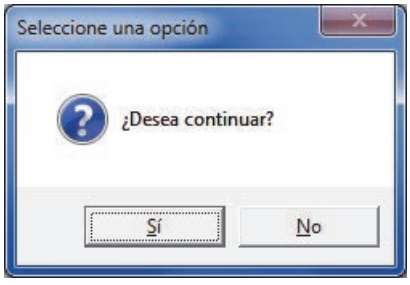
El primer parámetro contiene el mensaje que se desea visualizar. El resto de los parámetros que le siguen son opcionales.

En ocasiones tendremos que confirmar alguna acción. Por ejemplo, el segmento de código siguiente solicita una respuesta afirmativa para continuar.

```
Dim respuesta As MessageBoxResult
respuesta = MessageBox.Show("¿Desea continuar?",
                            "Seleccione una opción",
                            MessageBoxButton.YesNo,
                            MessageBoxImage.Question)

If respuesta = MessageBoxResult.Yes Then
   MessageBox.Show("Se pulsó el botón Sí")
Else
   MessageBox.Show("Se pulsó el botón No")
End If
```

El diálogo que se visualiza es el siguiente:



En función de la respuesta *Sí* o *No* dada, se ejecutará una u otra acción.

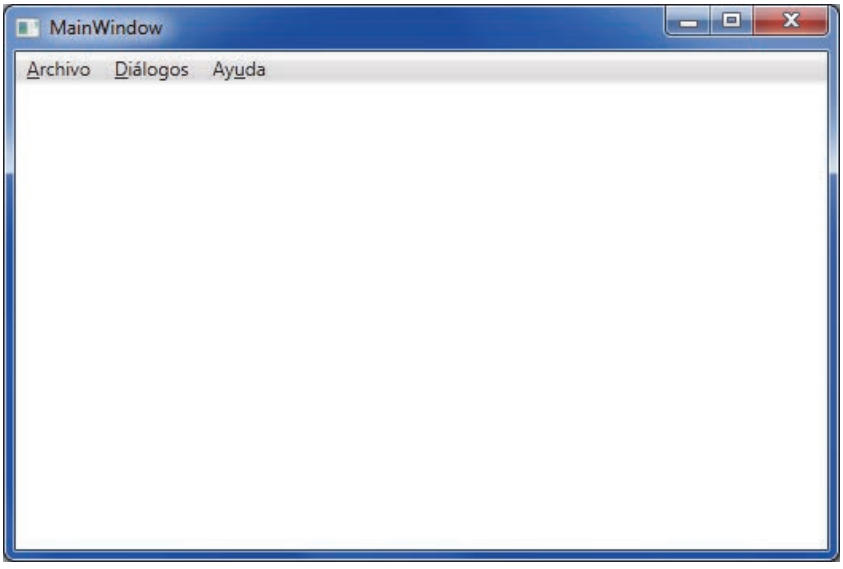
CAJAS DE DIÁLOGO PERSONALIZADAS

Además de los diálogos predefinidos, podemos crear nuestras propias ventanas de diálogo personalizadas mediante el diseñador del EDI. Basta con crear un objeto **Window** y agregar controles como **Label**, **TextBox** o **Button**, entre otros, para adaptar el diálogo a nuestras necesidades.

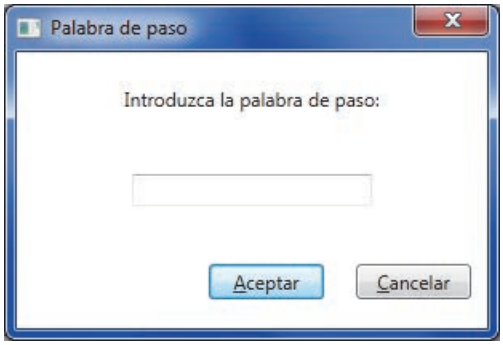
Si observamos las aplicaciones realizadas hasta ahora, la interfaz gráfica que presentaban era un objeto de una clase derivada de **Window** y los controles, objetos miembro de dicha clase. Para crear un diálogo se sigue el mismo criterio; esto es, un diálogo será un objeto de una clase derivada de **Window** y los controles serán objetos miembro de esta clase. Este diálogo será *modal* cuando se utilice el método **ShowDialog** para mostrarlo y *no modal* cuando se utilice el método **Show**. Por ello, es aconsejable que repase los conceptos que explicamos en el capítulo *Aplicación WPF* acerca de las ventanas, su ciclo de vida, sus propiedades, etc.

Como ejemplo, vamos a crear una aplicación denominada *CajasDeDialogo* que utilice un objeto **Window** como ventana principal, con la intención de visualizar, a partir de ella, diferentes diálogos. Para ello, cree el esqueleto de la aplicación y añada al formulario una barra de menús como muestra la figura siguiente.

El menú *Archivo* contendrá sólo una orden, *Salir*, el menú *Diálogos* inicialmente lo dejamos vacío y el menú *Ayuda* contendrá sólo una orden, *Acerca de*.



Supongamos ahora que deseamos que el usuario introduzca una palabra de paso (una clave) en el momento de iniciar la aplicación. Esto lo hará a través de un diálogo como el siguiente:



En el supuesto de que la palabra de paso introducida sea incorrecta, la aplicación lo notificará con un mensaje y le permitirá al usuario dos intentos más para subsanar el error. También, el usuario podrá cancelar la iniciación de la aplicación. Las operaciones descritas habrá que realizarlas escribiendo el código necesario en el proceso de iniciación de la aplicación. Por lo tanto, añada al constructor de la clase de la ventana principal, *MainWindow*, el código que se indica a continuación:

```
Class MainWindow
  Private Enum ResultadoDlg
    Aceptar
    Cancelar
    [Error]
  End Enum

  Public Sub New()
    InitializeComponent()

    ' Se admiten tres intentos para introducir la palabra de paso
    Dim nCuenta As Integer = 0
    Dim vr As ResultadoDlg
    ' Aceptar, Cancelar o Error
    Do
      nCuenta += 1
      vr = contraseña()
    Loop While vr = ResultadoDlg.[Error] AndAlso nCuenta < 3
    If vr = ResultadoDlg.Cancelar OrElse nCuenta = 3 Then
      Close()
    End If
  End Sub
End Class
```

El método *contraseña* de la clase *MainWindow* construirá la caja de diálogo, la mostrará para que el usuario pueda introducir la contraseña y devolverá *Aceptar* si la contraseña fue correcta, *Cancelar* si el usuario canceló la operación de inicio y *Error* si la contraseña no fue correcta. Estos tres valores son definidos por la enumeración *ResultadoDlg*. La aplicación se cerrará si el inicio se cancela o si después de tres intentos, la contraseña fue incorrecta.

Este diálogo para la obtención de la contraseña, podríamos construirlo diseñando una caja de diálogo personalizada con un control **PasswordBox**, entre otros, con su propiedad **PasswordChar** puesta al valor ● (viñeta), que es el valor predeterminado para esta propiedad. De esta forma, la caja de texto mostrará viñetas (●) en lugar de los caracteres tecleados en la misma. La contraseña se almacena en la propiedad **Password**, pero al obtenerla se expone en la memoria como texto sin formato. A fin de evitar este posible riesgo de seguridad, puede usar la propiedad **SecurePassword** para obtener la contraseña como un **SecureString** (un objeto de esta clase mantiene el texto cifrado).

Crear una caja de diálogo

Para crear una caja de diálogo como la que acabamos de describir, durante la fase de diseño, siga los pasos indicados a continuación:

1. Agregue una ventana al proyecto; para ello, diríjase al explorador de soluciones y haga clic con el botón secundario del ratón en el nombre del proyecto, seleccione *Agregar* y, a continuación, haga clic en *Ventana*. Asigne a la ventana el nombre *DlgPalabraDePaso*.

2. Diríjase a la ventana de código XAML y asigne a la caja de diálogo el título "Palabra de paso".

3. Las cajas de diálogo no suelen incluir barra de menús, barras de desplazamiento de ventana, menú de control, botones minimizar (muestra la ventana en la barra de tareas) y maximizar, barra de estado y, en algunos casos, ni borde de tamaño ajustable. Por lo tanto, en la ventana propiedades, cambie su propiedad **ResizeMode** a *NoResize* y **ShowInTaskbar** a **False**.

4. Para que el diálogo se muestre inicialmente en el centro de la ventana propietaria, que en nuestro caso es *MainWindow*, asigne a su propiedad **WindowStartupLocation** el valor *CenterOwner*.

5. Normalmente una caja de diálogo mostrará un botón *Aceptar* para que los usuarios puedan cerrar el diálogo y continuar con el proceso, otro botón *Can-*

celar para que puedan cerrar el diálogo y detener el proceso iniciado y, de forma predeterminada, un botón *Cerrar* en la barra de título del diálogo.

6. Añada a la caja de diálogo los controles necesarios. En nuestro caso añadiremos los controles especificados en la tabla siguiente correspondientes a los mostrados en la figura expuesta anteriormente. Para distribuirlos, utilice un **Grid** de tres filas y, a su vez, en la tercera fila incluya un **StackPanel** horizontal para ubicar los botones en la parte derecha del diálogo.

Objeto	**Propiedad**	**Valor**
Etiqueta	Name Content	etPalabraDePaso Introduzca la palabra de paso:
Caja de contraseña	Name HorizontalAlignment VerticalAlignment Width	ctPalabraDePaso Center Center 150
Botón de pulsación	Name Content Width Margin IsDefault	btAceptar _Aceptar 72 10 True
Botón de pulsación	Name Content Width Margin IsCancel	btCancelar _Cancelar 72 10 True

Respecto a los botones añadidos, es recomendable configurar la acción de pulsar la tecla *Entrar* como un método abreviado de teclado de pulsar el botón *Aceptar*, esto se logra asignando a la propiedad **IsDefault** del botón *Aceptar* el valor **True**, y configurar la acción de pulsar la tecla *Esc* como un método abreviado de teclado de pulsar el botón *Cancelar*, lo cual se logra asignando a la propiedad **IsCancel** del botón *Cancelar* el valor **True**.

Otro detalle es establecer el foco en el elemento de la caja de diálogo que nos gustaría que recibiera inicialmente la entrada del teclado. En nuestro caso este elemento será la caja de texto. Esto puede hacerse como explicamos en el apartado *Eventos relacionados con el foco* del capítulo *Introducción a WPF*, o vinculando la propiedad asociada **FocusManager.FocusedElement** con ese elemento en el código XML que define la ventana:

```
<Window x:Class="DlgPalabraDePaso"
  ...
  Title="Palabra de paso" Height="200" Width="300"
```

```
  ShowInTaskbar="False"
  ResizeMode="NoResize"
  WindowStartupLocation="CenterOwner"
  FocusManager.FocusedElement="{
                      Binding ElementName=ctPalabraDePaso}">
```

Como resultado del trabajo realizado hasta ahora, disponemos de una clase *DlgPalabraDePaso* derivada de **Window** que permitirá crear objetos (cajas de diálogo) con los atributos y métodos que hayamos establecido (propiedades, métodos, variables, controles, controladores de eventos, etc.).

```
Public Class DlgPalabraDePaso

End Class
```

Así, un objeto *Dlg* de la clase *DlgPalabraDePaso*,

```
Dim Dlg As New DlgPalabraDePaso()
```

será una caja de diálogo como la mostrada en la figura anterior.

Mostrar una caja de diálogo

La ventana principal (en nuestro caso *MainWindow*) se cargará automáticamente cuando se ejecute la aplicación. Para mostrar durante la ejecución de la aplicación una segunda ventana secundaria o caja de diálogo, desde esta ventana principal habrá que crear un objeto de su clase (**New**) y mostrarlo (**ShowDialog** o **Show**). Por ejemplo:

```
Private Function contraseña() As ResultadoDlg
  Dim Dlg As New DlgPalabraDePaso()
  ' Mostrar el diálogo
  Dlg.ShowDialog()
  ' ...
End Function
```

Si el objeto **Window** se muestra invocando al método **ShowDialog**, estaremos en el caso de una ventana modal y será no modal si se muestra invocando a **Show**.

Si ahora ejecuta la aplicación, observará que se muestra la caja de diálogo *Palabra de paso*. También podrá observar que si pulsa el botón *Aceptar* no ocurre nada y que si pulsa el botón *Cancelar* se cierra el diálogo; esto último es así porque se asignó a la propiedad **IsCancel** el valor **True**. Más detalles, a continuación.

Gestionar los botones Aceptar y Cancelar

Ahora que sabemos cómo mostrar la caja de diálogo, necesitamos saber también cómo programar si serán o no procesados los datos a los que hace referencia el diálogo. Esto puede programarse en función del valor que devuelva el método **ShowDialog**. El prototipo de este método es así:

```
Public Function ShowDialog() As Nullable(Of Boolean)
```

El tipo **Nullable(Of Boolean)**, que equivale al tipo **Boolean?**, representa un objeto cuyo tipo subyacente es un tipo valor (**True** o **False**) al que también se le puede asignar **Nothing** como tipo referencia. El valor **True** especifica que el proceso iniciado continúa y **False** que se cancela. El valor retornado corresponde al valor de la propiedad **DialogResult** de la caja de diálogo antes de que se cierre. Cuando se abre el diálogo, el valor para esta propiedad es **Nothing**, y es **False** cuando, estando la propiedad **IsCancel** a **True**, se invoca al método **Close** o cuando el usuario pulsa las teclas *Alt+F4*, hace clic en el botón *Cerrar* o selecciona *Cerrar* en el menú de sistema (menú desplegado al hacer clic en el icono de la barra de título).

Según lo estudiado hasta ahora, una caja de diálogo con un botón *Cancelar* cuya propiedad **IsCancel** está establecida en **True** cerrará automáticamente el diálogo cuando se pulse ese botón o cuando se presione la tecla *Esc*. En cualquiera de estos casos, el valor de **DialogResult** será establecido a **False**. Y una caja de diálogo con un botón *Aceptar* cuya propiedad **IsDefault** está establecida en **True** no cerrará automáticamente el diálogo cuando se pulse ese botón o se presione la tecla *Entrar*, ni establecerá la propiedad **DialogResult** a **True**; para que esto ocurra, es preciso escribir manualmente el código que lo haga, normalmente en el controlador del evento **Click** del botón *Aceptar* de la caja de diálogo:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  DialogResult = True
End Sub
```

Si ahora ejecuta la aplicación, observará que se muestra la caja de diálogo *Palabra de paso*. También podrá observar que si pulsa el botón *Aceptar* o el botón *Cancelar*, se cierra el diálogo. Obsérvese que para que se cierre el diálogo no es necesario invocar explícitamente al método **Close**, de esto se encarga, supuestamente, directa o indirectamente, **ShowDialog**.

Según lo expuesto, el método *contraseña* se perfila así:

```
Private Function contraseña() As ResultadoDlg
  Dim Dlg As New DlgPalabraDePaso()
```

```
  ' Mostrar el diálogo
  If Dlg.ShowDialog() = True Then
    ' Recuperar los datos del diálogo y continuar
  Else
    ' Detener el proceso iniciado
    Return ResultadoDlg.Cancelar
  End If

  ' Verificar si la palabra de paso es correcta
End Function
```

Introducción de datos y recuperación de los mismos

Cuando la ventana de inicio necesite pedir datos al usuario, en muchos casos lo hará mostrando una caja de diálogo en la que el usuario los introducirá, y cuando éste cierre el diálogo, la ventana de inicio recuperará esos datos desde los atributos del diálogo.

Para recuperar los datos de una caja de diálogo es importante saber cómo se cerró: valor **DialogResult** devuelto por **ShowDialog**. Por ejemplo, si el usuario hace clic en el botón *Aceptar* los datos se recuperarán, pero si lo hace en *Cancelar*, los datos escritos por el usuario se desecharán en lugar de conservarse.

```
If Dlg.ShowDialog() = True Then
  ' Recuperar los datos del diálogo y continuar
Else
  ' Detener el proceso iniciado
End If
```

Obsérvese que *Dlg.ShowDialog()* devuelve el valor de la propiedad **DialogResult** del diálogo y que ha sido establecida explícitamente en el caso del botón *Aceptar* e implícitamente en el caso del botón *Cancelar* (en este último caso, tendría que ser establecida explícitamente si no se hubiera asignado a la propiedad **IsCancel** el valor **True**). Esto quiere decir que, incluso cuando no haya botones, tiene que establecer el valor de la propiedad **DialogResult** mediante código para posteriores comprobaciones. Otro ejemplo:

```
Public Class DlgPalabraDePaso
  ' ...
  Private Sub textBox1_PreviewTextInput(sender As Object,
  e As TextCompositionEventArgs)
    Dim cr As String = Convert.ToChar(13).ToString()
    Dim esc As String = Convert.ToChar(27).ToString()
    If e.Text = cr Then
      DialogResult = True
    End If
```

```
    If e.Text = esc Then
      DialogResult = False
    End If
  End Sub
End Class
```

Cuando el diálogo modal o no modal se cierra el objeto **Window** no se destruye mientras esté referenciado por una variable. Esto permite acceder a sus propiedades para obtener los valores de las mismas, allí donde la variable que lo identifica sea visible. Por ejemplo:

```
If Dlg.ShowDialog() = True Then
  sPalabraDePaso = Dlg.ctPalabraDePaso.Password
Else
  ' ...
End If
```

Según lo expuesto y continuando con nuestro ejemplo, edite el método *contraseña* como se indica a continuación:

```
Private Function contraseña() As ResultadoDlg
  Dim sPalabraDePaso As String
  ' Crear la caja de diálogo de entrada
  Dim Dlg As New DlgPalabraDePaso()

  ' Mostrar el diálogo
  If Dlg.ShowDialog() = True Then
    ' Obtener los datos
    sPalabraDePaso = Dlg.ctPalabraDePaso.Password
  Else
    Return ResultadoDlg.Cancelar
  End If

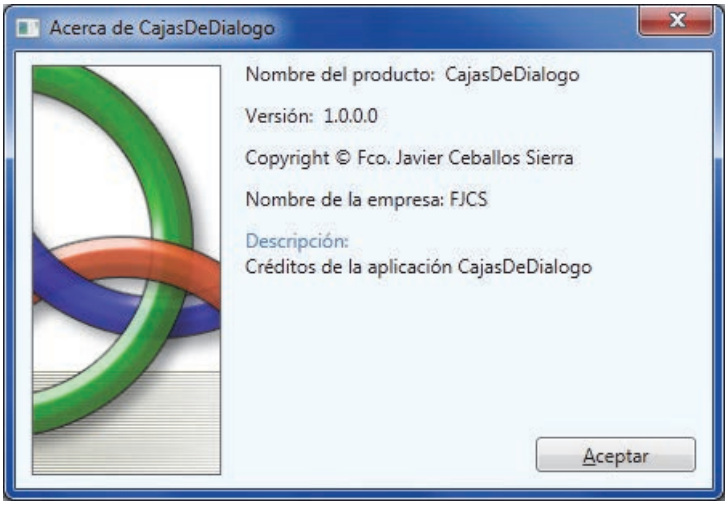
  ' Verificar si la palabra de paso es correcta
  Dim pp As String = MySettings.Default.PalabraDePaso
  If sPalabraDePaso.CompareTo(pp) <> 0 Then
    MessageBox.Show("La palabra de paso no es correcta", "Error",
                    MessageBoxButton.OK, MessageBoxImage.[Error])
    Return ResultadoDlg.[Error]
  End If
  Return ResultadoDlg.Aceptar
End Function
```

El código anterior requiere un parámetro de configuración que hemos denominado *PalabraDePaso* de tipo **String**. Para añadirlo ejecute *Proyecto* > *Propiedades* > *Configuración*. Este parámetro quedará definido como una propiedad de la clase *MySettings* (para más detalles vea el apartado *Configuración de una aplicación* en el capítulo *Aplicación WPF*).

Puede ver el código completo de esta aplicación en la carpeta *Cap04\CajasDeDialogo* del CD que acompaña al libro.

DIÁLOGO ACERCA DE

El diálogo *Acerca de* tiene como finalidad mostrar los créditos de la aplicación. Siguiendo los pasos expuestos en los apartados anteriores, puede diseñar una caja de diálogo personalizada análoga a la de la figura siguiente. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y ejecute *Agregar > Ventana*. Después, realice el diseño que considere oportuno.



El diseño anterior ha sido realizado a partir del siguiente código:

```
<Window x:Class="AcercaDe"
  ...
  Height="300" Width="440" ResizeMode="NoResize"
  WindowStartupLocation="CenterOwner" ShowInTaskbar="False"
  Topmost="False" Background="AliceBlue">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Image Grid.Column="0" Grid.RowSpan="2"
      Source="Imagenes/logo.png" Margin="8" />
    <StackPanel Grid.Column="1" Grid.Row="0">
```

```
      <Label Name="etNombreProd" Content="Nombre del producto: " />
      <Label Name="etVersion" Content="Versión: " />
      <Label Name="etCopyright" Content="Copyright: " />
      <Label Name="etNombreEmpr" Content="Nombre de la empresa: "/>
      <TextBlock Name="bktDescrip" MinWidth="0" MinHeight="100"
        MaxHeight="100" Padding="5" TextWrapping="Wrap"
        Foreground="SteelBlue" Text="Descripción:" >
        <LineBreak/>
        <Run Name="bktDescrip_sec1" Foreground="Black" Text="Texto" />
      </TextBlock>
    </StackPanel>
    <Button Grid.Column="1" Grid.Row="1" Name="btAceptar"
      Content="_Aceptar" HorizontalAlignment="Right"
      Margin="5,5,10,10" Padding="1" Width="100"
      Click="btAceptar_Click" />
  </Grid>
</Window>
```

En este diseño observamos un **Grid** con dos filas y dos columnas. La primera columna, en sus dos filas, es ocupada por una imagen. La segunda columna en su primera fila es ocupada por un **StackPanel** que incluye cuatro etiquetas y un bloque de texto. El bloque de texto, a su vez, incluye dos secciones: la definida por su propiedad **Text** y la definida por **Run** separada de la anterior por **LineBreak**. Los objetos **LineBreak** y **Run** también se pueden utilizar como elementos secundarios de **TextBlock** para representar texto con formato. **Run** es un elemento de texto con o sin formato y **LineBreak** representa explícitamente una nueva línea. La segunda columna en su segunda fila incluye el botón *Aceptar*.

Y, ¿de dónde se proveen los controles de la información que visualizan? Pues de los atributos globales de la aplicación definidos en el fichero *AssemblyInfo.vb* del proyecto (véase el apartado *Atributos globales de una aplicación* del capítulo *Aplicación WPF*). Por ejemplo:

```
Imports System.Reflection

Public Class AcercaDe
  Public Sub New()
    InitializeComponent()

    ' Atributos globales
    Me.Title = DirectCast(ObtenerInfo( _
                GetType(AssemblyTitleAttribute)),
                AssemblyTitleAttribute).Title
    Me.etNombreProd.Content = "Nombre del producto:  " + _
         DirectCast(ObtenerInfo( _
                GetType(AssemblyProductAttribute)),
                AssemblyProductAttribute).Product
    ' ...
```

```
  End Sub

  Private Function ObtenerInfo(atributo As Type,
  Optional h As Boolean = False) As Object
    ' Atributos globales
    Dim ensamblado As Assembly = Assembly.GetExecutingAssembly()
    Dim atributos As Object() =
          ensamblado.GetCustomAttributes(atributo, h)
    Return atributos(0)
  End Function

  Private Sub btAceptar_Click(sender As Object,
  e As RoutedEventArgs)
    Close()
  End Sub
End Class
```

Finalmente, la función del botón *Aceptar* de este diálogo es cerrar el mismo.

Para mostrar esta ventana añada a *MainWindow* el controlador para el evento **Click** de la orden *Acerca de* del menú *Ayuda* y complételo como se indica a continuación:

```
Private Sub AyudaAcercaDe_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dlg As New AcercaDe()
  dlg.ShowDialog()
End Sub
```

VENTANA PROPIETARIA

Una ventana que se abre llamando al método **ShowDialog** o **Show** no tiene automáticamente una relación con la ventana que la abrió: ventana propietaria; esto es, la ventana abierta no sabe qué ventana la abrió. Esta relación puede establecerse mediante la propiedad **Owner** de la ventana abierta y puede ser administrada mediante la propiedad **OwnedWindows** de la ventana propietaria. Por ejemplo:

```
Private Sub MostrarDlg_Click(sender As Object,
e As RoutedEventArgs)
  Dim dlg As New MiDlg()
  dlg.Owner = Me
  If dlg.ShowDialog() = True Then
    ' ...
  End If
End Sub
```

O bien,

```
Private Sub MostrarDlg_Click(sender As Object,
e As RoutedEventArgs)
  Dim dlg As New MiDlg()
  dlg.Owner = Me
  dlg.Show()
  ' ...
End Sub
```

A diferencia de una ventana abierta con **ShowDialog**, una ventana que se abre llamando al método **Show** no impide que los usuarios interactúen con otras ventanas de la aplicación. El método **Show** no devuelve nada, por lo que la propiedad **DialogResult** no tiene sentido en este tipo de ventanas. Para ambos tipos de ventanas, modales y no modales, el método **Close** simplemente cierra la ventana, no destruye el objeto **Window**, por lo que es posible acceder a su funcionalidad pública desde cualquier parte en la que se disponga de una referencia al mismo.

La propiedad **Owner** de una ventana tiene como función guardar una referencia a la ventana propietaria de esa ventana (el valor predeterminado es **Nothing**). De esta forma, utilizando el valor de **Owner**, la ventana que es propiedad podrá acceder a la funcionalidad pública de su ventana propietaria.

La propiedad **OwnedWindows** obtiene una colección de ventanas de las que una determinada ventana es propietaria.

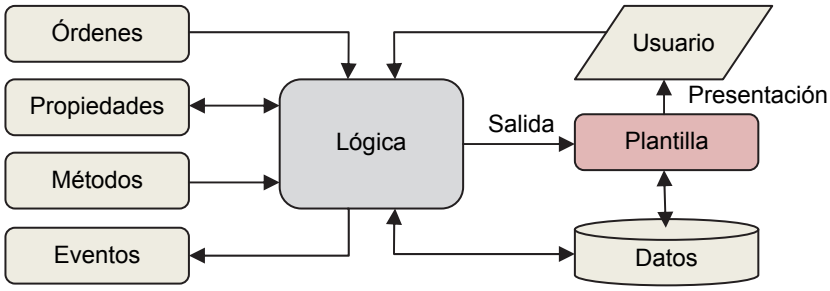
OTROS CONTROLES WPF

Según lo estudiado hasta ahora, las ventanas WPF se llenan con elementos para formar la interfaz gráfica del usuario, pero sólo algunos de estos elementos son controles, objetos derivados de la clase **Control**, entendiendo por control un elemento que puede recibir el foco y aceptar la entrada desde el teclado o desde el ratón. Ejemplos evidentes son las cajas de texto y los botones. Un **ToolTip** es un control porque su contenido aparece y desaparece en función del ratón y una etiqueta es un control debido a que soporta el acceso directo para transferir el foco a controles relacionados como la caja de texto. Sin embargo, un **TextBlock** (objeto derivado de **FrameworkElement**) no se considera un control, ya que no puede recibir el foco, pero, al igual que los controles, tiene una presencia visible sin necesidad de heredar de la clase **Control**.

La clase **System.Windows.Controls.Control** añade la infraestructura básica para un control; esto es, la capacidad para establecer la alineación del contenido dentro del control, la capacidad para establecer el orden de tabulación, soporte para establecer el color de fondo y del primer plano, el borde, y para dar formato al contenido de texto. Por lo tanto, es factible modificar el aspecto de un control cambiando el valor de sus propiedades, pero también podemos hacerlo creando un

estilo, **Style**, para el control o creando una nueva plantilla, objeto de la clase **ControlTemplate**, ya que todas las clases que heredan de la clase **Control** contienen una plantilla de este tipo que permite cambiar radicalmente el aspecto de éste sin tener que crear una nueva subclase.

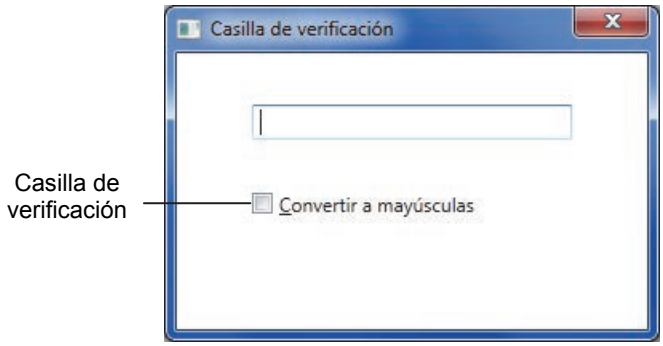
La figura siguiente muestra la relación de un control con sus elementos visuales. El control expone una interfaz programática formada por órdenes (por ejemplo, las cajas de texto soportan las órdenes cortar, copiar y pegar), propiedades, métodos y eventos. El control puede vincularse a determinados datos. Los elementos visuales del control se proporcionan por su plantilla (véase el apartado *Plantillas* en el capítulo *Introducción a WPF*), en lugar de por el propio control, y son el medio utilizado por el control para presentar información al usuario, y por el usuario para proporcionar información al control.



Todo esto ya lo hemos experimentado en mayor o menor medida en los capítulos anteriores, por lo que en este apartado, vamos a exponer otros controles WPF como son las casillas de verificación, los botones de opción, las listas simples y desplegables, los controles de rango definido, el control de pestañas, los controles para la gestión de fechas, vistas como tablas y vistas jerarquizadas o árboles.

Casillas de verificación

Una *casilla de verificación* es un control que indica si una opción particular está activada o desactivada. En la caja de diálogo de la figura mostrada a continuación, se le da al usuario la opción de convertir el texto de la caja a mayúsculas. Cada casilla de verificación es independiente de las demás, ya que cada una de ellas tiene su propio identificador. El número de opciones representadas de esta forma puede ser cualquiera y, de ellas, el usuario *puede seleccionar todas las que desee* cada vez. La funcionalidad para manipular este tipo de controles es proporcionada por la clase **CheckBox** del espacio de nombres **System.Windows.Controls**.



Si durante la ejecución se hace clic en una casilla de verificación, la opción queda seleccionada (☑). Una opción ya seleccionada puede pasar a no estarlo haciendo clic de nuevo sobre la casilla de verificación correspondiente (☐).

Un clic sobre un objeto **CheckBox** para seleccionarlo (activarlo) genera un evento **Checked** seguido de un evento **Click** y un clic para deseleccionarlo (desactivarlo) genera un evento **Unchecked** seguido de un evento **Click**. Para saber el tipo de cambio experimentado por un objeto **CheckBox**, hay que verificar el valor de su propiedad **IsChecked**. Este valor puede ser: **False**, la casilla de verificación no está seleccionada, o **True**, la casilla de verificación está seleccionada.

En realidad, la propiedad **IsChecked** devuelve un valor de tipo **Nullable(Of Boolean)**, que equivale al tipo **Boolean?**, que representa un objeto cuyo tipo subyacente es un tipo valor (**True** o **False**) al que también se le puede asignar **Nothing** como tipo referencia.

La propiedad **IsThreeState**, de tipo **Boolean**, determina si el control soporta dos o tres estados. De manera predeterminada su valor es **False**, lo que indica que soporta dos estados. Cuando su valor es **True**, el control soporta tres estados identificados por la propiedad **IsChecked** que valdría: **True** (casilla seleccionada), **False** (casilla no seleccionada) o **Nothing** (casilla presentando una apariencia sombreada: estado indeterminado). En este supuesto, clics sucesivos sobre la casilla de verificación harían que ésta cambiase al estado seleccionado, indeterminado, no seleccionado y así sucesivamente. Un clic sobre la casilla que la haga cambiar al estado indeterminado genera un evento **Indeterminate** seguido de un evento **Click**.

Por ejemplo, el siguiente método verifica si el control *checkBox1* ha sido seleccionado:

```
Private Sub button1_Click(sender As Object, e As RoutedEventArgs)
  If checkBox1.IsChecked = True Then
    ' La casilla de verificación está marcada
```

```
  End If
End Sub
```

Para aplicar la teoría expuesta, vamos a diseñar una caja de diálogo como la de la figura anterior, de tal forma que cuando el usuario seleccione la opción "Convertir a mayúsculas", todo el texto que haya escrito en la caja de texto aparezca en mayúsculas. El proceso para añadir una nueva caja de diálogo a una aplicación ya fue expuesto anteriormente en este mismo capítulo. Según esto, cargue la aplicación *CajasDeDialogo* que hemos implementado anteriormente y añada una nueva caja de diálogo como la de la figura anterior. La caja de diálogo será un objeto de la clase *DlgCasillaVerificacion* derivada de **Window**. La tabla de controles y sus propiedades puede ser la siguiente:

Objeto	**Propiedad**	**Valor**
Caja de diálogo	Name Title ResizeMode ShowInTaskbar WindowStartupLocation	DlgCasillaVerificacion Casilla de verificación NoResize False CenterOwner
Caja de texto	Name VerticalAlignment Width	ctTexto Center 200
Casilla de verificación	Name Content VerticalAlignment Width	cvConverMayus _Convertir a mayúsculas Top 200

Para que la caja de texto aparezca inicialmente enfocada, en la ventana puede vincular la propiedad **FocusManager.FocusedElement** con ese elemento.

Una vez finalizado el diseño, podremos observar que se ha añadido a la aplicación una nueva clase *DlgCasillaVerificacion* a partir de la cual podremos crear ese tipo de cajas de diálogo. A continuación puede ver el código XAML que necesita escribir para añadir a un diálogo una casilla de verificación:

```
<Window x:Class="DlgCasillaVerificacion"
...
   Title="Casilla de verificación" Height="200" Width="300"
   ResizeMode="NoResize" ShowInTaskbar="False"
   WindowStartupLocation="CenterOwner"
   FocusManager.FocusedElement="{Binding ElementName=ctTexto}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
```

```
    </Grid.RowDefinitions>
    <TextBox Grid.Row="0" Name="ctTexto"
             VerticalAlignment="Center" Width="200" />
    <CheckBox Grid.Row="1" Name="cvConverMayus"
              VerticalAlignment="Top" Width="200"
              Content="_Convertir a mayúsculas" />
  </Grid>
</Window>
```

Para mostrar un diálogo modal de la clase *DlgCasillaVerificacion*, añadiremos al menú *Diálogos* de la aplicación *CajasDeDialogo* un nuevo elemento *DialogoCasillaV* y después, escribiremos el controlador para su evento **Click** así:

```
Private Sub DialogoCasillaV_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
    Dim dlg As New DlgCasillaVerificacion()
    dlg.ShowDialog()
End Sub
```

Este método crea un nuevo objeto *DlgCasillaVerificacion* y lo muestra como una caja de diálogo modal.

A continuación, escribimos el código que permita que la caja de diálogo se comporte de la forma descrita. Para ello, seleccione el control *cvConverMayus* y añada a la clase *DlgCasillaVerificacion* un controlador de eventos **Click** que responda a la acción de cambiar el estado de la casilla de verificación. Dicha respuesta será la ejecución del método siguiente:

```
Private Sub cvConverMayus_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim texto As String = ctTexto.Text
  If cvConverMayus.IsChecked = True Then
    ctTexto.Text = texto.ToUpper()
  Else
    ctTexto.Text = texto.ToLower()
  End If
  ctTexto.Focus()
  ctTexto.SelectionStart = ctTexto.Text.Length
End Sub
```

Este método obtiene el texto de la caja de texto y lo escribe de nuevo en la misma en mayúsculas si se seleccionó la casilla de verificación, o en minúsculas si la casilla de verificación no está seleccionada. Finalmente, sitúa el foco, que estaba sobre la casilla de verificación, en la caja de texto y coloca el punto de inserción a continuación del último carácter.

También, podríamos haber utilizado los eventos **Checked** y **Unchecked** así:

```
Private Sub cvConverMayus_Checked(sender As Object,
e As RoutedEventArgs)
  ctTexto.Text = ctTexto.Text.ToUpper()
End Sub

Private Sub cvConverMayus_Unchecked(sender As Object,
e As RoutedEventArgs)
  ctTexto.Text = ctTexto.Text.ToLower()
End Sub

Private Sub cvConverMayus_Click(sender As Object,
e As RoutedEventArgs)
  ctTexto.Focus()
  ctTexto.SelectionStart = ctTexto.Text.Length
End Sub
```

Otra solución puede ser la siguiente: mientras la casilla no esté seleccionada todo el texto que se escriba deberá visualizarse en minúsculas (independientemente de que estén o no activadas las mayúsculas), mientras la casilla esté seleccionada todo el texto que se escriba deberá visualizarse en mayúsculas y cuando esté en un estado indeterminado el texto que se escriba aparecerá tal cual. Para realizar esto, en primer lugar, asigne a la propiedad **IsThreeState** de la casilla el valor **True** y, en segundo lugar, intercepte la tecla pulsada (evento **PreviewTextInput**) y en función del estado de la casilla de verificación visualice el carácter como corresponda. La solución a este planteamiento puede ser la siguiente:

```
Private Sub ctTexto_PreviewTextInput(sender As System.Object,
e As System.Windows.Input.TextCompositionEventArgs)
  Dim car As Char = e.Text(0)
  If cvConverMayus.IsChecked = False Then
    If Char.IsUpper(car) Then
      car = Char.ToLower(car)
    End If
  ElseIf cvConverMayus.IsChecked = True Then
    If Char.IsLower(car) Then
      car = Char.ToUpper(car)
    End If
  End If
  ctTexto.SelectedText = car.ToString()
  ctTexto.SelectionStart = ctTexto.Text.Length
  e.Handled = True
End Sub

Private Sub cvConverMayus_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ctTexto.Focus()
End Sub
```

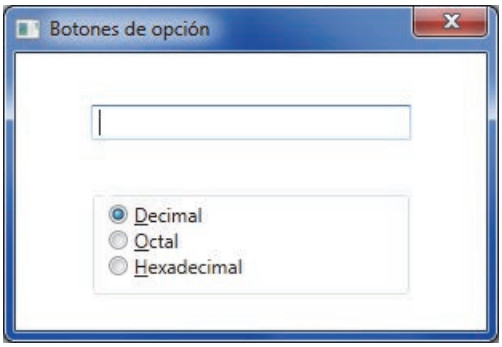
El método *ctTexto_PreviewTextInput* visualiza el carácter tecleado en minúsculas siempre que la casilla no esté marcada (si el usuario tecleó una letra mayúscula, se convierte a minúscula), si está marcada, entonces se visualiza en mayúsculas (si el usuario tecleó una letra minúscula, se convierte a mayúscula), y en otro caso (estado indeterminado) se visualiza tal cual. Observe las tres últimas líneas de este método; la primera de estas líneas añade de una forma rápida texto a la caja sin necesidad de reescribir el contenido de la misma, la segunda sitúa el punto de inserción al final quitando la selección y la tercera da por manipulado el evento.

El método *cvConverMayus_Click* sitúa el foco, que estaba sobre la casilla de verificación, en la caja de texto.

Botones de opción

Un *botón de opción* es un control que indica si una determinada opción está activada o desactivada. Cada botón de opción es independiente de los demás, ya que cada uno de ellos tiene su propio identificador. El número de opciones representadas de esta forma puede ser cualquiera y, de ellas, el usuario *sólo puede seleccionar una* cada vez. La funcionalidad para manipular este tipo de controles es proporcionada por la clase **RadioButton** del espacio de nombres **System.Windows.Controls**.

Todos los controles **RadioButton** de un contenedor determinado, como **Window**, constituyen un grupo. Cuando el usuario selecciona un botón de opción en un grupo, los demás se desactivan automáticamente. Para crear varios grupos en un mismo formulario, coloque cada grupo en su propio contenedor, por ejemplo, en un control **GroupBox** o en un panel, o bien utilice la propiedad **GroupName** del control **RadioButton**.



Si durante la ejecución se hace clic sobre un botón de opción perteneciente a un grupo, la opción queda seleccionada (⦿) y, a diferencia de las casillas de veri-

ficación, ese botón de opción ya seleccionado puede pasar a no estarlo haciendo clic en otro botón de opción del mismo grupo.

Un clic sobre un objeto **RadioButton** para seleccionarlo (activarlo) genera un evento **Checked** seguido de un evento **Click** y un clic para deseleccionarlo (desactivarlo) genera un evento **Unchecked** seguido de un evento **Click**. Para saber el tipo de cambio experimentado por un objeto **RadioButton**, hay que verificar el valor de su propiedad **IsChecked**. Este valor puede ser: **False**, el botón de opción no está seleccionado, o **True**, el botón de opción está seleccionado.

Por ejemplo, el siguiente método verifica qué botón de opción está seleccionado de entre varios:

```
Private Sub button1_Click(sender As Object, e As RoutedEventArgs)
  If RadioButton1.IsChecked = True Then
    ' RadioButton1 está seleccionado
  ElseIf RadioButton2.IsChecked = True Then
    ' RadioButton2 está seleccionado
  End If
End Sub
```

Tenga presente que cuando se hace clic sobre un botón de opción que pertenece a un grupo, hay dos que cambian de estado: el que estaba seleccionado y el nuevo seleccionado.

Como ejemplo, cargue la aplicación denominada *CajasDeDialogo* y añada una nueva caja de diálogo como la de la figura anterior, de tal forma que cuando seleccione uno de los botones de opción "Decimal", "Octal" o "Hexadecimal", el número tecleado en la caja de texto aparezca en la base indicada. Dicho diálogo será un objeto de la clase *DlgBasesNum* derivada de **Window**. La tabla de controles y sus propiedades puede ser la siguiente:

Objeto	**Propiedad**	**Valor**
Caja de diálogo	Name Title ResizeMode ShowInTaskbar WindowStartupLocation	DlgBasesNum Botones de opción NoResize False CenterOwner
TextBox	Name VerticalAlignment Width	ctDato Center 200
GroupBox	VerticalAlignment Width	Top 200

Border	Margin Background Padding	2 White 3
StackPanel		
Botón de opción	Name Content IsChecked	btopDecimal _Decimal True
Botón de opción	Name Content	btopOctal _Octal
Botón de opción	Name Content	btopHex _Hexadecimal

Los botones formarán parte del contenido del **StackPanel**, éste será el contenido del **Border** y éste, el contenido del **GroupBox**.

Para que la caja de texto aparezca inicialmente enfocada, en la ventana puede vincular la propiedad **FocusManager.FocusedElement** con ese elemento.

Una vez finalizado el diseño, podremos observar que se ha añadido a la aplicación una nueva clase *DlgBasesNum* a partir de la cual podremos crear ese tipo de cajas de diálogo. A continuación puede ver el código XAML necesario para añadir a un diálogo un botón de opción en un grupo de botones:

```
<Window x:Class="DlgBasesNum"
  ...
  Title="Botones de opción" Height="200" Width="300"
  ResizeMode="NoResize" ShowInTaskbar="False"
  WindowStartupLocation="CenterOwner"
  FocusManager.FocusedElement="{Binding ElementName=ctDato}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <TextBox Grid.Row="0" Name="ctDato"
             VerticalAlignment="Center" Width="200" />
    <GroupBox Grid.Row="1" VerticalAlignment="Top" Width="200" >
      <Border Margin="2" Background="White" Padding="3">
        <StackPanel >
          <RadioButton Name="btopDecimal" Content="_Decimal" />
          <RadioButton Name="btopOctal" Content="_Octal" />
          <RadioButton Name="btopHex" Content="_Hexadecimal" />
        </StackPanel>
      </Border>
    </GroupBox>
  </Grid>
</Window>
```

Continuando con la aplicación, añada al menú *Diálogos* de la aplicación *CajasDeDialogo* un nuevo elemento denominado *DialogoBtOpcion* con la intención de que cuando el usuario haga clic en esa orden, se visualice una caja de diálogo de la clase *DlgBasesNum*. Por lo tanto, añada a la clase *MainWindow* el controlador del evento **Click** para este nuevo elemento y edítelo como se indica a continuación:

```
Private Sub DialogoBtOpcion_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dlg As New DlgBasesNum()
  dlg.ShowDialog()
End Sub
```

Con esto hemos finalizado la fase de diseño. Compile y ejecute la aplicación, y observe los resultados. El siguiente paso es programar las operaciones para las que ha sido diseñado el diálogo:

1. Cada vez que el usuario escriba o modifique el contenido de la caja de texto, ésta se lo notifica a la caja de diálogo con el evento **TextChanged** (entre otros); para manipularlo, añada el controlador correspondiente. Para ello, diríjase al diálogo *DlgBasesNum* y haga doble clic en la caja de texto:

   ```
   Private Sub ctDato_TextChanged(sender As System.Object,
   e As System.Windows.Controls.TextChangedEventArgs)
     ' ...
   End Sub
   ```

2. Lo que va a hacer el método *ctDato_TextChanged* es almacenar el valor decimal, octal o hexadecimal de la caja de texto (dependiendo esto de la opción seleccionada), en una variable miembro denominada *numeroActual* de tipo **Int32**, de la clase *DlgBasesNum* (el tipo **Int32** define valores enteros de 32 bits y es independiente del sistema). Añada, por lo tanto, esta variable a la clase citada (su valor inicial predeterminado es 0).

   ```
   Public Class DlgBasesNum
     Private numeroActual As Integer
     ' ...
   End Class
   ```

3. Después, edite el método privado *ctDato_TextChanged* como se indica a continuación:

   ```
   Private Sub ctDato_TextChanged(sender As System.Object,
   e As System.Windows.Controls.TextChangedEventArgs)
     If ctDato.Text.Length = 0 Then Return
   ```

```
    Dim texto As String = ctDato.Text
    Dim baseNum As Integer = 0

    If btopDecimal.IsChecked = True Then
      baseNum = 10
    ElseIf btopOctal.IsChecked = True Then
      baseNum = 8
    ElseIf btopHex.IsChecked = True Then
      baseNum = 16
    End If

    Try
      numeroActual = Convert.ToInt32(texto, baseNum)
    Catch ex As FormatException
      If texto = "-" OrElse texto = "+" Then Return
      MessageBox.Show(ex.Message)
    Catch ex As Exception
      MessageBox.Show(ex.Message)
    End Try
  End Sub
```

El método **ToInt32** de la clase **Convert** convierte una cadena de caracteres a un número entero de tipo **Int32**. La cadena a convertir se supone que está expresada en la base especificada por *baseNum*.

4. Cuando el usuario seleccione una de las opciones *Decimal*, *Octal* o *Hexadecimal*, se producirá, entre otros, el evento **Checked** asociado con ese control; para manipularlo, añada el controlador correspondiente, que, en este caso, puede ser el mismo para los tres botones. Para ello, diríjase al diálogo *DlgBasesNum*, seleccione uno a uno los botones de opción, elija ese evento en la ventana de propiedades y vincule con cada uno de los botones el método especificado a continuación:

```
Private Sub btopDecOcHex_Checked(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' ...
End Sub
```

5. El método *btopDecOcHex_Checked* tiene que mostrar en la caja de texto el valor *numeroActual* en decimal, octal o hexadecimal, respectivamente. También, el botón de opción sobre el que se ha hecho clic habrá quedado enfocado. Para situar el foco de nuevo en la caja de texto, invocaremos a continuación a su método **Focus**. Según lo expuesto, edite este método como se muestra a continuación:

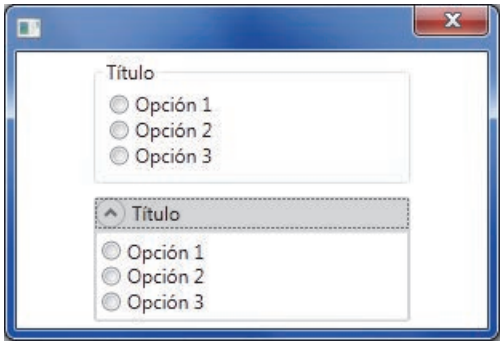
```
Private Sub btopDecOcHex_Checked(sender As System.Object,
e As System.Windows.RoutedEventArgs)
```

```
    If btopDecimal.IsChecked = True Then
      ctDato.Text = Convert.ToString(numeroActual, 10)
    ElseIf btopOctal.IsChecked = True Then
      ctDato.Text = Convert.ToString(numeroActual, 8)
    ElseIf btopHex.IsChecked = True Then
      ctDato.Text = Convert.ToString(numeroActual, 16).ToUpper()
    End If
    ctDato.Focus()
  End Sub
```

El método **ToString** de la clase **Convert** convierte un entero en la cadena de caracteres equivalente expresada en la base especificada y el método **ToUpper** de la clase **String** convierte a mayúsculas una cadena de caracteres.

GroupBox y Expander

El control **GroupBox** representa un control que crea un contenedor que tiene un borde y un encabezado para el contenido y el control **Expander** es análogo, pero tiene una ventana contraíble que muestra el contenido. La figura siguiente muestra ambos.



El código XAML que da lugar al control **GroupBox** de la figura anterior es:

```
<GroupBox Grid.Row="0" Header="Título"
          VerticalAlignment="Center" Width="200" >
  <Border Margin="2" Background="White" Padding="3">
    <StackPanel Name="stackPanel1" >
      <RadioButton Name="rb1" Content="Opción 1" />
      <RadioButton Name="rb2" Content="Opción 2" />
      <RadioButton Name="rb3" Content="Opción 3" />
    </StackPanel>
  </Border>
</GroupBox>
```

y el código XAML que da lugar al control **Expander** de la figura anterior es:

```
<Expander Grid.Row="1" Header="Título" VerticalAlignment="Center"
          Width="200" Background="LightGray">
  <Border Margin="2" Background="White" Padding="3">
    <StackPanel Name="stackPanel11" >
      <RadioButton Name="rb4" Content="Opción 1" />
      <RadioButton Name="rb5" Content="Opción 2" />
      <RadioButton Name="rb6" Content="Opción 3" />
    </StackPanel>
  </Border>
</Expander>
```

Listas simples

Una *lista* es un control de la clase **ListBox** que pone a disposición del usuario un conjunto de elementos, que podrá elegir haciendo clic en ellos. Los elementos de un **ListBox** son objetos **ListBoxItem**. Por ejemplo:

```
<ListBox Name="listBoxColores"
         SelectionChanged="listBoxColores_SelectionChanged">
  <ListBoxItem>Rojo</ListBoxItem>
  <ListBoxItem>Verde</ListBoxItem>
  <ListBoxItem>Azul</ListBoxItem>
</ListBox>
```

La selección de los elementos puede ser simple o múltiple, dependiendo del valor de la propiedad **SelectionMode**:

- *Single*. Sólo puede seleccionar un elemento cada vez.
- *Multiple*. Se pueden seleccionar varios elementos sin mantener presionada una tecla modificadora.
- *Extended*. Se pueden seleccionar varios elementos consecutivos manteniendo presionada la tecla *Mayús* o elementos no consecutivos manteniendo presionada la tecla *Ctrl*.

Resumiendo, un objeto **ListBox** es una colección de objetos **ListBoxItem** derivada de **ItemsControl** (véase el capítulo *Introducción a WPF*). A su vez, **ListBoxItem** se deriva de **ContentControl**, lo que significa que puede contener un solo elemento anidado de cualquier tipo (**String**, **Image**, **CheckBox**, panel, etc.). Si ese elemento es un **UIElement** éste se mostrará sin más en el **ListBox**, ahora si se trata de un elemento de algún otro tipo, el **ListBoxItem** llamará al método **ToString** y mostrará el texto resultante. Por ejemplo, el siguiente código XAML define una lista con dos elementos de tipo **StackPanel**, cada uno de los cuales contiene una imagen y una etiqueta:

```
<ListBox Name="listBox2" Grid.Row="2"
         SelectionChanged="listBox2_SelectionChanged">
  <StackPanel Orientation="Horizontal">
    <Image Source="imagen1.jpg" Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">Imagen 1</Label>
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <Image Source="imagen2.jpg" Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">Imagen 2</Label>
  </StackPanel>
</ListBox>
```

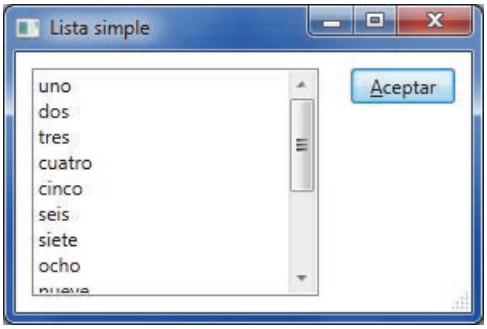
Cuando se añade un elemento a un **ListBox** y no se especifica explícitamente que ese elemento es de tipo **ListBoxItem**, **ListBox** crea implícitamente el objeto **ListBoxItem** con el contenido correspondiente al elemento añadido. Por ejemplo, el primer elemento de la lista anterior sería equivalente al siguiente:

```
<ListBoxItem>
  <StackPanel Orientation="Horizontal">
    <Image Source="imagen1.jpg" Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">Imagen 1</Label>
  </StackPanel>
</ListBoxItem>
```

Este otro ejemplo que se muestra a continuación define una lista con tres elementos de tipo **CheckBox**:

```
<ListBox Name="listBox1"
         SelectionChanged="listBox1_SelectionChanged"
         CheckBox.Click="CheckBox_Click">
  <CheckBox Margin="3">Opción 1</CheckBox>
  <CheckBox Margin="3">Opción 2</CheckBox>
  <CheckBox Margin="3">Opción 3</CheckBox>
</ListBox>
```

Cuando la cantidad de elementos de una lista rebasa el número de los que pueden ser visualizados simultáneamente en el espacio disponible, aparecerá automáticamente una barra de desplazamiento vertical para que el usuario pueda desplazar los elementos de la lista hacia arriba o hacia abajo.



Las propiedades **Items** y **SelectedItems** proporcionan acceso a las dos colecciones que utiliza un control **ListBox**. La primera contiene todos los elementos de la lista y la segunda contiene los elementos seleccionados.

Hay dos maneras de llenar la lista de elementos. La más sencilla consiste en añadirlos directamente a la colección **Items** en el propio código XAML. Sin embargo, en WPF, es más común utilizar un enlace a los datos. En este caso, se establece la propiedad **ItemsSource** de la lista con el objeto que tiene la colección de elementos de datos que se desea mostrar (los enlaces a los datos serán estudiados en un capítulo posterior).

Para añadir elementos a la colección **Items** durante la ejecución se puede utilizar el método **Add**, que permite añadir un elemento cada vez. Por ejemplo:

```
listBoxColores.Items.Add("Verde")
```

Para añadir elementos a la colección **Items** durante la ejecución por medio de la propiedad **ItemsSource**, podemos crear la colección de elementos y después, asignarla a la propiedad **ItemsSource** (podemos clasificarla previamente, si procede). Por ejemplo:

```
Dim elementos As New List(Of String)()
elementos.Add("Rojo")
elementos.Add("Verde")
elementos.Add("Azul")
elementos.Sort()
listBoxColores.ItemsSource = elementos
```

La propiedad **SelectedIndex** permite acceder al índice (a partir de 0) del elemento actualmente seleccionado en la lista y la propiedad **SelectedItem** es una referencia a este elemento, que puede ser **Nothing** si no hay ningún elemento seleccionado; cuando la selección es múltiple, esta propiedad hace referencia al primer elemento de la selección actual. También, cuando seleccionamos un elemento de la lista, se modifica su propiedad **SelectedIndex** para almacenar el índi-

ce del nuevo elemento seleccionado y esto hace que se produzca el evento **SelectionChanged**. Cuando no hay ningún elemento seleccionado, la propiedad **SelectedIndex** toma el valor -1, y cuando hay varios elementos seleccionados, esta propiedad almacena el índice del primer elemento de la selección.

Diseñar la lista

Generalmente, una lista es apropiada cuando se quiere limitar la entrada a una serie determinada de elementos. Como ejemplo, cargue la aplicación denominada *CajasDeDialogo* y añada una nueva caja de diálogo como la de la figura anterior, de tal forma que cuando seleccione uno o más elementos de la lista y pulse el botón *Aceptar*, muestre su contenido en un diálogo predefinido. La caja de diálogo será un objeto de la clase *DlgListaSimple* derivada de **Window**. La tabla de controles y sus propiedades puede ser la siguiente:

Objeto	**Propiedad**	**Valor**
Caja de diálogo	Name Title ShowInTaskbar ResizeMode WindowStartupLocation	DlgListaSimple Lista simple False CanResizeWithGrip CenterOwner
Lista	Name	lsLista1
Botón de pulsación	Name Content	btAceptar _Aceptar

Una vez finalizado el diseño, podremos observar que se ha añadido a la aplicación una nueva clase *DlgListaSimple* a partir de la cual podremos crear ese tipo de cajas de diálogo. A continuación puede ver el código XAML necesario para añadir una lista fija a un diálogo:

```
<Window x:Class="DlgListaSimple" ...
  Title="Lista simple" Height="200" Width="300"
  ShowInTaskbar="False" ResizeMode="CanResizeWithGrip"
  WindowStartupLocation="CenterOwner" >
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <ListBox Name="lsLista1" Margin="10" />
    <Button Grid.Column="1" Name="btAceptar" Content="_Aceptar"
            VerticalAlignment="Top" Margin="10" IsDefault="True"
            Padding="10,1" />
  </Grid>
</Window>
```

Observe que, en este caso, se permite redimensionar la caja de diálogo. Cuando esto suceda, la lista se ajustará proporcionalmente al nuevo tamaño del diálogo y el botón permanecerá en su sitio. Esto se consigue con la definición de dos columnas en la rejilla, la segunda de tamaño automático.

Para mostrar un diálogo de la clase *DlgListaSimple*, añadiremos al menú *Diálogos* de la aplicación *CajasDeDialogo* una nueva orden identificada por *DialogoListaSimple*. Para ello, proceda de igual forma que cuando añadió las órdenes para mostrar los otros diálogos. El controlador asociado con esta orden será el siguiente:

```
Private Sub DialogoListaSimple_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dlg As New DlgListaSimple()
  dlg.ShowDialog()
End Sub
```

Iniciar la lista

La iniciación de una lista la podemos hacer durante el diseño, a través de su propiedad **Items**, o durante la ejecución, por ejemplo respondiendo al evento **Loaded**, a través de su propiedad **Items** o **ItemsSource**. Como ejemplo, vamos a añadir a la lista una serie de elementos de tipo **String**. Para ello, añadimos a la clase *DlgListaSimple* un método privado *DlgListaSimple_Loaded* que se ejecute como respuesta al evento **Loaded** del diálogo:

```
Private Sub DlgListaSimple_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim elemento As String() = New String() {
    "uno", "dos", "tres", "cuatro", "cinco", "seis", _
    "siete", "ocho", "nueve", "diez", "once", "doce", _
    "trece", "catorce"}
  Me.lsLista1.ItemsSource = elemento
End Sub
```

Este método inicia la lista *lsLista1* con los valores de la matriz *elemento* asignando esta matriz a la propiedad **ItemsSource** de dicha lista.

Acceder a los elementos seleccionados

Para seleccionar un elemento de una lista, haga clic sobre él. Para seleccionar varios, sólo si la propiedad **SelectionMode** lo permite, mantenga pulsada la tecla *Ctrl* o *Mayús* (*Shift*) y haga clic sobre los elementos que desea seleccionar.

Para acceder al elemento seleccionado de una lista, utilizaremos la propiedad **SelectedItem** de la clase **ListBox**. Por ejemplo, vamos a añadir a la clase *DlgLis-*

taSimple el controlador del evento **Click** del botón *Aceptar* para que cuando se ejecute, muestre el valor del elemento seleccionado:

```
Private Sub btAceptar_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  If lsLista1.SelectedIndex < 0 Then Return
  Dim elementoSeleccionado As Object
  elementoSeleccionado = lsLista1.SelectedItem
  MessageBox.Show(elementoSeleccionado.ToString())
End Sub
```

Si son varios los elementos seleccionados (propiedad **SelectionModel** = *Extended*; el valor por omisión es *Single*: sólo se puede seleccionar un elemento cada vez), todos pueden ser obtenidos por medio de la colección **SelectedItems**. Por ejemplo, para este caso, podemos modificar el método *btAceptar_Click* para que muestre todos los elementos seleccionados así:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  If lsLista1.SelectedIndex < 0 Then Return
  Dim s As String = ""
  For Each s1 As String In lsLista1.SelectedItems
    s = (s & s1) + Environment.NewLine
  Next
  MessageBox.Show(s)
End Sub
```

Colección de elementos de una lista

Anteriormente dijimos que la colección referenciada por la propiedad **Items** de un objeto **ListBox** se corresponde con los elementos mostrados por esa lista. Por lo tanto, las operaciones que realicemos sobre el contenido de esa colección, para añadir, insertar o quitar uno o más elementos, se reflejarán automáticamente en la lista. Dicha colección es de la clase **ItemCollection**.

Echando una ojeada a la funcionalidad proporcionada por esta clase, observamos que el método **Add** permite añadir un solo objeto a la colección y el método **Insert** insertar un objeto en una ubicación específica de la colección. Para quitar un elemento determinado, proporciona el método **Remove**, o bien **RemoveAt** cuando la ubicación del elemento en la colección es conocida, o **Clear** si lo que se quiere es quitar todos los elementos de la colección.

Por ejemplo, vamos a modificar la caja de diálogo *DlgListaSimple* para que permita realizar las operaciones anteriormente descritas. Para ello:

1. Eliminamos el controlador del evento **Loaded** del diálogo *DlgListaSimple*. Tenga presente que la operación de añadir elementos a la colección **Items** no es válida mientras **ItemsSource** esté en uso.

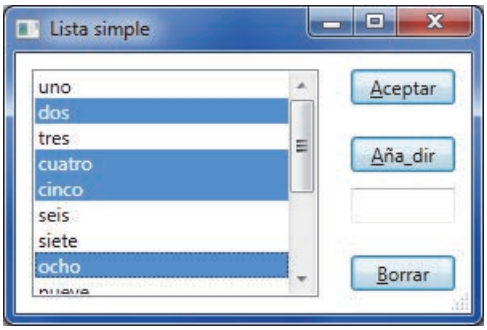
2. Añadimos al diálogo otro botón, *btAñadir*, y una caja de texto, *ctAñadir*, que permita al usuario introducir un elemento para añadirlo a la lista. Después, procediendo de forma análoga a como hizo para el botón *Aceptar*, añada el controlador para manejar el evento **Click** de *btAñadir* según se muestra a continuación:

```
Private Sub btAñadir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Añadir el elemento introducido en la
  ' caja de texto ctAñadir
  If ctAñadir.Text.Length <> 0 Then
    lsLista1.Items.Add(ctAñadir.Text)
  End If
End Sub
```

3. Finalmente, añadimos otro botón, *btBorrar*, que permita al usuario borrar un elemento seleccionado de la lista. Después, procediendo de forma análoga a como hizo para el botón *Añadir*, añada el controlador para manejar el evento **Click** de *btBorrar* según se muestra a continuación:

```
Private Sub btBorrar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  If lsLista1.SelectedIndex < 0 Then Return
  lsLista1.Items.RemoveAt(lsLista1.SelectedIndex)
End Sub
```

Compile la aplicación, ejecútela y analice los resultados.



Lista de elementos de tipo CheckBox

Según hemos visto anteriormente, un objeto de la clase **CheckBox** puede ser un elemento de un **ListBox**:

```
<ListBox Name="listBox1"
         SelectionChanged="listBox1_SelectionChanged">
  <CheckBox Margin="3">Opción 1</CheckBox>
  <CheckBox Margin="3">Opción 2</CheckBox>
  <CheckBox Margin="3">Opción 3</CheckBox>
</ListBox>
```

Para trabajar con el elemento seleccionado de la lista, añadimos el controlador que responda al evento **SelectionChanged**:

```
Private Sub listBox1_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  ' Respuesta al evento SelectionChanged
End Sub
```

En general, cuando se trabaja con una lista que tiene elementos de cualquier tipo, hay un detalle a tener en cuenta: cuando hace clic en un elemento de la lista y se obtiene el valor de **SelectedItem**, no podrá ver los objetos **ListBoxItem**; en su lugar, verá el objeto colocado en la lista.

Utilizando elementos **CheckBox** hay, además, otro detalle a tener en cuenta. Cuando se hace clic en la casilla de verificación o en su contenido, el elemento padre **ListBoxItem** no se selecciona y, por lo tanto, no se genera el evento **SelectionChanged**, pero si se hace clic en el espacio en blanco junto a la casilla de verificación, todo funciona como se esperaba. Esto es así porque, además del seguimiento del elemento seleccionado actualmente, hay que realizar también el seguimiento de si la casilla está o no seleccionada. Una solución al problema planteado es añadir el controlador para el evento adjunto **CheckBox.Click** generado cuando se hace clic en la casilla de verificación y, a continuación, provocar la generación del evento **SelectionChanged** desde este controlador.

Según lo expuesto, para trabajar con una lista como la expuesta, tendremos que manipular dos eventos: el evento adjunto **CheckBox.Click** procedente del objeto **CheckBox** y el evento **SelectionChanged** de la lista:

```
<ListBox Name="listBox1"
         SelectionChanged="listBox1_SelectionChanged"
         CheckBox.Click="CheckBox_Click">
  <CheckBox Margin="3">Opción 1</CheckBox>
  <CheckBox Margin="3">Opción 2</CheckBox>
  <CheckBox Margin="3">Opción 3</CheckBox>
```

```
</ListBox>
```

Añadimos, entonces, el controlador del evento adjunto **CheckBox.Click** que tendrá como misión provocar la generación del evento **SelectionChanged** siempre que se produzca un cambio del elemento seleccionado de la lista, el cual es referenciado por la propiedad **SelectedItem**:

```
Private Sub CheckBox_Click(sender As Object, e As RoutedEventArgs)
  If TypeOf e.OriginalSource Is CheckBox Then
    listBox1.SelectedItem = e.OriginalSource
  End If
End Sub
```

Observe la línea sombreada; asigna a **SelectedItem** el elemento **CheckBox** seleccionado (el valor inicial de esta propiedad es **Nothing**). Entonces, cada vez que se realice una asignación que haga que **SelectedItem** cambie de valor se generará el evento **SelectionChanged** y, además, la propiedad **IsChecked** de la casilla de verificación nos dará información de si ésta está o no seleccionada. El código siguiente muestra cómo acceder al índice del elemento seleccionado y a la propiedad **IsChecked** del objeto **CheckBox** sobre el que se hizo clic.

```
Private Sub CheckBox_Click(sender As Object, e As RoutedEventArgs)
  If TypeOf e.OriginalSource Is CheckBox Then
    lsLista1.SelectedItem = e.OriginalSource
  End If
  If lsLista1.SelectedItem Is Nothing Then
    Return
  End If

  Dim s As String = String.Format(
    "La posición del elemento elegido es {0} y su estado es {1}.",
    lsLista1.SelectedIndex,
    DirectCast(lsLista1.SelectedItem, CheckBox).IsChecked)
  MessageBox.Show(s)
End Sub
```

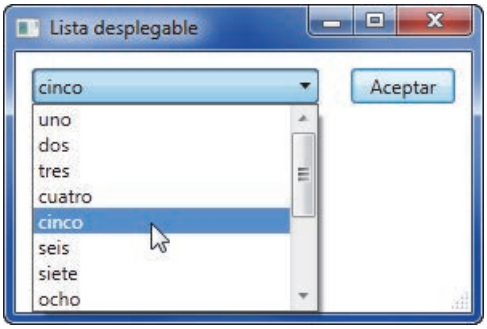
La línea sombreada invocará al controlador *listBox1_SelectionChanged* cada vez que el valor de **SelectedItem** cambie a otro elemento porque el elemento seleccionado en la lista ha cambiado.

Listas desplegables

Una lista desplegable es un objeto de la clase **ComboBox**. Al igual que **ListBox**, se deriva de **ItemsControl** y sus elementos son de tipo **ComboBoxItem**. La diferencia entre una *lista simple* y una *lista desplegable* (también denominada cuadro

combinado) es que la *lista desplegable* es una combinación de una lista simple y una caja de texto, y que permite la selección de un solo elemento cada vez.

El comportamiento del **ComboBox** queda definido por las propiedades **IsEditable** e **IsReadOnly**, cuyos valores pueden ser **True** o **False**. **IsReadOnly** sólo tiene efecto si **IsEditable** es **True**. El comportamiento se refiere a si se puede o no escribir una cadena para seleccionar un elemento del **ComboBox**, escribir una cadena que no se corresponda con ningún elemento, o realizar las operaciones de cortar, copiar y pegar sobre la caja de texto.



El alto que muestra la lista emergente está predeterminado, pero puede modificar este valor por medio de la propiedad **MaxDropDownHeight**. Se trata de un valor de tipo **Double** que se interpreta como unidades independientes del dispositivo (1 unidad = 1/96 pulgadas).

La propiedad **Items** proporciona acceso a la colección que utiliza el **ComboBox** para almacenar todos los elementos de la lista. Para añadir elementos a la colección **Items** durante la ejecución, proceda de igual forma que para las listas simples; esto es, añadiendo elementos a la colección **Items** o asignando una colección de elementos a la propiedad **ItemsSource**.

La propiedad **SelectedIndex** permite acceder al índice (a partir de 0) del elemento actualmente seleccionado en la lista, la propiedad **SelectedItem** es una referencia a este elemento y la propiedad **Text** especifica la cadena que se muestra en la caja de texto. Cuando seleccionamos un elemento de la lista, se modifica su propiedad **SelectedIndex** para almacenar el índice del nuevo elemento seleccionado; esto hace que se produzca el evento **SelectionChanged**. Cuando no hay ningún elemento seleccionado, la propiedad **SelectedIndex** vale -1.

Diseñar la lista

Como ejemplo, cargue la aplicación denominada *CajasDeDialogo* y añada una nueva caja de diálogo como la de la figura anterior, de tal forma que cuando se-

leccione un elemento de la lista y pulse el botón *Aceptar*, muestre su contenido en un diálogo predefinido. La caja de diálogo será un objeto de la clase *DlgListaDesplegable* derivada de **Window**. La tabla de controles y sus propiedades puede ser la siguiente:

Objeto	**Propiedad**	**Valor**
Caja de diálogo	Name Title ShowInTaskbar ResizeMode WindowStartupLocation	DlgListaDesplegable Lista desplegable False CanResizeWithGrip CenterOwner
Lista desplegable	Name Margin VerticalAlignment MaxDropDownHeight	ldLista1 10 Top 130
Botón de pulsación	Name Content	btAceptar &Aceptar

Una vez finalizado el diseño, podremos observar que se ha añadido a la aplicación una nueva clase *DlgListaDesplegable* a partir de la cual podremos crear ese tipo de cajas de diálogo. A continuación puede ver el código XAML necesario para añadir una lista desplegable a un diálogo:

```
<Window x:Class="DlgListaDesplegable"
  ...
  Title="Lista desplegable" Height="200" Width="300"
  ShowInTaskbar="False" ResizeMode="CanResizeWithGrip"
  WindowStartupLocation="CenterOwner" >
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <ComboBox Name="ldLista1" Margin="10" VerticalAlignment="Top"
              MaxDropDownHeight="130"/>
    <StackPanel Grid.Column="1">
      <Button Name="btAceptar" Content="_Aceptar"
              VerticalAlignment="Top" Margin="10"
              IsDefault="True" Padding="10,1" />
    </StackPanel>
  </Grid>
</Window>
```

Para mostrar este diálogo desde el menú *Diálogos* de la clase *CajasDeDialogo*, añada al mismo una orden identificada por *DialogoListaDesplegable*. Para

ello, proceda de igual forma que cuando añadió las órdenes para mostrar los otros diálogos.

```
Private Sub DialogoListaDesplegable_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dlg As New DlgListaDesplegable()
  dlg.ShowDialog()
End Sub
```

Iniciar la lista

Un **ComboBox**, igual que sucedía con un **ListBox**, puede contener una colección de objetos de cualquier tipo. Igualmente, la colección de elementos de un **ComboBox** se puede asignar a través de su propiedad **Items** o **ItemsSource**.

Acceder al elemento seleccionado

Para acceder al elemento seleccionado de una lista desplegable (el que se muestra en la caja de texto), utilizaremos el método **SelectedItem** de la clase **ComboBox**. Por ejemplo, vamos a añadir a la clase *DlgListaDesplegable* el controlador del evento **Click** del botón *Aceptar* para que cuando se ejecute se muestre el valor del elemento seleccionado:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  If ldLista1.SelectedIndex < 0 Then Return
  Dim elementoSeleccionado As Object
  elementoSeleccionado = ldLista1.SelectedItem
  MessageBox.Show(elementoSeleccionado.ToString())
End Sub
```

Este método será invocado cuando el usuario haga clic en el botón *btAceptar*.

Colección de elementos de una lista desplegable

Anteriormente dijimos que la colección referenciada por la propiedad **Items** de un objeto **ComboBox** se corresponde con los elementos mostrados por esa lista. Por lo tanto, las operaciones que realicemos sobre el contenido de esa colección, para añadir, insertar o quitar uno o más elementos, se reflejarán automáticamente en la lista. Dicha colección es de la clase **ItemCollection**.

Por ejemplo, vamos a modificar la caja de diálogo *DlgListaDesplazable* para que permita realizar las operaciones anteriormente descritas. Para ello:

1. Eliminamos el controlador del evento **Loaded** del diálogo *DlgListaDesplegable*, si es que anteriormente lo utilizó para iniciar la lista. Tenga presente que la operación de añadir elementos a la colección **Items** no es válida mientras **ItemsSource** esté en uso.

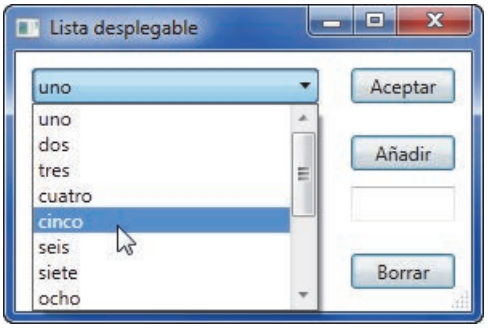
2. Añadimos al diálogo otro botón, *btAñadir*, y una caja de texto, *ctAñadir*, que permita al usuario añadir un elemento a la lista. Después, procediendo de forma análoga a como hizo para otros botones, añada el controlador para manejar el evento **Click** de *btAñadir* según se muestra a continuación:

```
Private Sub btAñadir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Añadir el elemento introducido en la
  ' caja de texto ctAñadir
  If ctAñadir.Text.Length <> 0 Then
    ldLista1.Items.Add(ctAñadir.Text)
  End If
End Sub
```

3. Finalmente, añadimos otro botón, *btBorrar*, que permita al usuario borrar un elemento de la lista. Después, procediendo de forma análoga a como hizo para otros botones, añada el controlador para manejar el evento **Click** de *btBorrar* según se muestra a continuación:

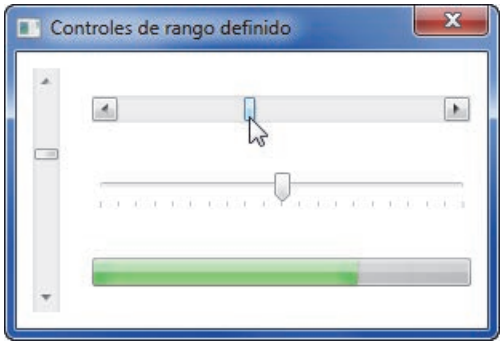
```
Private Sub btBorrar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  If ldLista1.SelectedIndex < 0 Then Return
  ldLista1.Items.RemoveAt(ldLista1.SelectedIndex)
End Sub
```

La propiedad **SelectedIndex** de **ComboBox** devuelve el índice del elemento seleccionado en la lista (el primer índice es el 0; -1 si no hay ningún elemento seleccionado) y el método **RemoveAt** de **ItemCollection** elimina el elemento de la lista que tenga por índice el pasado como argumento.



Controles de rango definido

Como su nombre indica, se trata de controles que definen un valor **Double** perteneciente a un rango previamente definido. A este grupo pertenecen los controles **ScrollBar**, **Slider** y **ProgressBar** que, en este orden, podemos ver en la figura siguiente. Las clases correspondientes heredan todas de la clase **RangeBase** que a su vez hereda de **Control**.



Todos estos controles de WPF definen básicamente las siguientes propiedades heredadas de la clase **RangeBase**:

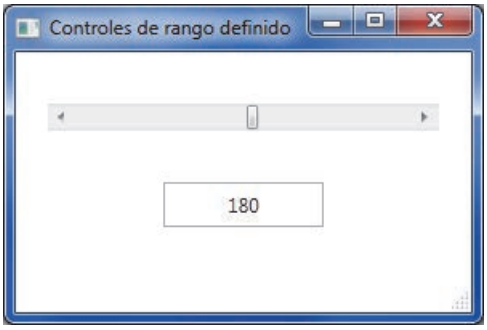
- **Minimum** Valor del extremo inferior del rango de valores.
- **Maximum** Valor del extremo superior del rango de valores.
- **Value** Valor actualmente seleccionado en el rango representado, especificado por el cursor del control.
- **SmallChange** Valor mínimo que se desplaza el cursor cuando se hace clic sobre el control.
- **LargeChange** Valor máximo que se desplaza el cursor cuando se hace clic sobre el control.

Cuando se modifica el valor de la propiedad **Value**, el control correspondiente generará un evento **ValueChanged**.

ScrollBar

Un objeto **ScrollBar** es una barra de desplazamiento horizontal o vertical. Se trata de un control que tiene un cursor (un cuadrado) que se desplaza a lo largo de la misma para fijar un valor. La posición más a la izquierda o más arriba se corresponde con el valor mínimo, la posición más a la derecha o más abajo se corresponde con el valor máximo y cualquier otra posición es un valor entre estos dos. La propiedad **Orientation** de **ScrollBar** permite definir la orientación vertical u horizontal de la barra de desplazamiento.

Como ejercicio, añadiremos al menú *Diálogos* de la aplicación *CajasDeDialogo* anterior un nuevo elemento identificado por *DialogoRangos* de forma que cuando el usuario haga clic sobre él, se visualice una caja de diálogo como la que muestra la figura siguiente. Dicha caja de diálogo muestra un contador ascendente que es actualizado por un temporizador (véase el apartado *Temporizadores y modelo de subprocesos* al final de este capítulo).



La actualización del contador se hará cada 10 * *k* milisegundos, donde *k* será un valor entre 1 y 150 proporcionado por la barra de desplazamiento que muestra la figura. Esto es, la velocidad de conteo será menor cuanto más a la derecha esté el cuadrado de desplazamiento.

De acuerdo con el planteamiento del problema, pasamos a diseñar esa caja de diálogo. Asígnele el título "Controles de rango definido" y ponga sobre ella los controles con las propiedades que se especifican en la tabla siguiente.

Objeto	**Propiedad**	**Valor**
Caja de diálogo	Name Title ShowInTaskbar ResizeMode WindowStartupLocation	DlgRangos Controles de rango definido False CanResizeWithGrip CenterOwner
Barra de desplazamiento	Name Orientation VerticalAlignment Minimum Maximum Value LargeChange SmallChange	bdhIntervalo Horizontal Center 1 151 76 30 1

Etiqueta	Name	etContador
	Content	0
	VerticalAlignment	Top
	Width	100
	Height	28
	HorizontalContentAlignment	Center

Una vez finalizado el diseño, podremos observar que se ha añadido a la aplicación una nueva clase *DlgRangos* a partir de la cual podremos crear ese tipo de cajas de diálogo. A continuación puede ver el código XAML necesario para añadir una barra de desplazamiento horizontal a un diálogo:

```
<Window x:Class="DlgRangos"
  ...
  Title="Controles de rango definido" Height="200" Width="300"
  ShowInTaskbar="False" ResizeMode="CanResizeWithGrip"
  WindowStartupLocation="CenterOwner">
    <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <ScrollBar Name="bdhIntervalo" Orientation="Horizontal"
               VerticalAlignment="Center" Margin="20" Maximum="151"
               Minimum="1" SmallChange="1" LargeChange="30"
               Value="76" />
    <Label Grid.Row="1" Name="etContador" Content="0"
           VerticalAlignment="Top" Height="28" Width="100"
           HorizontalContentAlignment="Center" BorderThickness="1"
           BorderBrush="DarkGray" />
  </Grid>
</Window>
```

Para mostrar este diálogo desde el menú *Diálogos* de la clase *CajasDeDialogo*, añada al mismo una orden denominada *DialogoRangos*. Para ello, proceda de igual forma que cuando añadió las otras órdenes para mostrar los otros diálogos.

```
Private Sub DialogoRangos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dlg As New DlgRangos()
  dlg.ShowDialog()
End Sub
```

¿Para qué necesitamos la barra de desplazamiento? Para variar el retardo del temporizador entre 10×1 y 10×150 milisegundos (1 ms = 10.000 ticks) con el fin de variar la velocidad de conteo. Según esto, añada a la clase *DlgRangos* un tem-

porizador y el controlador que responda a su evento **Tick**. Este método tiene que actualizar el valor de la etiqueta *etContador* incrementándolo en una unidad.

```
Imports System.Windows.Threading

Public Class DlgRangos
  Private temporizador As DispatcherTimer

  Public Sub New()
    CrearTemporizador()
    InitializeComponent()
    temporizador.Start()
  End Sub

  Private Sub CrearTemporizador()
    temporizador = New DispatcherTimer()
    temporizador.Interval = New TimeSpan(0, 0, 1)
    AddHandler temporizador.Tick, AddressOf Temporizador_Tick
  End Sub

  Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
    etContador.Content =
      (Convert.ToInt32(etContador.Content) + 1).ToString()
  End Sub
End Class
```

¿A intervalos de cuántos milisegundos se producirá el evento **Tick**? Esto es lo que indicará la barra de desplazamiento. Según esto, añada a la clase *DlgRangos* un controlador de eventos **ValueChanged** que responda a las variaciones del cuadrado de desplazamiento que el usuario haga sobre la barra de desplazamiento y complételo como se indica a continuación:

```
Private Sub bdhIntervalo_ValueChanged(sender As Object,
e As RoutedPropertyChangedEventArgs(Of Double))
  temporizador.Interval =
    New TimeSpan(100000 * CLng(bdhIntervalo.Value)) ' ticks
End Sub
```

Como se puede observar, un objeto **ScrollBar** genera eventos **ValueChanged** siempre que experimenta un cambio. Lo que en realidad sucede es que al variar el cuadrado de desplazamiento se genera un evento **Scroll**, se actualiza la propiedad **Value** y, como consecuencia, se genera el evento **ValueChanged**. O sea, que podríamos haber utilizado el evento **Scroll** en lugar de **ValueChanged** para establecer el intervalo de tiempo del temporizador.

Slider

Un objeto **Slider** es un control deslizante o deslizador. Igual que una barra de desplazamiento, un deslizador, dependiendo del valor de su propiedad **Orientation** (*Horizontal* o *Vertical*), puede tener orientación horizontal o vertical y, además, se puede mejorar su apariencia con marcas. El espaciado de estas marcas viene determinado por la propiedad **TickFrequency** (valor **Double**) y su posición por **TickPlacement** (*None*, *Topleft*, *BottomRight*, *Both*).

Como ejemplo, vamos a modificar la caja de diálogo anterior como muestra la figura siguiente:



Se puede observar que hemos cambiado la barra de desplazamiento por un deslizador y tres etiquetas con las siguientes propiedades:

Objeto	**Propiedad**	**Valor**
Deslizador	Name Minimum Maximum Value LargeChange SmallChange TickPlacement IsTabStop	desIntervalo 1 151 76 30 5 BottomRight False
Etiqueta	Content HorizontalAlignment	Rápido Left
Etiqueta	Content HorizontalAlignment	Normal Center
Etiqueta	Content HorizontalAlignment	Lento Right

Una vez finalizado el diseño, podremos observar en la clase *DlgRangos* el código XAML necesario para añadir un deslizador a un diálogo:

```
<Window x:Class="DlgRangos"
  ...
  Title="Controles de rango definido" Height="200" Width="300"
  ShowInTaskbar="False" ResizeMode="NoResize"
  WindowStartupLocation="CenterOwner">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
    <StackPanel VerticalAlignment="Center" Margin="10">
      <Slider Name="desIntervalo" Minimum="1" Maximum="151"
              SmallChange="1" LargeChange="30" TickFrequency="5"
              Value="76" TickPlacement="BottomRight"
              AutoToolTipPlacement="None" IsTabStop="False"
              ValueChanged="desIntervalo_ValueChanged" />
      <WrapPanel ItemWidth="86" >
        <Label Content="Rápido" HorizontalAlignment="Left" />
        <Label Content="Normal" HorizontalAlignment="Center" />
        <Label Content="Lento" HorizontalAlignment="Right" />
      </WrapPanel>
    </StackPanel>
      <Label Grid.Row="1" Name="etContador" Content="0" Height="28"
             VerticalAlignment="Top" Width="100"
             HorizontalContentAlignment="Center"
             BorderThickness="1" BorderBrush="DarkGray" />
  </Grid>
</Window>
```

Finalmente, modifique el controlador del evento **ValueChanged** para que ahora haga referencia al control *desIntervalo*.

```
Private Sub desIntervalo_ValueChanged(sender As Object,
e As RoutedPropertyChangedEventArgs(Of Double))
  temporizador.Interval =
    New TimeSpan(100000 * CLng(desIntervalo.Value)) ' ticks
End Sub
```

ProgressBar

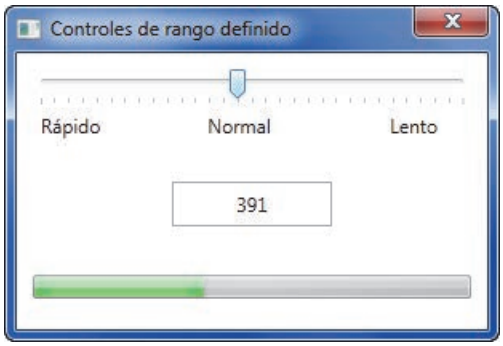
Un objeto **ProgressBar** es una barra de progreso; esto es, un rectángulo de una longitud arbitraria, del cual, en un instante determinado, estará coloreado un tanto por ciento en función de la cantidad ya finalizada de un proceso por el que estamos esperando, o bien puede ser un rectángulo que muestra un patrón de repetición; la elección de uno u otro modelo lo determina la propiedad **IsIndeterminate**

de tipo **Boolean**. Es una forma de indicar al usuario que una aplicación no está bloqueada, sino que está realizando una tarea larga.

Para el caso de una barra que se rellena en función de un valor, las propiedades **Minimum** y **Maximum** definen el intervalo de valores que representarán el progreso de una tarea. Normalmente, la propiedad **Minimum** se pone a valor 0 y **Maximum** al valor que indica que la tarea ha terminado. La propiedad **Value** representa el progreso que la aplicación realiza para terminar la operación.

Como ejemplo, vamos a añadir a la caja de diálogo anterior una barra de progreso, que supervise una cuenta de 0 a 1000, con las propiedades indicadas a continuación. Cuando se alcance el valor 1000, la cuenta finalizará. Este valor podría corresponderse, por ejemplo, con el número de ficheros que tiene que leerse para instalar una determinada aplicación.

Objeto	**Propiedad**	**Valor**
Barra de progreso	Name Minimum Maximum Value	bpCuenta 0 100 0



Obsérvese que hemos establecido como valor máximo 100 para la barra. Esto lo hacemos así porque la resolución de la barra, en función del rango de valores a representar, puede ser baja y, por lo tanto, habrá muchos valores en ese rango que no hagan ningún cambio visual en la misma, pero sí consumen tiempo de UCP. Por eso, la mejor solución es especificar la cantidad ya finalizada de un proceso en tanto por ciento, de ahí el rango 0 (cero por ciento) a 100 (cien por cien).

Una vez finalizado el diseño, podremos observar en la clase *DlgRangos* el código XAML necesario para añadir una barra de progreso a un diálogo:

```
<ProgressBar Grid.Row="2" Name="bpCuenta" Height="15" Margin="10"
             IsIndeterminate="False"
```

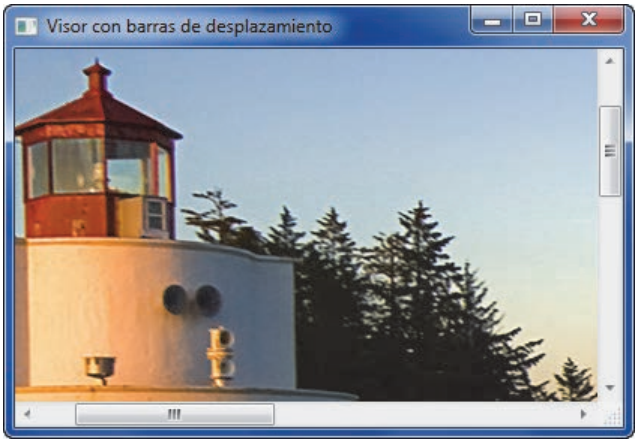
```
Minimum="0" Maximum="100" Value="0" />
```

Finalmente, modifique el controlador del evento **Tick** del temporizador para que, además de incrementar la cuenta, aumente proporcionalmente la barra de progreso:

```
Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
  Dim inc As Integer = 1 ' incremento
  Dim tpHecho As Integer = 0, carga As Integer = 1000
  Dim cuenta As Integer = Convert.ToInt32(etContador.Content) + inc
  etContador.Content = Convert.ToString(cuenta)
  ' Mostrar progreso
  tpHecho = CInt(Math.Truncate(CDbl(cuenta) / carga * 100))
  If tpHecho > bpCuenta.Value Then
    bpCuenta.Value = tpHecho
  End If
  If cuenta = carga Then temporizador.Stop()
End Sub
```

Visor con barras de desplazamiento

Para mostrar el contenido en un área menor que su tamaño real podemos utilizar un objeto **ScrollViewer**. Utilizando este control, cuando el contenido no está completamente visible, se muestran unas barras de desplazamiento que el usuario puede usar para mover el contenido. La ventana siguiente muestra una imagen en un visor con barras de desplazamiento.



El código XAML que da lugar al contenido mostrado por esta ventana es el siguiente:

```
<Grid>
  <ScrollViewer HorizontalScrollBarVisibility="Auto">
    <Image Source="Imagenes/Lighthouse.jpg" />
  </ScrollViewer>
</Grid>
```

El valor (*Disable*, *Auto*, *Hidden* o *Visible*) de las propiedades **VerticalScrollBarVisibility** y **HorizontalScrollBarVisibility** indica si debe mostrarse una barra de desplazamiento horizontal. El valor predeterminado para la primera es *Visible* y para la segunda, *Disable*.

Para desplazar el contenido que muestra la ventana de la figura anterior, puede hacer clic en la barra de desplazamiento con el ratón, puede usar la rueda del ratón, o puede hacer clic en alguna parte de la superficie visible y utilizar las teclas de movimiento del cursor. Si esto no le da la flexibilidad deseada, entonces puede utilizar los métodos de la clase **ScrollViewer** para desplazarse. Por ejemplo, **LineUp** y **LineDown** son equivalentes a un clic en los botones de flecha de la barra de desplazamiento vertical para moverse hacia arriba o hacia abajo una vez. **PageUp** y **PageDown** son equivalentes a hacer clic en la barra de desplazamiento, por encima o por debajo del cursor. Los métodos **LineLeft**, **LineRight**, **PageLeft** y **PageRight** son análogos a los anteriores. Por último, puede utilizar el **ScrollTo***???* para ir a un lugar específico. Por ejemplo, **ScrollToEnd** y **ScrollToHome** para desplazarse a la parte inferior o superior del contenido, o **ScrollToVerticalOffset** para desplazarse a una posición específica.

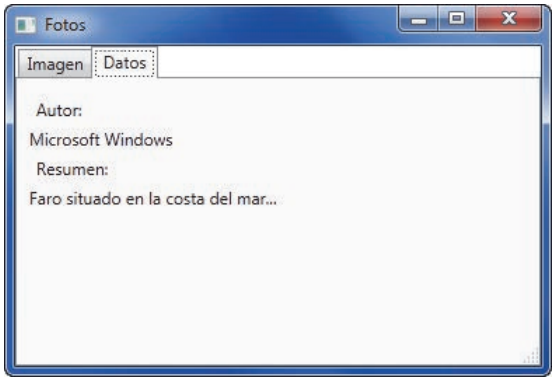
Si **ScrollViewer** contiene un gran número de elementos, puede que el desplazamiento se vea afectado. Para solucionar este problema, asigne a la propiedad **IsDeferredScrollingEnabled** el valor **True**. Esto hace que la vista mostrada permanezca estática mientras se arrastra el cursor de las barras y que sólo se actualice cuando se suelte dicho cursor.

Control con pestañas

Para exponer una gran cantidad de datos minimizando el uso del espacio de pantalla podemos utilizar el elemento **TabControl**. Este objeto está compuesto de varios objetos **TabItem** (un elemento o página con pestaña) que comparten el espacio definido por **TabControl**, de los cuales sólo uno está visible cada vez. Para hacer referencia al elemento que se muestra se utiliza la propiedad **SelectedItem** o **SelectedIndex**. Por ejemplo, el código XAML siguiente daría lugar al contenido de la ventana que se muestra a continuación:

```
<Grid>
  <TabControl>
    <TabItem Header="Imagen">
```

```
      <ScrollViewer HorizontalScrollBarVisibility="Auto">
        <Image Source="Imagenes/Lighthouse.jpg" />
      </ScrollViewer>
    </TabItem>
    <TabItem Header="Datos">
      <StackPanel Margin="5">
        <Label Content="Autor:" />
        <TextBlock Text=... />
        <Label Content="Resumen:" />
        <TextBlock Text=... />
      </StackPanel>
    </TabItem>
  </TabControl>
</Grid>
```



Gestión de fechas

WPF incluye dos controles para manipular fechas: **Calendar** y **DatePicker**. Ambos están diseñados para permitir al usuario elegir una fecha.



El control **Calendar** muestra un calendario que visualiza un solo mes a la vez y permite desplazarse de mes en mes (haciendo clic en los botones de flecha) o saltar a un mes específico (haciendo clic en el encabezado de mes para ver todo un año, y luego hacer clic en el mes).

Calendar, dependiendo del valor de la propiedad **DisplayMode**, puede mostrar los días de un mes, los meses de un año o los años de una década (*Month* muestra un mes cada vez, *Year* muestra un año y *Decade*, diez años) y dependiendo de la propiedad **SelectionMode**, el usuario puede seleccionar una fecha, un intervalo de fechas o varios intervalos de fechas. Las propiedades **DisplayDateStart** y **DisplayDateEnd** determinan el intervalo de fechas mostrado.

La propiedad **SelectedDates** de **Calendar** obtiene la colección de las fechas seleccionadas. Cuando esta colección cambia se produce el evento **SelectedDatesChanged**. Por ejemplo, el siguiente código analiza las fechas seleccionadas en el elemento **Calendar**:

```
Private Sub Calendar_SelectedDatesChanged(sender As System.Object,
e As System.Windows.Controls.SelectionChangedEventArgs)
  ' Verificar los elementos nuevos añadidos
  For Each fechaSeleccionada As DateTime In e.AddedItems
    If (fechaSeleccionada.DayOfWeek = DayOfWeek.Saturday) OrElse
       (fechaSeleccionada.DayOfWeek = DayOfWeek.Sunday) Then
      etError.Text = "Los fines de semana no se pueden seleccionar."
      ' Borrar la fecha no válida
      DirectCast(sender, Calendar).SelectedDates.Remove(fechaSeleccionada)
    End If
  Next
End Sub
```

Para especificar fechas que no se pueden seleccionar se puede utilizar la propiedad **BlackoutDates**. Estas fechas se mostrarán atenuadas y deshabilitadas.

De manera predeterminada, el valor del día actual se mostrará resaltado (propiedad **IsTodayHighlighted** a **True**).

El control **DatePicker** requiere menos espacio que el **Calendar**, según se puede ver en la figura anterior. Es análogo a una lista desplegable que cuando se abre muestra el mes actual, igual que **Calendar** y con la misma funcionalidad. La fecha elegida será mostrada por la caja de texto en formato de fecha largo o corto.

Muchas de las propiedades de **DatePicker** sirven para administrar su objeto **Calendar** integrado y funcionan del mismo modo que la propiedad equivalente de **Calendar**. Por ejemplo, las propiedades **IsTodayHighlighted**, **BlackoutDates**, **DisplayDateStart** y **DisplayDateEnd**, entre otras, funcionan del mismo modo que las propiedades equivalentes de **Calendar**.

La propiedad **SelectedDate** de **DatePicker** obtiene la fecha seleccionada. Cuando ésta cambia se produce el evento **SelectedDateChanged**. También son de interés el evento **CalendarOpened** que se produce cuando se abre el control **Calendar** desplegable, **CalendarClosed** que se produce cuando se cierra el control **Calendar** desplegable y **DateValidationError** que se produce cuando **Text** tiene asignado un valor que no se puede interpretar como una fecha. Por ejemplo, el siguiente código se ejecutará si en la caja de texto del **DatePicker** se escribe una fecha no válida:

```
Private Sub DatePicker_DateValidationError(sender As Object,
e As DatePickerDateValidationErrorEventArgs)
  etError.Text = "'" & Convert.ToString(e.Text) &
    "' no es un valor válido porque " &
    Convert.ToString(e.Exception.Message)
End Sub
```

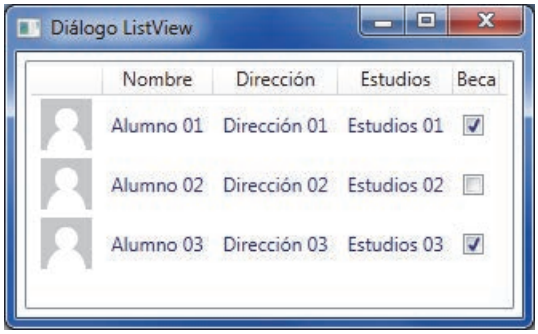
ListView

Un control **ListView** se deriva de **ListBox**, por lo tanto es un **ItemsControl**, lo que significa que contiene una colección de objetos que pueden ser de cualquier tipo. Este control añade una visualización basada en columnas y permite cambiar rápidamente entre diferentes "vistas" por medio de la propiedad **View**. WPF proporciona una "vista" predefinida denominada **GridView** que divide el contenido de los elementos de datos de **ListView** en columnas. Para dar estilo a estas columnas y especificar su contenido utilizaremos las propiedades y métodos de **GridView**. Por ejemplo, el siguiente código XAML define un control **ListView** cuya propiedad **View** hace referencia a una vista **GridView**. Dicha vista define cinco columnas: la primera es para mostrar una foto y las siguientes van encabezadas por los títulos *Nombre*, *Dirección*, *Estudios* y *Beca*. Para definir estas columnas hay que añadir por cada una de ellas un objeto **GridViewColumn** a la colección **Columns** de **GridView**.

```
<Grid DataContext="{Binding Source={StaticResource xmlAlumnos}}">
  <ListView Margin="5" Name="lstAlumnos"
            ItemsSource="{Binding XPath=alumno}">
    <ListView.View>
      <GridView>
        <GridView.Columns>
          <GridViewColumn>
            <GridViewColumn.CellTemplate>
              <DataTemplate>
                <Image Source="{Binding XPath=foto}"
                       Height="32" Width="32"/>
              </DataTemplate>
            </GridViewColumn.CellTemplate>
          </GridViewColumn>
```

```
          <GridViewColumn Header="Nombre"
             DisplayMemberBinding="{Binding XPath=nombre}" />
          <GridViewColumn Header="Dirección"
             DisplayMemberBinding="{Binding XPath=direccion}" />
          <GridViewColumn Header="Estudios"
             DisplayMemberBinding="{Binding XPath=estudios}" />
          <GridViewColumn Header="Beca">
            <GridViewColumn.CellTemplate>
              <DataTemplate>
                <CheckBox IsChecked="{Binding XPath=beca}" />
              </DataTemplate>
            </GridViewColumn.CellTemplate>
          </GridViewColumn>
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
</Grid>
```

Este código daría lugar a un resultado análogo al de la figura siguiente:



ListView no se ha diseñado para acceder a sus elementos a través de su propiedad **Items** como vimos al hablar del control **ListBox**, sino que se ha diseñado para que esté enlazado a una fuente de datos. Para mostrar un **ListView** trabajando, hemos utilizado un proveedor de datos XML, *xmlAlumnos*, cuya fuente de datos es el fichero *alumnos.xml* de la carpeta *datos* del proyecto (el enlace de datos lo iniciamos en el capítulo *Introducción a WPF* y lo continuaremos, detalladamente, en los próximos capítulos, así como el estudio de los proveedores de datos):

```
<Window.Resources>
  <XmlDataProvider x:Key="xmlAlumnos"
                   Source="datos/alumnos.xml" XPath="alumnos"/>
</Window.Resources>
```

Observe que el contexto de datos del **Grid** (código XAML expuesto anteriormente, cuando diseñamos el **ListView**) hace referencia al origen de datos definido por este recurso.

La estructura de cada nodo *alumno* de la lista almacenada en el fichero *alumnos.xml* es así:

```
<?xml version="1.0" encoding="utf-8" ?>
<alumnos>
  <alumno>
    <foto>Imagenes/foto.jpg</foto>
    <nombre>Alumno 01</nombre>
    <direccion>Dirección 01</direccion>
    <estudios>Estudios 01</estudios>
    <beca>True</beca>
  </alumno>
  ...
</alumnos>
```

Ahora, haciendo uso de lo aprendido en el capítulo *Introducción a WPF*, observe en el diseño de **ListView** que las columnas segunda, tercera y cuarta utilizan la propiedad **DisplayMemberBinding** para indicar qué dato debe mostrar cada una. En cambio, las columnas primera y quinta utilizan un mecanismo alternativo: la propiedad **CellTemplate**; este camino permite definir una plantilla, objeto **DataTemplate**, para definir cómo se construirá esa celda en cada fila del control; en este caso, se utiliza para la primera columna un control **Image** con su propiedad **Source** enlazada al dato *Foto* y para la segunda, un **CheckBox** con su propiedad **IsChecked** enlazada al dato *Beca*. El contenido de **DataTemplate** se convierte en la estructura visual de los objetos de datos.

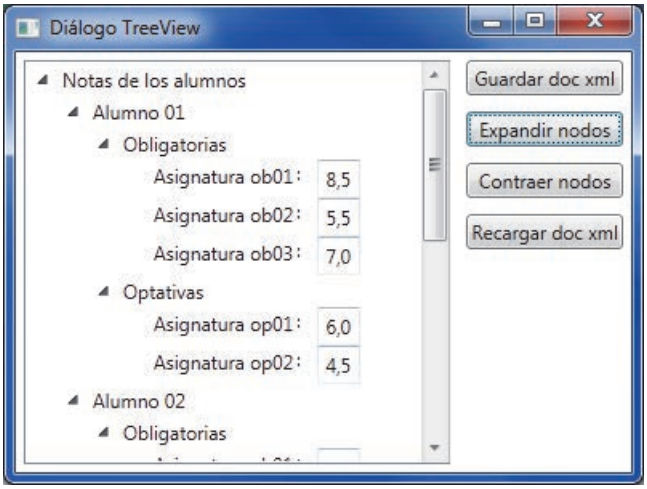
El tamaño de las columnas se ajusta automáticamente en función del tamaño de la cabecera o del contenido de las celdas.

TreeView

TreeView, derivado de **ItemsControl**, es un contenedor jerárquico (contenido estructurado en árbol), lo que significa que puede utilizar varias capas de visualización para mostrar los datos, permitiendo que sus elementos (también llamados nodos) se puedan expandir y contraer. Por ejemplo, podría crear un control **TreeView** que mostrara grupos de categorías en su primer nivel (nodos primarios) y los elementos relacionados en cada nodo categoría en su segundo nivel (nodos secundarios). Esto significa que cada nodo del árbol, objeto de la clase **TreeViewItem**, puede tener anidados otros nodos del mismo tipo formando así una jerarquía de elementos. El diseño de este control jerárquico puede resultar sencillo si se utiliza la clase **HierarchicalDataTemplate**, que representa un **DataTempla-**

te que admite controles **HeaderedItemsControl**, como **TreeViewItem** o **MenuItem**, y puede obtener los datos enlazando con un origen de datos como **XmlDataProvider** u **ObservableCollection(Of *T*)**.

Como ejemplo vamos a diseñar la ventana que muestra la figura siguiente. Dicha ventana visualiza un contenedor **TreeView**, que mostrará los datos jerárquicos (alumnos y las notas de las asignaturas en las que está matriculado cada alumno) almacenados en un fichero XML, y cuatro botones que realizarán las funciones siguientes: *Guardar doc xml* guardará el documento dinámico mostrado por el **TreeView** en el fichero XML desde el cual se cargó para salvar los posibles cambios en las notas, *Expandir nodos* expandirá todos los nodos del **TreeView**, *Contraer nodos* contraerá todos los nodos del **TreeView** y *Recargar doc xml* volverá a cargar el documento XML cuando sea necesario.



Para mostrar el control **TreeView** trabajando, hemos utilizado un proveedor de datos XML, *xmlAlumnos*, cuya fuente de datos es el fichero *alumnos.xml* guardado en la carpeta *datos* del proyecto.

```
<Window.Resources>
  <!-- Proveedor de datos XML -->
  <XmlDataProvider x:Key="xmlAlumnos" Source="datos/alumnos.xml"/>
</Window.Resources>
```

La estructura del fichero *alumnos.xml* es así:

```
<?xml version="1.0" encoding="utf-8"?>
<alumnos>
  <alumno>
    <foto>Imagenes/foto.jpg</foto>
    <nombre>Alumno 01</nombre>
```

```
    <direccion>Dirección 01</direccion>
    <estudios>Estudios 01</estudios>
    <beca>True</beca>
    <asignaturas tipo="Obligatorias">
      <asignatura>
        <nombre>Asignatura ob01</nombre>
        <nota>8,5</nota>
      </asignatura>
      <asignatura>
        <nombre>Asignatura ob02</nombre>
        <nota>5,5</nota>
      </asignatura>
      <asignatura>
        <nombre>Asignatura ob03</nombre>
        <nota>7,0</nota>
      </asignatura>
    </asignaturas>
    <asignaturas tipo="Optativas">
      <asignatura>
        <nombre>Asignatura op01</nombre>
        <nota>6,0</nota>
      </asignatura>
      <asignatura>
        <nombre>Asignatura op02</nombre>
        <nota>4,5</nota>
      </asignatura>
    </asignaturas>
  </alumno>
  <alumno>
    ...
  </alumno>
  ...
</alumnos>
```

Suponiendo que también hemos añadido una nueva ventana *Diálogo TreeView* al proyecto sobre el que estamos trabajando, el diseño correspondiente a los controles de esta ventana, según muestra la figura anterior, podría realizarse de acuerdo al siguiente código:

```
<Grid DataContext="{Binding Source={StaticResource xmlAlumnos},
                            XPath=alumnos}">
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>

  <TreeView  Name="trvAlumnos" Margin="5"
             ItemsSource="{Binding}"
             ItemTemplate="{StaticResource kAlumnos}">
  </TreeView>
```

```
  <StackPanel Grid.Column="1">
    <Button Name="btGuardarDocXml" Content="Guardar doc xml"
            VerticalAlignment="Top" Margin="5"
            Click="btGuardarDocXml_Click" />
    <Button Name="btExpandirNodos" Content="Expandir nodos"
            VerticalAlignment="Top" Margin="5"
            Click="btExpandirNodos_Click" />
    <Button Name="btContraerNodos" Content="Contraer nodos"
            VerticalAlignment="Top" Margin="5"
            Click="btContraerNodos_Click" />
    <Button Name="btRecargarDocXml" Content="Recargar doc xml"
            VerticalAlignment="Top" Margin="5"
            Click="btRecargarDocXml_Click" />
  </StackPanel>
</Grid>
```

Observe el control **TreeView**. Su propiedad **ItemsSource** especifica el origen de datos e **ItemTemplate** define el contenido de **TreeViewItem**. El origen de datos es el documento XML proporcionado por el proveedor de datos XML *xmlAlumnos* asociado al contexto de datos del **Grid** (*ItemsSource="{Binding}"* especifica que se seleccionará el nodo actual del contexto actual, la raíz en este caso) y el contenido del **TreeView** está definido por el recurso *kAlumnos*.

```
<TreeView>
  <TreeViewItem>
    Elementos TreeViewItem
  </TreeViewItem>
</TreeView>
```

Como dijimos anteriormente, para definir el diseño de un control **TreeViewItem**, puede utilizar también objetos **HierarchicalDataTemplate**. En nuestro caso, según muestra el esquema siguiente, tendremos cuatro tipos de nodos atendiendo a su contenido: *asignatura*, *asignaturas*, *alumno* y *alumnos*. El nodo *alumnos* contiene nodos *alumno*, el nodo *alumno* contiene nodos *asignaturas* y el nodo *asignaturas* contiene nodos *asignatura*:

```
<alumnos>
  Contenido de alumnos
  <alumno>
    Contenido de alumno
    <asignaturas>
      Contenido de asignaturas
      <asignatura>
        Contenido de asignatura
      </asignatura>
      ...
    </asignaturas>
    ...
```

```
  </alumno>
  ...
<alumnos>
```

Definamos entonces, en los recursos de la ventana, las plantillas correspondientes a los nodos descritos (eche una ojeada al apartado *Plantillas de datos* en el capítulo *Enlaces de datos en WPF*). El nodo *asignatura* es un nodo terminal, por lo tanto su plantilla quedará definida por un objeto **DataTemplate**. En cambio, el resto de los nodos son nodos jerárquicos (contienen nodos secundarios) por lo que su plantilla quedará definida por un objeto **HierarchicalDataTemplate**:

```
<Window.Resources>
  <!-- Proveedor de datos XML -->
  <XmlDataProvider x:Key="xmlAlumnos" Source="datos/alumnos.xml"/>

  <!-- PLANTILLA ASIGNATURA -->
  <DataTemplate x:Key="kAsignatura">
    <WrapPanel>
      <TextBlock Text="{Binding XPath=nombre}" Padding="2" />
      <TextBlock Text=":   " />
      <TextBox Text="{Binding XPath=nota}" Padding="2" />
    </WrapPanel>
  </DataTemplate>

  <!-- PLANTILLA ASIGNATURAS -->
  <HierarchicalDataTemplate x:Key="kAsignaturas"
                    ItemsSource="{Binding XPath=asignatura}"
                    ItemTemplate="{StaticResource kAsignatura}">
    <TextBlock Text="{Binding XPath=@tipo}" Padding="2" />
  </HierarchicalDataTemplate>

  <!-- PLANTILLA ALUMNO -->
  <HierarchicalDataTemplate x:Key="kAlumno"
                    ItemsSource="{Binding XPath=asignaturas}"
                    ItemTemplate="{StaticResource kAsignaturas}">
    <TextBlock Text="{Binding XPath=nombre}" Padding="2" />
  </HierarchicalDataTemplate>

  <!-- PLANTILLA ALUMNOS -->
  <HierarchicalDataTemplate x:Key="kAlumnos"
                    ItemsSource="{Binding XPath=alumno}"
                    ItemTemplate="{StaticResource kAlumno}">
    <TextBlock Text="Notas de los alumnos" Padding="2" />
  </HierarchicalDataTemplate>
</Window.Resources>
```

Si un elemento no es un control **TreeViewItem**, se incluye automáticamente en un control **TreeViewItem** cuando se muestra el control **TreeView**.

Observe las plantillas de los nodos jerárquicos; al igual que **TreeView**, definen su propiedad **ItemsSource** para especificar su origen de datos, **ItemTemplate** para definir el contenido de los **TreeViewItem** secundarios, así como los controles que mostrarán el contenido del propio nodo; estos controles hacen referencia a los datos que tienen que mostrar mediante la propiedad **XPath** del enlace de tipo **Binding** empleado. Observe la sintaxis de **XPath**, cuando el identificador del dato asociado es un atributo de un elemento XML, por ejemplo *XPath=@tipo*, va precedido por @, no sucediendo esto cuando el identificador del dato se corresponde con un elemento XML, por ejemplo *XPath=alumno*.

Veamos a continuación cómo se implementa la función de cada uno de los botones que hemos añadido a la interfaz gráfica.

Guardar el documento XML

En el ejemplo que estamos desarrollando sobre el diseño y utilización del control **TreeView**, las notas son mostradas a través de un control **TextBox** con la intención de poder modificarlas. Para que los cambios persistan, el usuario guardará el documento dinámico haciendo clic en el botón *Guardar doc xml*. Por lo tanto, escriba el controlador de este botón según se indica a continuación:

```
Private Sub btGuardarDocXml_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Recuperar el proveedor XML de los recursos de MainWindow
  Dim dataProvider As XmlDataProvider =
        TryCast(Me.FindResource("xmlAlumnos"), XmlDataProvider)
  Dim nombreFicheroXml As String = "../../Datos/alumnos.xml"
  dataProvider.Document.Save(nombreFicheroXml)
End Sub
```

Observe que para guardar el documento XML en el fichero especificado, simplemente invocamos al método **Save** del objeto **XmlDocument** que se utiliza como origen de enlace, referenciado por la propiedad **Document** del proveedor de datos.

Recargar el documento XML

Si en algún momento el usuario necesita recargar el documento visualizado por el control **TreeView** (incluso, utilizando el diálogo *Abrir*, podríamos cargar otro documento XML con la misma estructura de elementos), hará clic en el botón *Recargar doc xml*. Por lo tanto, escriba el controlador de este botón según se indica a continuación (cuando estudie el apartado *Vistas de colección* en el capítulo *Enlaces de datos en WPF* comprenderá por qué actuamos así):

```
Private Sub btRecargarDocXml_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim dataProvider As XmlDataProvider =
        TryCast(Me.FindResource("xmlAlumnos"), XmlDataProvider)
  Dim docXml As New XmlDocument()
  Dim nombreFicheroXml As String = "../../Datos/alumnos.xml"
  docXml.Load(nombreFicheroXml)
  ' Asignar el documento modificado, usando la raíz de docXml
  dataProvider.Document = docXml
End Sub
```

Observe que para recargar el documento XML guardado en el fichero especificado, simplemente creamos un nuevo documento **XmlDocument**, invocamos al método **Load** del mismo para cargar el documento XML y asignamos este objeto a la propiedad **Document** del proveedor de datos.

Expandir o contraer los nodos

Para expandir todos los nodos de un control **TreeView** basta con asignar a la propiedad **IsExpanded** de cada uno de ellos (recuerde que son elementos del tipo **TreeViewItem**) el valor **True** o el valor **False** si lo que deseamos es contraer todos los nodos. Esto lo podemos hacer definiendo para el control **TreeView** dos estilos: *Expandir* y *Contraer*. Ambos estilos, por facilidad, los definiremos como recursos de la ventana así:

```
<Window.Resources>
  ...

  <!-- Estilos para facilitar la expansión y contracción de los nodos -->
  <Style x:Key="Expandir" TargetType="{x:Type TreeView}">
    <Style.Resources>
      <Style TargetType="TreeViewItem">
        <Setter Property="IsExpanded" Value="True" />
      </Style>
    </Style.Resources>
  </Style>
  <Style x:Key="Contraer" TargetType="{x:Type TreeView}">
    <Style.Resources>
      <Style TargetType="TreeViewItem">
        <Setter Property="IsExpanded" Value="False" />
      </Style>
    </Style.Resources>
  </Style>
</Window.Resources>
```

La propiedad **Resources** de **Style** permite acceder a la colección de recursos que se pueden utilizar en el ámbito del estilo que estamos definiendo.

Cada uno de estos estilos será aplicado al control **TreeView** cuando el usuario pulse el botón correspondiente, *Expandir nodos* o *Contraer nodos*. Por lo tanto, escriba los controladores de estos botones según se indica a continuación:

```
Private Sub btExpandirNodos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  trvAlumnos.Style = DirectCast(Me.FindResource("Expandir"), Style)
End Sub

Private Sub btContraerNodos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  trvAlumnos.Style = DirectCast(Me.FindResource("Contraer"), Style)
End Sub
```

DataGrid

El control **ListView**, expuesto anteriormente, permite mostrar contenido distribuido en columnas así como cambiar rápidamente entre diferentes vistas o modos de visualización sin necesidad de volver a enlazar los datos y reconstruir la lista, pero no permite, por ejemplo, la edición de los datos presentados. En cambio, el control **DataGrid** de WPF es más completo para mostrar datos. La información que muestra en una rejilla de columnas, filas y celdas la obtiene, a través de su propiedad **ItemsSource**, de una colección de objetos que implementa la interfaz **IEnumerable**. Cada fila se corresponde con un objeto particular, y cada columna, con una propiedad en ese objeto. Además del acceso a nivel de celda, tiene otras características de formato (por ejemplo, la capacidad de congelación de las columnas y filas con estilo individual) que **ListView** no tiene y, además, soporta la edición de los datos presentados. También, igual que ocurría con el control **ListBox**, cada vez que la selección cambia se genera el evento **SelectionChanged**.

Una de las características importantes del **DataGrid** es su soporte para la edición. Una celda cambia a modo de edición cuando el usuario hace doble clic sobre ella (el primer clic para seleccionarla y el segundo para pasar a modo de edición). La edición será efectiva cuando se cambie el cursor a otra celda. También se puede abandonar la edición de la celda actual pulsando la tecla *Esc*. Sin embargo, podemos restringir esta capacidad de varias maneras: por medio de la propiedad **DataGrid.IsReadOnly**, cuando esta propiedad vale **True** el usuario no puede editar nada, o por medio de la propiedad **DataGridColumn.IsReadOnly**, cuando esta propiedad vale **True** el usuario no puede editar la columna correspondiente.

Columnas del DataGrid

Los tipos de columnas que integra **DataGrid** incluyen: columnas de cajas de texto (**DataGridTextColumn**), columnas de casillas (**DataGridCheckBoxColumn**),

columnas de plantilla (**DataGridTemplateColumn**) para hospedar contenido personalizado, columnas de lista desplegable (**DataGridComboBoxColumn**) y columnas de hiperenlace (**DataGridHyperlinkColumn**). Además, el tipo de fila que integra incluye también una sección de detalles desplegable que se puede utilizar para mostrar contenido adicional debajo de los valores de celda. Por ejemplo, análogamente a como procedimos en el ejemplo acerca de **ListView**, el siguiente código XAML define un control **DataGrid** que mostrará los datos del origen de datos *xmlAlumnos* a los que hace referencia su propiedad **ItemsSource**:

```
<Grid DataContext="{Binding Source={StaticResource xmlAlumnos},
                                    XPath=alumnos}">
  <DataGrid Name="dgAlumnos" Margin="5"
            AutoGenerateColumns="False"
            ItemsSource="{Binding XPath=alumno}">
    <DataGrid.Columns>
      <DataGridTemplateColumn>
        <DataGridTemplateColumn.CellTemplate>
          <DataTemplate>
            <Image Source="{Binding XPath=foto}" Height="32" Width="32"/>
          </DataTemplate>
        </DataGridTemplateColumn.CellTemplate>
      </DataGridTemplateColumn>
      <DataGridTextColumn Header="Nombre"
                          Binding="{Binding XPath=nombre}"/>
      <DataGridTextColumn Header="Dirección"
                          Binding="{Binding XPath=direccion}"/>
      <DataGridTextColumn Header="Estudios"
                          Binding="{Binding XPath=estudios}"/>
      <DataGridCheckBoxColumn Header="Beca"
                          Binding="{Binding XPath=beca}"/>
    </DataGrid.Columns>
  </DataGrid>
</Grid>
```

Observe que la vista del **DataGrid** define cinco columnas: la primera es para mostrar una foto y las siguientes van encabezadas por los títulos *Nombre*, *Dirección*, *Estudios* y *Beca*. Para definir estas columnas hay que añadir a la colección **Columns** de **DataGrid** por cada una de ellas un objeto *DataGrid???Column*, del tipo adecuado al contenido a mostrar.

Los datos mostrados son proporcionados por el contexto de datos del **Grid**, que, como se puede ver, está vinculado con el proveedor de datos definido en los recursos de la ventana así:

```
<Window.Resources>
  <XmlDataProvider x:Key="xmlAlumnos" Source="datos/alumnos.xml"/>
</Window.Resources>
```

El ejemplo que acabamos de exponer daría lugar a un resultado análogo al de la figura siguiente:



Evidentemente, **DataGrid** se ha diseñado para que esté enlazado a una fuente de datos. En los capítulos siguientes veremos con más detalle el enlace a datos, que ya fue iniciado en el capítulo *Introducción a WPF*, pero para mostrar un **DataGrid** trabajando, hemos utilizado el mismo proveedor de datos XML que utilizamos al hablar del control **ListView**. Según lo aprendido en el capítulo *Introducción a WPF*, observe que todas las columnas del **DataGrid**, excepto la primera, utilizan la propiedad **Binding** para indicar qué dato debe mostrar cada una. En cambio, la primera columna utiliza un mecanismo alternativo: la propiedad **CellTemplate**; este camino permite definir una plantilla, objeto **DataTemplate**, para definir cómo se construirá esa celda en cada fila del control; en este caso, se utiliza un control **Image** con su propiedad **Source** enlazada al dato *Foto*.

El tamaño de las columnas se ajusta automáticamente en función del tamaño de la cabecera o del contenido de las celdas, también se puede ajustar manualmente, las filas se pueden ordenar por cualquier columna haciendo clic en su cabecera y las celdas se pueden editar.

Obsérvese también que la propiedad **AutoGenerateColumns** del **DataGrid** vale **False** para que las columnas no se generen automáticamente a partir del contenido referenciado por su propiedad **ItemsSource**. Cuando las columnas se generan automáticamente, todas son del tipo caja de texto, excepto cuando el valor enlazado es de tipo **Boolean**, que serían de tipo casilla de verificación. Cuando el contenido enlazado no es editable, como sucede con la foto por ejemplo, la columna se genera de sólo lectura. Entonces, para crear una columna personalizada, columna del tipo **DataGridTemplateColumn**, hay que establecer **AutoGenerateColumns** en **False**. En este caso, las propiedades **CellEditingTemplate** y **Cell-**

Template le permiten especificar plantillas de contenido para los modos de edición y presentación, respectivamente.

Para agregar, insertar, quitar y cambiar las columnas del **DataGrid** mediante programación hay que actuar sobre la colección **DataGrid.Columns**.

Inmovilizar columnas

La propiedad **FrozenColumnCount** del **DataGrid** permite fijar el número de columnas del **DataGrid** que se desea inmovilizar. Por ejemplo:

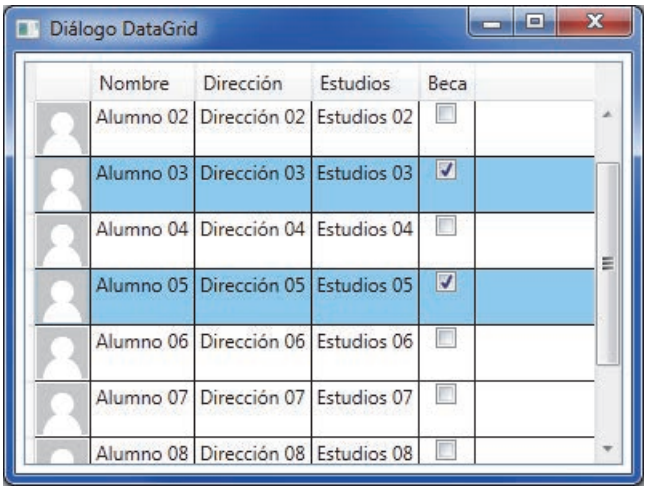
```
<DataGrid Name="dgAlumnos" FrozenColumnCount="2" ...>
```

Las columnas inmovilizadas se muestran siempre y se corresponden con las columnas situadas más a la izquierda en el orden de presentación. Por ejemplo, si establece **FrozenColumnCount** en 2, como en el ejemplo anterior, se inmovilizarán las dos columnas de la izquierda. La propiedad **IsFrozen** de **DataGridColumn** nos da información de si una columna está o no inmovilizada.

Filas del DataGrid

Configurando las propiedades de las columnas del **DataGrid**, podemos controlar el formato de las mismas pero, en muchos casos, es más útil señalar filas que contienen datos específicos. Por ejemplo, en el diseño anterior, podríamos querer resaltar las filas de los alumnos que tienen beca. Esta forma de proceder requiere aplicar formato a las filas, para lo cual hay que controlar el evento **DataGrid.LoadingRow**. Este evento da acceso al objeto de datos de la fila actual, lo que permitirá escribir código basado en estos datos, y también proporciona el objeto **DataGridRow** de la fila, lo que permitirá dar formato a la misma, por ejemplo, para aplicarle un color diferente. Pero, ¿cuándo se genera este evento? El evento **LoadingRow** se genera una vez para cada fila cuando ésta aparece en pantalla. La ventaja de este enfoque es que no se actúa sobre todas las filas de la rejilla, sino sólo sobre las visibles.

Otro detalle es que el **DataGrid** reutiliza los mismos objetos **DataGridRow** para mostrar las nuevas filas mientras éstas se desplazan, lo que reduce el consumo de memoria, de ahí el nombre del evento. Por otra parte, esta manera de trabajar puede conducir a que el control pueda cargar datos en un **DataGridRow** ya con formato, y que estos datos no se correspondan con ese formato. Esto quiere decir que cada vez que un **DataGridRow** recargue datos, deberemos restaurarlo a su estado inicial. Según lo expuesto, ¿cómo haríamos para que el diálogo *DataGrid* mostrado anteriormente muestre los alumnos con beca en otro color según muestra la figura siguiente?



La respuesta a la pregunta anterior pasa por interceptar el evento **LoadingRow** del **DataGrid** y verificar por cada fila que se muestra, objeto **DataGridRow**, si el dato *beca* es **True** o **False**; si es **True**, aplicamos el nuevo formato a la fila (el nuevo color) y si es **False**, restauramos el color predeterminado para asegurarnos de que las filas recargadas con formato son restauradas. Los colores utilizados se han definido como atributos de la clase *DlgDataGrid* para favorecer la velocidad de ejecución en los cambios; de otra forma, los objetos **SolidColorBrush** tendrían que definirse por cada fila que cambie.

¿Cómo accedemos al **DataGridRow** correspondiente a la fila para la que se produce el evento? Pues a través de la propiedad **Row** del objeto **DataGridRowEventArgs** de información del evento. Y, ¿cómo accedemos a su contenido? Como **DataGridRow** participa de un enlace de datos, será el contexto de datos de este objeto, propiedad **DataContext**, el que nos proporcione esta información, que, en nuestro ejemplo, se corresponde con un objeto **XmlNode** cuyas propiedades nos permitirán acceder al contenido deseado: elemento *beca*.

```
Imports System.Xml

Public Class DlgDataGrid
  Private azulClaro As New SolidColorBrush(Colors.LightSkyBlue)
  Private colorPredeterminado As New SolidColorBrush(Colors.White)

  Private Sub dgAlumnos_LoadingRow(sender As System.Object,
  e As System.Windows.Controls.DataGridRowEventArgs)
    ' Comprobar los datos de esta fila
    Dim filaActual As XmlNode = TryCast(e.Row.DataContext, XmlNode)
    Dim sbeca As String = filaActual("beca").InnerText
    ' Aplicar formato a la fila si...
    If sbeca.CompareTo("True") = 0 Then
```

```
      e.Row.Background = azulClaro
    Else
      ' Restaurar el color predeterminado para asegurar que
      ' las filas recargadas con formato son restauradas
      e.Row.Background = colorPredeterminado
    End If
  End Sub
End Class
```

Detalles de las filas

Cada fila de un **DataGrid** se puede expandir para mostrar una sección de detalles. La sección de detalles, que especifica el aspecto y los datos que va a mostrar, la define la plantilla **DataTemplate** que se implementa en la sección de detalles de la fila definida por la propiedad **RowDetailsTemplate**.

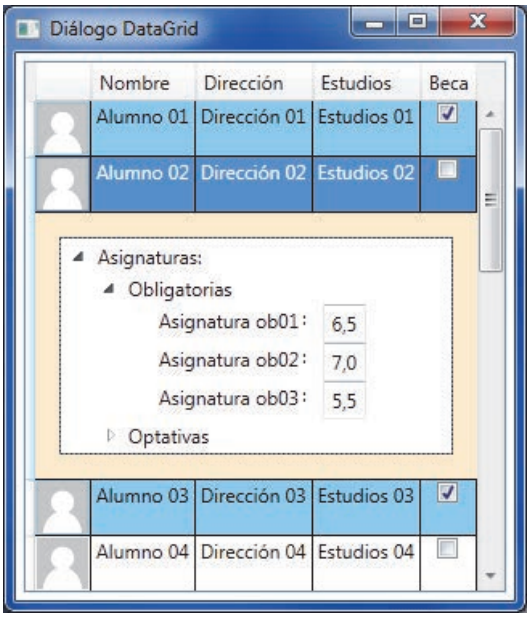
```
<DataGrid ... >
  <DataGrid.RowDetailsTemplate>
    <DataTemplate>
      detalles de la fila
    </DataTemplate>
  </DataGrid.RowDetailsTemplate>
</DataGrid>
```

Se puede implementar la sección de detalles usando XAML alineado, como muestra el código anterior, o usando un recurso, como muestra el código siguiente:

```
<DataGrid Name="dgAlumnos"
          RowDetailsTemplate="{StaticResource DetalleAsignaturas}"
          ...
</DataGrid>
```

El valor asignado a la propiedad **RowDetailsVisibilityMode** indicará el modo en el que se mostrará la sección de detalles: sólo para la fila seleccionada (*VisibleWhenSelected*; es el valor predeterminado), para todas las filas (*Visible*) o no se muestra (*Collapsed*), y utilizando la propiedad **AreRowDetailsFrozen** se puede también inmovilizar la sección de detalles para que no se desplace horizontalmente cuando se desplaza el **DataGrid**.

Como ejemplo, veamos cómo implementamos la sección de detalles mostrada por la figura siguiente:



En esta figura observamos que la sección de detalles se corresponde con un **TreeView** que muestra las asignaturas, con sus notas, del alumno seleccionado. El diseño de este objeto ya fue explicado en el apartado anterior, por lo tanto sólo queda ver qué hacer para que pase a ser la sección de detalles de las filas del **DataGrid**. Esto es lo que muestra el código XAML siguiente, en el que se observa que el diseño del **TreeView** está definido por el recurso *kAlumno*.

```
<DataGrid Name="dgAlumnos" ...>
  <DataGrid.Columns>
    ...
  </DataGrid.Columns>
  <!-- Sección de detalles de la fila -->
  <DataGrid.RowDetailsTemplate>
    <DataTemplate>
      <Border BorderThickness="0" Background="BlanchedAlmond" Padding="10">
        <TreeView  Name="trvAlumnos" Margin="5"
            ItemsSource="{Binding XPath=.}"
            ItemTemplate="{StaticResource kAlumno}">
        </TreeView>
      </Border>
    </DataTemplate>
  </DataGrid.RowDetailsTemplate>
</DataGrid>
```

CAJAS DE DIÁLOGO ESTÁNDAR

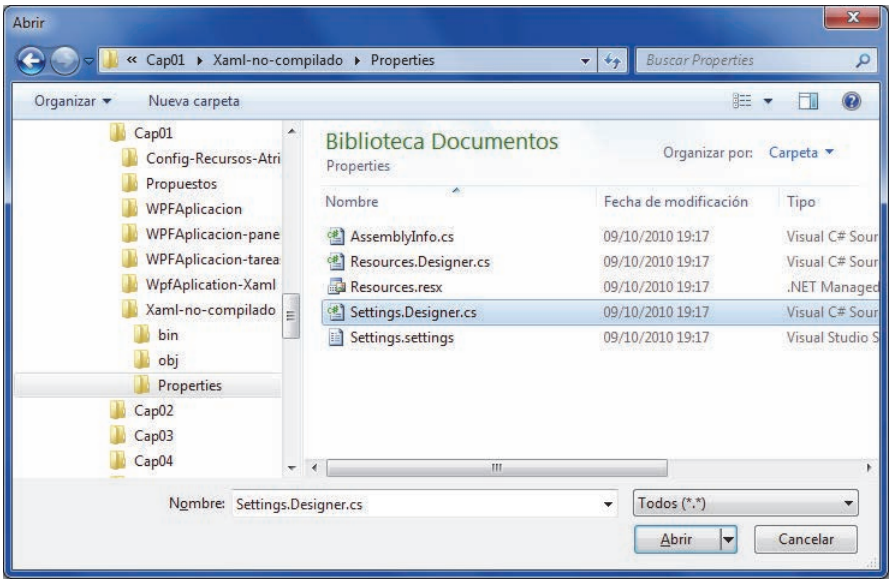
El sistema operativo Windows incluye muchas cajas de diálogo construidas, accesibles a través de la API de Windows, pero WPF proporciona envoltorios sólo para unas pocas de éstas. ¿Por qué? Pues porque una de las metas de WPF es desacoplar la funcionalidad ofrecida de la API de Windows para que sea útil en otros entornos (como un navegador) o sea portable a otras plataformas. Además, Windows 7 tiene preferencia por los paneles basados en tareas y la navegación. Por eso, WPF sólo incluye tres clases definidas en el espacio de nombres **System.Windows.Controls**, que permiten visualizar las tres cajas de diálogo más comúnmente empleadas en el diseño de aplicaciones: la caja de diálogo *Abrir* (**OpenFileDialog**), la caja de diálogo *Guardar* (**SaveFileDialog**) y la caja de diálogo *Imprimir* (**PrintDialog**).

Las clases **OpenFileDialog** y **SaveFileDialog** se derivan de la clase abstracta **FileDialog** del espacio de nombres **Microsoft.Win32** y la clase **PrintDialog** se deriva de **Object** y pertenece al espacio de nombres **System.Windows.Controls**.

Para ver cómo se utilizan estas cajas de diálogo, añada al menú *Archivo* de la aplicación *CajasDeDialogo* dos nuevas órdenes, *Abrir* y *Guardar*; la primera debe mostrar la caja de diálogo estándar *Abrir* y la segunda la de *Guardar*. Como resultado mostraremos, en los dos casos, una caja de diálogo predefinida con la elección realizada.

Cajas de diálogo Abrir y Guardar

La caja de diálogo *Abrir* permite al usuario seleccionar una carpeta de su sistema de ficheros, una extensión de fichero y un fichero. Una vez realizada la selección, la propiedad **FileName** de la caja de diálogo contiene el nombre completo del fichero elegido.



Para visualizar la caja de diálogo *Abrir*:

1. Creamos un objeto de la clase **OpenFileDialog**.
2. Modificamos los valores por omisión de sus propiedades, si es necesario.
3. Mostramos el diálogo invocando a **ShowDialog**.

El valor devuelto por **ShowDialog** indica si el usuario aceptó la operación (*True*) o la canceló (*False*).

De acuerdo con lo expuesto, el método asociado con la orden *Abrir* del menú *Archivo* puede ser el siguiente:

```
Imports Microsoft.Win32 ' para OpenFileDialog
Imports System.IO
...

Private Sub ArchivoAbrir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim DlgAbrir As New OpenFileDialog()

  DlgAbrir.Filter = "ficheros txt (*.txt)|*.txt|Todos (*.*)|*.*"
  DlgAbrir.FilterIndex = 2

  ' Mostrar el diálogo Abrir
  If DlgAbrir.ShowDialog() = True Then
    Dim fs As Stream = DlgAbrir.OpenFile()
    ' Código para trabajar con el fichero
    ' ...
```

```
    MessageBox.Show(DlgAbrir.FileName)
  End If
End Sub
```

El método **OpenFile** de la clase **OpenFileDialog** abre el fichero seleccionado por el usuario con permiso de sólo lectura y **Close** lo cierra. La propiedad **FileName** especifica el nombre de ese fichero incluyendo la ruta de acceso.

La propiedad **Filter** especifica la cadena actual del filtro de nombres de fichero que da lugar a las opciones que aparecen en la lista "Tipo" del diálogo, **FilterIndex** especifica el índice del filtro actualmente seleccionado en el diálogo e **InitialDirectory** especifica el directorio inicial que muestra la caja de diálogo.

La caja de diálogo *Guardar* es idéntica a la caja de diálogo *Abrir* (sólo cambia *Abrir* por *Guardar*). El siguiente método asociado con la orden *Guardar* del menú *Archivo* muestra esta caja de diálogo invocando a **ShowDialog**:

```
Imports Microsoft.Win32 ' para OpenFileDialog
Imports System.IO
...

Private Sub ArchivoGuardar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim fs As Stream
  Dim DlgGuardar As New SaveFileDialog()

  DlgGuardar.Filter = "ficheros txt (*.txt)|*.txt|Todos (*.*)|*.*"
  DlgGuardar.FilterIndex = 2

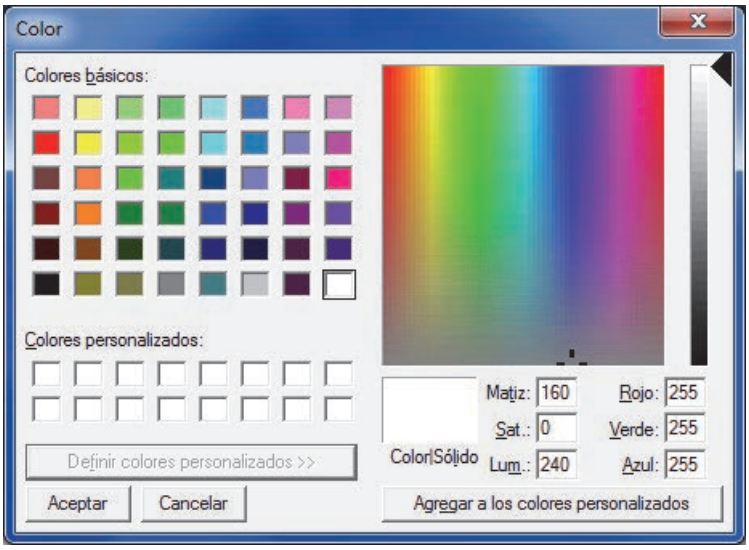
  If DlgGuardar.ShowDialog() = True Then
    ' Abrir el fichero para leer y escribir
    fs = DlgGuardar.OpenFile()
    If fs IsNot Nothing Then
      ' Código para trabajar con el fichero
      ' ...
      fs.Close()
    End If
  End If
End Sub
```

El método **OpenFile** de la clase **SaveFileDialog** abre el fichero seleccionado por el usuario con permiso de lectura y escritura. En cambio, para la clase **OpenFileDialog** lo abre sólo para lectura.

Cajas de diálogo Windows Forms estándar

Para acceder a otras cajas de diálogo estándar en *Windows Forms* pero no implementadas en WPF, como la caja de diálogo *Color*, tendremos que añadir a nuestro proyecto una referencia a la biblioteca **System.Windows.Forms** y, dependiendo del diálogo a utilizar, seguramente habrá que añadir también la biblioteca **System.Drawing.dll**. Una vez añadidas estas referencias podrá utilizar los diálogos estándar pero siendo conscientes de algunos problemas. Por ejemplo, el problema con **ColorDialog** es que el tipo de su propiedad **Color** es **System.Drawing.Color** y un objeto de esta clase no tiene la misma estructura que el objeto **System.Windows.Media.Color** empleado por WPF. Por ello, aunque es posible utilizar este diálogo en una aplicación WPF, no es la mejor opción. Veamos a continuación un ejemplo.

La caja de diálogo *Color* permite al usuario seleccionar un color de una paleta o crear y seleccionar un color personalizado. La figura siguiente muestra el aspecto de este diálogo:



Para visualizar la caja de diálogo *Color*:

1. Creamos un objeto de la clase **ColorDialog**.
2. Modificamos los valores por omisión de sus propiedades, si es necesario.
3. Mostramos el diálogo invocando al método estático **ShowDialog**.

El valor de tipo **DialogResult** devuelto por **ShowDialog** indica si el usuario aceptó la operación (*OK*) o la canceló (*Cancel*).

De acuerdo con lo expuesto, añada un bloque de texto a *MainWindow* y el controlador de eventos **Click** para la orden *Color* del menú *Diálogos*. Después, complete este controlador como se indica a continuación. La intención es que el usuario pueda modificar el color del texto del bloque de texto con el color seleccionado del diálogo *Color*.

```
Private Sub DialogoColor_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim DlgColor As New System.Windows.Forms.ColorDialog()

  ' Seleccionar inicialmente el color actual del fondo
  Dim cta As SolidColorBrush = TryCast(textBlock1.Foreground,
                                       SolidColorBrush)
  DlgColor.Color = System.Drawing.Color.FromArgb(cta.Color.A,
                       cta.Color.R, cta.Color.G, cta.Color.B)
  ' Actualizar el color del fondo de TextBox1
  If DlgColor.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
    Dim wpfColor = System.Windows.Media.Color.FromArgb( _
          DlgColor.Color.A, _
          DlgColor.Color.R, DlgColor.Color.G, DlgColor.Color.B)
    Dim ctn = New SolidColorBrush(wpfColor)
    textBlock1.Foreground = ctn
  End If
End Sub
```

La propiedad **Color** del diálogo especifica el color seleccionado por el usuario. Cuando se abra el diálogo *Color*, aparecerá inicialmente seleccionado el color indicado por esta propiedad, o el negro si a esta propiedad no se le asigna un valor. Otras propiedades son: **AllowFullOpen**, que indica si el usuario puede utilizar la caja de diálogo para definir colores personalizados, o **ShowHelp**, que indica si aparecerá el botón *Ayuda* en el cuadro de diálogo *Color*.

En el código anterior se puede observar que el método **ShowDialog** muestra el diálogo *Color* que inicialmente visualizará seleccionado el color del texto actual de *textBlock1*. Una vez mostrado este diálogo el usuario elegirá otro que será almacenado en la propiedad **Color** del diálogo y aplicado a *textBlock1*como nuevo color del texto.

Caja de diálogo Imprimir

La caja de diálogo *Imprimir* permite seleccionar una impresora, configurarla y realizar un trabajo de impresión.

El trabajo de impresión se lleva a cabo por medio del método **PrintDocument**. Por ello, en sentido estricto, puede utilizar este método sin llegar a abrir nunca el cuadro de diálogo.

El método **PrintDocument** imprime un objeto **DocumentPaginator** en la impresora **PrintQueue** actualmente seleccionada. **DocumentPaginator** admite la creación de elementos de varias páginas de un documento único y **PrintQueue** administra impresoras y trabajos de impresión.

```
DlgImprimir.PrintDocument(DirectCast(richTBox.Document,
  IDocumentPaginatorSource).DocumentPaginator, nombreDoc)
```

Un poco más adelante veremos un ejemplo completo.

CONTROLES DE DOCUMENTOS WPF

WPF divide los documentos en dos categorías: *documentos fijos* y *documentos dinámicos*.

Los documentos fijos están diseñados para conseguir una presentación concreta, invariable, independiente del hardware de pantalla o de impresión utilizado. Este libro constituye un ejemplo de un documento fijo en el que el diseño de cada página permanece inalterado en todos los casos.

El control **DocumentViewer** está diseñado para mostrar contenido **FixedDocument** (documentos fijos) en modo de sólo lectura; esto es, la edición o modificación de contenido no está disponible. Este control proporciona una interfaz que ofrece salida impresa, copia en el portapapeles, aplicación de zoom o búsqueda de texto.

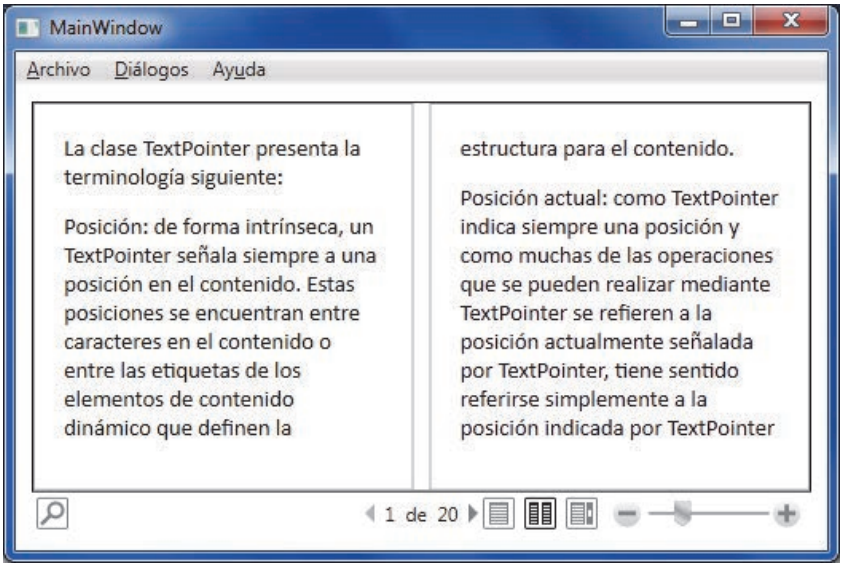
En comparación, los documentos dinámicos, **FlowDocument**, están diseñados para disponer de una presentación variable, ajustando y recolocando dinámicamente su contenido en función del tamaño del contenedor, la resolución del dispositivo y las preferencias opcionales del usuario. Una página web constituye un ejemplo sencillo de un documento dinámico.

Un objeto **FlowDocument** almacena y da formato al contenido dinámico con características de documento avanzadas, como paginación y columnas.

Los controles **FlowDocumentReader**, **FlowDocumentPageViewer** y **FlowDocumentScrollViewer** están diseñados para mostrar contenido **FlowDocument** (documento dinámico), pero no permiten la edición. Si se necesita la capacidad de edición, **RichTextBox** es una buena opción. Todos los controles mencionados hacen referencia al objeto **FlowDocument** que almacena el contenido dinámico a través de su propiedad **Document**.

FlowDocumentReader permite elegir dinámicamente distintos modos de visualización: una sola página, dos páginas a la vez o desplazamiento continuo. La

figura siguiente muestra un control de este tipo; observe debajo del documento las opciones que presenta este control al usuario. Si no se necesita la capacidad de cambiar dinámicamente entre distintos modos de visualización, **FlowDocumentPageViewer** y **FlowDocumentScrollViewer** son mejores opciones porque proporcionan visores más ligeros.



FlowDocumentPageViewer muestra el contenido en el modo de visualización de una sola página, mientras que **FlowDocumentScrollViewer** lo muestra en el modo de desplazamiento continuo. Ambos muestran siempre una barra de desplazamiento vertical y cuando es necesario muestran también la barra de desplazamiento horizontal.

Documentos dinámicos

Un documento dinámico (*flow document*) adapta su contenido al contenedor que lo visualiza. Está formado por una serie de elementos dinámicos derivados de **FrameworkContentElement** que, a su vez, se deriva de **ContentElement**, por lo tanto se trata de elementos de contenido.

Los elementos de contenido, por ejemplo **Paragraph**, se diferencian de los de no contenido (los que derivan de **UIElement**), por ejemplo **FlowDocumentReader**, en que los primeros no controlan su propia representación, por eso requieren un contenedor que lo haga.

Los elementos secundarios de nivel superior contenidos en un objeto **FlowDocument** se deben derivar de **Block** y son los siguientes: **BlockUIContai-**

ner, **List**, **Paragraph**, **Section** y **Table**. La figura siguiente muestra la jerarquía de la que forman parte estos elementos:

- System.Windows.**FrameworkContentElement**
  - System.Windows.Controls.DefinitionBase
  - System.Windows.Documents.**FixedDocument**
  - System.Windows.Documents.FixedDocumentSequence
  - System.Windows.Documents.**FlowDocument**
  - System.Windows.Documents.TableColumn
  - System.Windows.Documents.TextElement
    - System.Windows.Documents.**Block**
      - System.Windows.Documents.BlockUIContainer
      - System.Windows.Documents.List
      - System.Windows.Documents.Paragraph
      - System.Windows.Documents.Section
      - System.Windows.Documents.Table
    - System.Windows.Documents.**Inline**
      - System.Windows.Documents.AnchoredBlock
        - System.Windows.Documents.Figure
        - System.Windows.Documents.Floater
      - System.Windows.Documents.InlineUIContainer
      - System.Windows.Documents.LineBreak
      - System.Windows.Documents.Run
      - System.Windows.Documents.Span
    - System.Windows.Documents.ListItem
    - System.Windows.Documents.TableCell
    - System.Windows.Documents.TableRow
    - System.Windows.Documents.TableRowGroup

Cuando se crea un objeto **FlowDocument**, se genera automáticamente un **FlowDocumentPageViewer** primario que almacena el contenido.

Elementos Block

Los elementos derivados de **Block** (como **Paragraph**) se pueden utilizar para agrupar elementos bajo un elemento primario común o para aplicar atributos comunes a un grupo. Si lo compara con los elementos HTML, esto es similar a cómo se podría utilizar la etiqueta <DIV>. Por ejemplo, el siguiente código XAML muestra un objeto **FlowDocument**, con una serie de elementos dinámicos, que será presentado por un **FlowDocumentScrollViewer**:

```
<FlowDocumentScrollViewer Grid.Row="1" Name="fdocReader1"
                          Margin="10">
  <FlowDocument>
    <Section FontSize="12pt">
```

```
      <Paragraph>Una sección con dos párrafos.</Paragraph>
      <Paragraph>El tamaño de la fuente es 12.</Paragraph>
    </Section>

    <Paragraph>
      <Italic>Letra en cursiva.</Italic>
      Letra normal.
      <Bold>Letra en negrita.</Bold>
    </Paragraph>

    <Paragraph FontSize="11pt">
      <Span>Un párrafo con letra de tamaño 11.</Span>
    </Paragraph>
  </FlowDocument>
</FlowDocumentScrollViewer>
```

Un **BlockUIContainer** puede incluir un elemento secundario **UIElement**, sólo uno, pero éste puede incluir otros elementos secundarios propios.

```
<BlockUIContainer>
  <Button>Opciones</Button>
</BlockUIContainer>
```

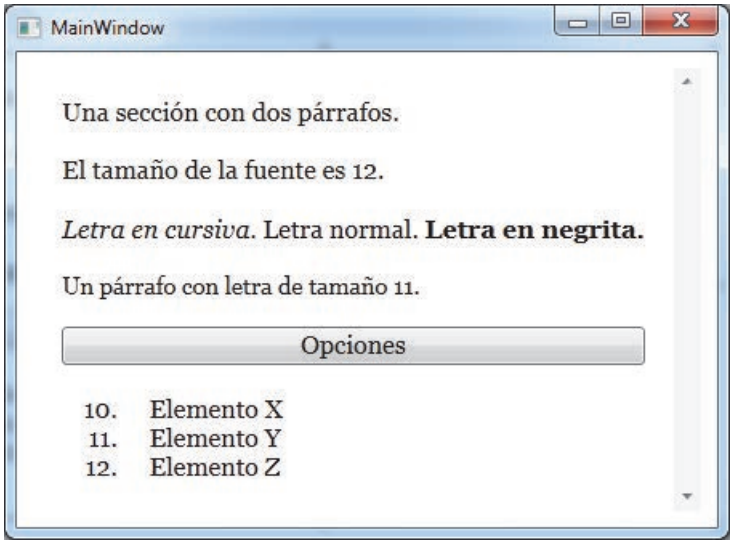
List es un elemento de contenido dinámico diseñado para presentar el contenido en una lista ordenada o sin ordenar, aplicando viñetas o numeración. Los elementos secundarios sólo pueden ser elementos **ListItem**. La propiedad **MarkerStyle** permite elegir el estilo para las viñetas o numeración y también la distancia entre sus elementos utilizando la propiedad **MarkerOffset**. Por ejemplo:

```
<List MarkerOffset="20" MarkerStyle="Decimal" StartIndex="10">
  <ListItem>
    <Paragraph>Elemento X</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Elemento Y</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Elemento Z</Paragraph>
  </ListItem>
</List>
```

Un **Paragraph**, además de texto sin formato, puede incluir otros elementos **Inline** como: **Figure**, **Floater**, **Hyperlink**, **InlineUIContainer**, **LineBreak**, **Run**, **Span**, **Bold**, **Italic** o **Underline**.

Un elemento **Section** se utiliza para agrupar otros elementos **Block**. Agrupando los elementos en una sección, podemos aplicar un formato común a toda una parte del documento.

Los ejemplos de código XAML anteriores puestos en un único documento darían lugar al siguiente documento:



Un elemento **Table** proporciona una presentación basada en una tabla organizada por filas y columnas. Este elemento puede estar contenido en los siguientes elementos: **FlowDocument**, **TableCell**, **ListBoxItem**, **ListViewItem**, **Section**, **Floater** y **Figure**.

Para crear una tabla los pasos a seguir son los siguientes:

1. Colocar en **Table** un elemento **TableRowGroup**. Este elemento agrupa las filas de la tabla y puede haber más de uno, por ejemplo para dar distintos formatos a distintos grupos de filas.

2. Para cada fila, añadir un elemento **TableRow** al elemento **TableRowGroup**.

3. Para cada columna de una fila, añadir un elemento **TableCell** al elemento **TableRow**.

4. Colocar en cada **TableCell** un elemento **Block** (típicamente un **Paragraph**) para añadir el contenido a la columna.

```
<Table BorderBrush="Black" BorderThickness="1">
  <TableRowGroup Background="LightBlue">
    <TableRow Background="SkyBlue">
      <TableCell ColumnSpan="2" TextAlignment="Center">
        <Paragraph FontSize="16pt"
                   FontWeight="Bold">Teléfonos</Paragraph>
```

```
      </TableCell>
    </TableRow>
    <TableRow FontWeight="Bold" Foreground="Blue">
      <TableCell>
        <Paragraph>Teléfono</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Nombre</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow Background="White">
      <TableCell>
        <Paragraph>999888777</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Nombre 1</Paragraph>
      </TableCell>
    </TableRow>
    <TableRow Background="LightGray">
      <TableCell>
        <Paragraph>666555444</Paragraph>
      </TableCell>
      <TableCell>
        <Paragraph>Nombre 2</Paragraph>
      </TableCell>
    </TableRow>
  </TableRowGroup>
</Table>
```

Cuando no se especifica el ancho de cada columna, WPF reparte el espacio total entre todas las columnas. No obstante, podemos modificar esta conducta análogamente a como procedemos con el elemento **Grid**. Por ejemplo:

```
<Table.Columns>
  <TableColumn Width="*"></TableColumn>
  <TableColumn Width="2*"></TableColumn>
</Table.Columns>
```

Otras cosas que se pueden hacer son utilizar las propiedades **ColumnSpan** y **RowSpan** de una celda para extenderla a más de una columna o fila.

La tabla definida por el código XAML anterior daría lugar al siguiente documento:



Elementos Inline

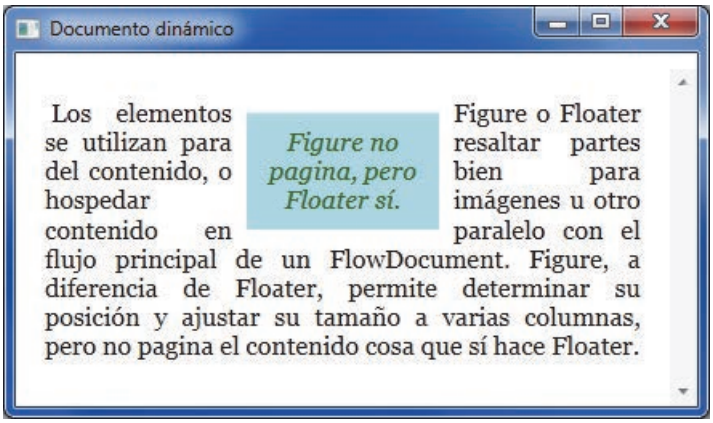
Los elementos derivados de **Inline** (como **Run**) son utilizados dentro de un elemento **Block** o dentro de otro elemento **Inline**. Por ejemplo:

```
<Paragraph FontSize="14pt">
  <Run>Un párrafo con letra de tamaño 14.</Run>
</Paragraph>
```

Los elementos **Figure** o **Floater** se utilizan para resaltar partes del contenido, o bien para hospedar imágenes u otro contenido en paralelo con el flujo principal de un **FlowDocument**. **Figure**, a diferencia de **Floater**, permite determinar su posición y ajustar su tamaño a varias columnas, pero no pagina el contenido, cosa que sí hace **Floater**. Para crear uno de estos elementos, simplemente tenemos que incluirlo en otro elemento **Block**, por ejemplo en un párrafo. Por ejemplo:

```
<Paragraph>
  <Figure Width="120" Height="45"
          Background="LightBlue" TextAlignment="Center"
          HorizontalAnchor="PageCenter" WrapDirection="Both">
    <Paragraph FontStyle="Italic" Foreground="DarkGreen" >
      Figure no pagina, pero Floater sí.
    </Paragraph>
  </Figure>
  Los elementos Figure o Floater se utilizan para resaltar partes
  del contenido, o bien para hospedar imágenes u otro contenido
  en paralelo con el flujo principal de un FlowDocument. Figure,
  a diferencia de Floater, permite determinar su posición y
  ajustar su tamaño a varias columnas, pero no pagina el contenido
  cosa que sí hace Floater.
</Paragraph>
```

Este párrafo daría lugar al texto que muestra la figura siguiente:



La posición del objeto **Figure** se fija mediante sus propiedades **VerticalAnchor** y **HorizontalAnchor**. En comparación, un objeto **Floater** puede situarse dentro de su contenedor utilizando su propiedad **HorizontalAlignment**.

Un **InlineUIContainer** puede incluir un elemento secundario **UIElement**, sólo uno, pero éste puede incluir otros elementos secundarios propios.

```
<InlineUIContainer>
  <Button>aquí</Button>
</InlineUIContainer>
```

El contenido secundario de los elementos **FlowDocument**, **Section**, **ListItem**, **TableCell**, **Floater** y **Figure** está definido por la colección **BlockCollection** y el de **Paragraph**, **Span** y **TextBlock**, por **InlineCollection**. Por lo tanto, un **BlockUIContainer** no se puede colocar dentro de un **Paragraph**, pero sí un **InlineUIContainer**.

Un elemento **LineBreak** se utiliza para iniciar una nueva línea de texto y, a diferencia de otros objetos **Inline**, no permite elementos secundarios.

Un elemento **Run** está pensado para soportar texto con o sin formato. Además, su propiedad **Text** es una propiedad de dependencia, lo que significa que se puede enlazar a un origen de datos.

```
<FlowDocument  FontFamily="Calibri">
  <Paragraph>
    <Run Typography.Variants="Superscript">
      Texto como superíndice.</Run> Texto normal.
  </Paragraph>
  <Paragraph>
    Si una fuente no soporta una forma particular (como
```

```
    superíndice) se utiliza la forma por defecto.
  </Paragraph>
</FlowDocument>
```

Un elemento **Span** agrupa otros elementos de contenido dinámico **Inline**. Normalmente utilizamos **Span** para especificar un formato para un texto determinado.

Para más detalles sobre estos elementos y de todos sus miembros (propiedades, métodos y eventos) véase la ayuda proporcionada por *Microsoft Developer Network* - MSDN.

Paragraph y Run

Por lo expuesto hasta ahora, podemos deducir que un elemento **Paragraph** representa un párrafo de texto, aunque, en la realidad, dicho elemento no contiene texto, sino una colección (**Paragraph.Inlines**) de elementos **Inline**. Esto significa, por una parte, que un elemento **Paragraph** puede contener más cosas que texto y por otra, que para contener texto necesita utilizar un elemento **Run**. Según esto, los dos ejemplos mostrados a continuación son equivalentes:

```
<Paragraph Name="p1" FontSize="14pt">
  <Run>Un párrafo con letra de tamaño 14.</Run>
</Paragraph>

<Paragraph Name="p1" FontSize="14pt">
  Un párrafo con letra de tamaño 14.
</Paragraph>
```

En el último ejemplo, el objeto **Paragraph** crea implícitamente un elemento **Run** para contener el texto especificado.

¿Cómo podríamos acceder al texto de un párrafo? Si observamos las propiedades de **Paragraph**, descubriremos que no tiene una propiedad **Text**. Entonces, para acceder al texto que representa, tendremos que hacerlo a través del objeto **Run** como se indica a continuación:

```
((Run)p1.Inlines.FirstInline).Text = "Un texto";
```

Interactuando con los elementos mediante programación

Hasta aquí hemos visto ejemplos de cómo escribir el XAML requerido para crear los elementos de un documento dinámico. No obstante, como podemos imaginar, este tipo de documentos también puede crearse mediante programación, aunque

resulta bastante tedioso por la gran cantidad de elementos que tendremos que utilizar. Por cada elemento que necesitemos, igual que en XAML, tendremos que crearlo, establecer sus propiedades y añadirlo al objeto contenedor del mismo.

Veamos a continuación un pequeño ejemplo:

```
' Párrafo a construir:
'   Crear un documento dinámico mediante programación.
' Crear la primera parte del párrafo
Dim parte01 As New Run()
parte01.Text = "Crear un "
' Crear la segunda parte del párrafo en cursiva
Dim parte02Cursiva As New Italic()
Dim parte02 As New Run()
parte02.Text = "documento dinámico "
parte02Cursiva.Inlines.Add(parte02)
' Crear la última parte del párrafo
Dim parte03 As New Run()
parte03.Text = "mediante programación."
' Añadir las tres partes al párrafo
Dim párrafo As New Paragraph()
párrafo.Inlines.Add(parte01)
párrafo.Inlines.Add(parte02Cursiva)
párrafo.Inlines.Add(parte03)
' Crear un documento y añadir este párrafo
Dim documento As New FlowDocument()
documento.Blocks.Add(párrafo)
' Mostrar el documento mediante un control FlowDocumentScrollViewer
fdocReader1.Document = documento
```

El resultado es: Crear un *documento dinámico* mediante programación.

Para entender el código anterior piense simplemente qué elemento debe estar dentro de otro elemento (observe el código de abajo a arriba). En XAML, como se muestra a continuación, esto es trivial:

```
<FlowDocumentScrollViewer Name="fdocReader1">
  <FlowDocument Name="documento">
    <Paragraph>
      <Run Name="parte01">Crear un </Run>
      <Italic Name="parte02Cursiva">
        <Run Name="parte02">documento dinámico</Run>
      </Italic>
      <Run Name="parte03">mediante programación.</Run>
    </Paragraph>
  </FlowDocument>
</FlowDocumentScrollViewer>
```

En otras ocasiones, quizás deseemos modificar partes concretas de un documento dinámico ya creado. Esto lo podemos hacer asignando un nombre a los elementos que deseamos cambiar. Ahora bien, debemos tener presente que los documentos dinámicos utilizan un contenido profundamente anidado con una estructura de flujo libre, lo que nos puede conducir a navegar a través de varios niveles para encontrar el elemento a modificar (recuerde que el contenido está siempre almacenado en un elemento **Run**, aunque no se haya declarado explícitamente). Por ejemplo, supongamos el siguiente documento con cuatro elementos modificables (en este caso se trata de elementos **Span**):

```
<FlowDocument Name="documento">
  <Paragraph FontSize="20" FontWeight="Bold">
    Novedades de WPF versión <Span Tag="Versión">4</Span>
  </Paragraph>
  <Paragraph  FontSize="15" FontWeight="Bold">
    Nuevos controles
  </Paragraph>
  <Paragraph>
    Se han agregado nuevos controles a WPF para facilitar
    la creación de aplicaciones empresariales. Estos controles
    son casi cien por cien compatibles con las versiones de
    <Span FontStyle="Italic">Silverlight</Span>. Esto permite
    a los desarrolladores reutilizar el código y crear
    rápidamente versiones web y de cliente. Entre estos
    controles destacamos los siguientes:  </Paragraph>
  <List MarkerOffset="20">
    <ListItem>
      <Paragraph>
        <Span Tag="Control 1">DataGrid</Span>
      </Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>
        <Span Tag="Control 2">Calendar</Span>
      </Paragraph>
    </ListItem>
    <ListItem>
      <Paragraph>
        <Span Tag="Control 3">DataPicker</Span>
      </Paragraph>
    </ListItem>
  </List>
</FlowDocument>
```

Para navegar por la estructura de un documento dinámico, las siguientes indicaciones pueden ser útiles:

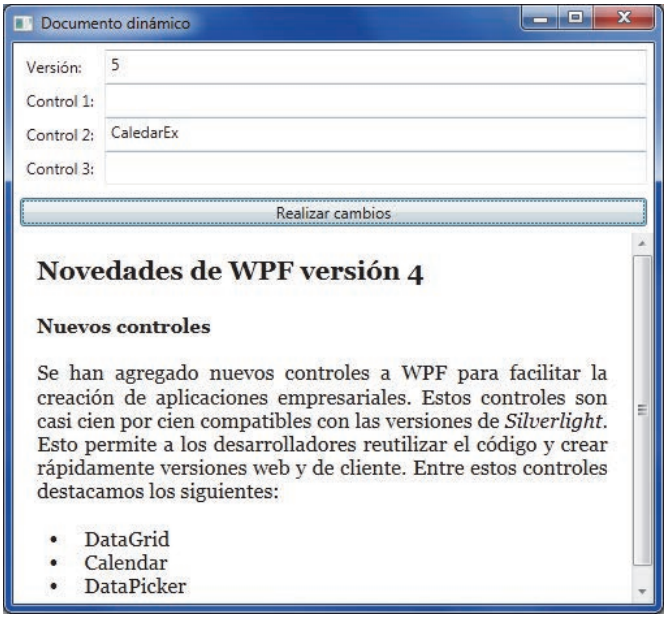
- Para obtener los **Block** hay que acceder a la colección **Blocks** del documento **FlowDocument**.

- Para determinar la clase de un **Block** obtenido de la colección, basta con convertirlo a la clase esperada (por ejemplo, *Dim parrafo As Paragraph = TryCast(block, Paragraph)*); el resultado será **Nothing** si la conversión no es posible. En el ejemplo, si *block* no es un objeto **Paragraph** el valor de *parrafo* será **Nothing**, en otro caso, el valor será la referencia al objeto **Paragraph** (*block*).

- Un elemento **Block** o **Inline** puede contener otros elementos. Por ejemplo, el elemento **List** tiene una colección **ListItems** de elementos **ListItem** y el elemento **Paragraph** tiene una colección **Inlines** de elementos **Inline**.

- Aislar lo que realmente se quiere cambiar usando elementos **Span**. Por ejemplo, en el documento dinámico anterior se identifican los elementos *Versión*, *Control 1*, *Control 2* y *Control 3* para facilitar la programación de los cambios de los fragmentos de texto que representan.

Para poner en práctica lo explicado, vamos a añadir a la aplicación que venimos desarrollando una nueva ventana de diálogo que muestre el documento dinámico y permita cambiarlo con los datos que el usuario escriba en cajas de texto. Habrá una caja de texto por cada elemento que se pueda cambiar y serán añadidas al formulario dinámicamente, en función del número de elementos modificables. La ventana tendrá el aspecto siguiente:



La estructura del contenido de esta ventana es la siguiente:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid Grid.Row="0" Name="gridCambios" Margin="5">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
  </Grid>
  <Button Grid.Row="1" Margin="5" Click="btCambios_Click">Realizar cambios />
  <FlowDocumentScrollViewer Grid.Row="2" Name="VisorDoc" >
    <FlowDocument Name="documento">
      ...
    </FlowDocument>
  </FlowDocumentScrollViewer>
</Grid>
```

Para hacer el ejemplo lo más genérico posible, el código que escribamos para realizar los cambios sobre el documento dinámico no tendrá conocimiento sobre dicho documento. Esto es, estará escrito de forma genérica para obtener de todos los párrafos sólo los elementos **Span** identificados por su propiedad **Tag** (esta propiedad es de tipo **Object** y su contenido será el que mostrarán las etiquetas que identifican las cajas). Para ello, simplemente se debe recorrer la colección de cada **Block** del documento en busca de párrafos y por cada párrafo, recorrer su colección **Inlines** en busca de objetos **Span** con **Tag** no igual a **Nothing**. Cada vez que se encuentra un objeto **Span**, se crea una etiqueta y una caja de texto que el usuario puede utilizar para proporcionar un nuevo valor para el elemento al que se refiere la etiqueta. Una vez creados, se agregan a una rejilla ubicada encima del documento. Así mismo, para facilitar el proceso de sustitución, almacenaremos en la propiedad **Tag** de la caja de texto una referencia al elemento **Run** correspondiente al texto del elemento **Span** correspondiente:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
  ' Buscar los elementos Span que se pueden cambiar
  Buscar_Paragraph(documento.Blocks)
  For Each block As Block In documento.Blocks
    Dim lista As List = TryCast(block, List)
    If lista IsNot Nothing Then
      For Each listItem As ListItem In lista.ListItems
        Buscar_Paragraph(listItem.Blocks)
      Next
```

```
    End If
  Next
End Sub

Private Sub Buscar_Paragraph(blockCollection As BlockCollection)
  ' Buscar elementos Paragraph
  For Each block As Block In blockCollection
    Dim parrafo As Paragraph = TryCast(block, Paragraph)
    If parrafo IsNot Nothing Then
      Buscar_Span(parrafo)
    End If
  Next
End Sub

Private Sub Buscar_Span(parrafo As Paragraph)
  ' Buscar elementos Span
  For Each inline As Inline In parrafo.Inlines
    Dim span As Span = TryCast(inline, Span)
    If span IsNot Nothing AndAlso inline.Tag IsNot Nothing Then
      ' Añadir una etiqueta y una caja de texto por cada Span
      Dim fila As New RowDefinition()
      gridCambios.RowDefinitions.Add(fila)

      Dim etiqueta As New Label()
      etiqueta.Content = inline.Tag.ToString() & ": "
      Grid.SetColumn(etiqueta, 0)
      Grid.SetRow(etiqueta, gridCambios.RowDefinitions.Count - 1)
      gridCambios.Children.Add(etiqueta)

      Dim cajaTexto As New TextBox()
      Grid.SetColumn(cajaTexto, 1)
      Grid.SetRow(cajaTexto, gridCambios.RowDefinitions.Count - 1)
      cajaTexto.Tag = span.Inlines.FirstInline
      ' elemento Run
      gridCambios.Children.Add(cajaTexto)
    End If
  Next
End Sub
```

Finalmente, cuando el usuario haga clic en el botón *Realizar cambios*, el controlador de este botón recorrerá la colección **Children** de la rejilla *gridCambios*, que contiene las etiquetas y las cajas de texto correspondientes, buscando las cajas de texto y obteniendo a partir de su propiedad **Tag** el elemento **Run** que se modificará con el contenido de la caja:

```
Private Sub btCambios_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  For Each elmentoEnGridCambios As UIElement In gridCambios.Children
    If Grid.GetColumn(elmentoEnGridCambios) = 1 Then
```

```
      Dim cajaTexto As TextBox = DirectCast(elmentoEnGridCambios, TextBox)
      If cajaTexto.Text <> "" Then
        DirectCast(cajaTexto.Tag, Run).Text = cajaTexto.Text
      End If
    End If
  Next
End Sub
```

Acceso a documentos en un fichero

Hasta aquí, hemos visto cómo construir un documento dinámico. Pero, una vez construido, seguro que, en más de una ocasión, necesitaremos reutilizarlo en la misma o en otras aplicaciones. Para hacer esto posible, necesitamos guardar dicho documento en un fichero en el disco para, posteriormente, poder cargarlo dinámicamente desde la aplicación que necesite reutilizarlo.

Para guardar un documento dinámico en un fichero podemos utilizar el método **Save** de la clase **XamlWriter** del espacio de nombres **System.Windows.Markup** y para leerlo, el método **Load** de la clase **XamlReader**.

Por ejemplo, el código para guardar el contenido de un **FlowDocument** podría ser el siguiente:

```
Using fs As FileStream = File.Open(NombreFichero, FileMode.Open)
  Dim documento As FlowDocument =
        TryCast(XamlReader.Load(fs), FlowDocument)
  If documento Is Nothing Then
    MessageBox.Show("Error al cargar documento.")
  Else
    fdocReader1.Document = documento
  End If
End Using
```

y el código para guardar el contenido de un documento, este otro:

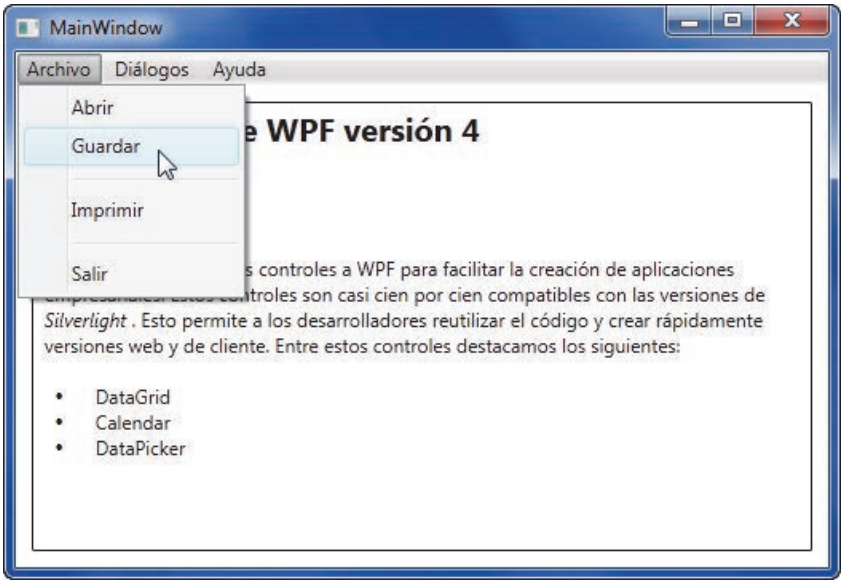
```
Using fs As FileStream = File.Open(NombreFichero, FileMode.Create)
  XamlWriter.Save(fdocReader1.Document, fs)
End Using
```

Editar un documento

Cuando empezamos a hablar de controles de documentos WPF dijimos que los controles **FlowDocumentReader**, **FlowDocumentPageViewer** y **FlowDocumentScrollViewer** estaban diseñados para mostrar contenido **FlowDocument**, pero que no permitían la edición, que si se necesitaba esta capacidad, **RichText-**

Box era una buena opción. Este control ha formado parte de las diferentes bibliotecas de .NET y ahora de WPF. No obstante, la versión que incluye WPF es significativamente diferente a la de sus predecesores. En esta versión, el control ya no está limitado a datos RTF estándar, característicos de los procesadores de textos, sino que su contenido se corresponde con un objeto **FlowDocument**. Esto quiere decir que admite contenido RTF, pero internamente utiliza el modelo de flujo de contenido que acabamos de exponer, haciendo mucho más fácil la manipulación del contenido del documento mediante programación. **FlowDocument** es el único elemento secundario admitido por **RichTextBox**. El único inconveniente es la velocidad, como ocurre con la mayoría de los controles de texto enriquecido.

Como ejemplo, vamos a añadir a la ventana principal del proyecto que venimos desarrollando (*MainWindow.xaml*) un control **RichTextBox**, para después implementar las órdenes *Abrir* y *Guardar* del menú *Archivo* de la aplicación que permitan cargar o guardar el documento que visualice dicho control.



Vamos a partir de que hemos generado el documento dinámico que presenta la figura anterior y que ahora deseamos guardarlo haciendo clic en la orden *Guardar* del menú *Archivo*. Para ello, lo primero que vamos a hacer es añadir el controlador para esta orden y después lo completaremos con el código que guarde el documento en el fichero especificado.

El contenido de este controlador podría ser el código que expusimos anteriormente para guardar un fichero utilizando la clase **XamlWriter**. Sin embargo, podríamos tener la necesidad de guardar el contenido en otros formatos como, por ejemplo, RTF. Para hacer esto necesitamos utilizar la clase **TextRange** del espa-

cio de nombres **System.Windows.Documents**. Un objeto de esta clase representa una selección de un contenido entre dos posiciones **TextPointer** (por ejemplo del contenido de un documento dinámico). Y un objeto **TextPointer** representa una posición dentro de un **FlowDocument** o **TextBlock**. Por ejemplo:

```
Dim selección As New TextRange(richTextBox.Document.ContentStart,
                               richTextBox.Document.ContentEnd)
```

El ejemplo anterior crea un objeto *selección* que contiene la porción de texto que hay entre las dos posiciones especificadas (**ContentStart** y **ContentEnd**) del documento; en este caso, todo el texto del **FlowDocument** del control **RichTextBox**. Ahora, para guardar en un fichero este contenido sólo hay que invocar al método **Save** de **TextRange**:

```
selección.Save(fStream, DataFormats.Rtf)
```

El método **Save** guarda la selección actual del **FlowDocument** en el fichero vinculado con el flujo *fStream* con el formato de datos especificado; en este caso, *Rtf*. Según lo expuesto, puede completar el controlador de la orden *Guardar* así:

```
Private Sub ArchivoGuardar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim DlgGuardar As New SaveFileDialog()

  DlgGuardar.Filter = "ficheros xaml (*.xaml)|*.xaml" & _
                      "|ficheros rtf (*.rtf)|*.rtf|Todos (*.*)|*.*"
  DlgGuardar.FilterIndex = 2

  If DlgGuardar.ShowDialog() = True Then
    Dim fStream As FileStream
    If richTextBox.Document IsNot Nothing Then
      Dim selección As New TextRange( _
                           richTextBox.Document.ContentStart,
                           richTextBox.Document.ContentEnd)
      fStream = New FileStream(DlgGuardar.FileName,
                               FileMode.Create)
      If DlgGuardar.FileName.ToUpper().EndsWith(".XAML") Then
        selección.Save(fStream, DataFormats.Xaml)
      Else
        selección.Save(fStream, DataFormats.Rtf)
      End If
      fStream.Close()
    End If
  End If
End Sub
```

Obsérvese que el código anterior obtiene el nombre del fichero, crea un objeto **TextRange** para contener la parte del documento que deseamos guardar y guarda dicho contenido en el fichero seleccionado con el formato elegido.

La operación inversa, cargar (**Load**) el contenido de un fichero para ser mostrado en un control **RichTextBox** es análoga. Se seleccionan el formato y el nombre del fichero (el fichero tiene que existir), se crea el documento dinámico, inicialmente vacío, que va a mostrar el control **RichTextBox**, se crea un objeto **TextRange** que incluya la selección de todo ese documento, se invoca al método **Load** de **TextRange** para almacenar en la selección actual el contenido del fichero y se asigna el documento dinámico a la propiedad **Document** del control **RichTextBox** para que lo muestre. Según lo expuesto y continuando con el desarrollo de la aplicación, añada el controlador para la orden *Abrir* y complételo como se indica a continuación:

```
Private Sub ArchivoAbrir_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  Dim DlgAbrir As New OpenFileDialog()

  DlgAbrir.Filter = "ficheros xaml (*.xaml)|*.xaml" & _
                    "|ficheros rtf (*.rtf)|*.rtf|Todos (*.*)|*.*"
  DlgAbrir.FilterIndex = 2
  ' Mostrar el diálogo Abrir
  If DlgAbrir.ShowDialog() = True Then
    Dim flowDocument As New FlowDocument()
    Dim textRange As New TextRange(flowDocument.ContentStart,
                                   flowDocument.ContentEnd)
    Dim fStream As New FileStream(DlgAbrir.FileName, FileMode.Open)
    If DlgAbrir.FileName.ToUpper().EndsWith(".XAML") Then
      textRange.Load(fStream, DataFormats.Xaml)
    Else
      textRange.Load(fStream, DataFormats.Rtf)
    End If
    fStream.Close()
    richTextBox.Document = flowDocument
  End If
End Sub
```

Imprimir un documento

Para imprimir un documento dinámico sólo hay que invocar al método **Print** de su contenedor. Todos los contenedores objetos **FlowDocument** (excepto **RichTextBox**) implementan la orden **Print**. El método **Print** mostrará el diálogo **Print** para que el usuario pueda elegir la impresora, el número de copias y otras opciones. Por ejemplo, el siguiente código XAML implementa la orden **Print** para imprimir el documento mostrado por el contenedor *fdocReader1*:

```
<MenuItem Name="ArchivoImprimir" Header="Imprimir"
          Command="ApplicationCommands.Print"
          CommandTarget="{Binding ElementName=fdocReader1}"/>
```

Otra opción válida para todos los contenedores de documentos dinámicos, incluido **RichTextBox**, es utilizar el método **PrintVisual** del diálogo **PrintDialog**. Este método imprime un objeto visual (no texto, sino un objeto que se deriva de la clase **Visual**), en la impresora seleccionada. Por ejemplo, añada una orden *Imprimir* al menú *Archivo* de la aplicación anterior y escriba el controlador vinculado con la misma como se indica a continuación:

```
Private Sub ArchivoImprimir_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim descripcionDoc As String = "Documento acerca de WPF"
  Dim DlgImprimir As New PrintDialog()
  If DlgImprimir.ShowDialog() = True Then
    DlgImprimir.PrintVisual(richTextBox, descripcionDoc)
  End If
End Sub
```

Observe el primer parámetro del método **PrintVisual**: se trata del control **RichTextBox** (se deriva de **Visual**). El segundo parámetro es una descripción del trabajo que se va a imprimir.

Otra opción más para imprimir un documento dinámico es utilizar el método **PrintDocument** de **PrintDialog**. La ventaja de este nuevo enfoque es que podemos manipular documentos con muchas páginas.

El método **PrintDocument** imprime el objeto **DocumentPaginator** pasado como argumento en la impresora seleccionada. Esto es, este método no espera como argumento el documento dinámico a imprimir, sino un objeto que proporcione dicho documento paginado:

```
Sub PrintDocument(paginador_del_documento As DocumentPaginator,
                  descripción As String)
```

DocumentPaginator es una clase abstracta especializada que tiene como función tomar el contenido de un documento dinámico, dividirlo en páginas y suministrar cada una de ellas cuando le sean solicitadas. Cada página es un objeto de la clase **DocumentPage** que envuelve un objeto **Visual**, la página a imprimir, lo cual significa que **PrintDocument**, comparado con **PrintVisual**, imprime varios objetos **Visual**, uno cada vez.

Entonces, ¿cómo obtenemos un objeto **DocumentPaginator** de **FlowDocument**? La clase **FlowDocument** implementa la interfaz **IDocumentPaginatorSource** que proporciona la propiedad **DocumentPaginator** que define el objeto

que realiza la paginación del contenido real del documento. Por lo tanto, el truco está en convertir el **FlowDocument** a un **IDocumentPaginatorSource** y luego utilizar la propiedad **DocumentPaginator**. Por ejemplo:

```
Private Sub ArchivoImprimir_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  Dim descripcionDoc As String = "Documento"
  Dim DlgImprimir As New PrintDialog()
  If DlgImprimir.ShowDialog() = True Then
    ' Imprimir el documento paginado existente
    Dim docFuente As IDocumentPaginatorSource = richTextBox.Document
    DlgImprimir.PrintDocument(docFuente.DocumentPaginator,
                              descripcionDoc)
  End If
End Sub
```

Dependiendo del contenedor que estemos utilizando para mostrar el documento dinámico, la impresión del documento puede ser o no satisfactoria. Si no es de nuestro gusto podemos personalizar la impresión. Por ejemplo, es posible que queramos establecer propiedades como **ColumnWidth** y **ColumnGap** para establecer un número de columnas. El problema de este enfoque es que una vez que se han cambiado las propiedades deseadas, los cambios se aplican también al contenedor que muestra el documento. Si no deseamos esto, lo que podemos hacer es almacenar los valores originales para luego utilizarlos para deshacer los cambios. Por ejemplo:

```
Private Sub ArchivoImprimir_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  Dim descripcionDoc As String = "Documento"
  Dim DlgImprimir As New PrintDialog()

  If DlgImprimir.ShowDialog() = True Then
    Dim docFuente As FlowDocument = richTextBox.Document

    ' Guardar la configuración actual
    Dim pageWidth As Double = docFuente.PageWidth
    Dim pageHeight As Double = docFuente.PageHeight
    Dim columnGap As Double = docFuente.ColumnGap
    Dim columnWidth As Double = docFuente.ColumnWidth
    Dim pagePadding As Thickness = docFuente.PagePadding

    ' Hacer coincidir la página FlowDocument con la página impresa
    docFuente.PageHeight = DlgImprimir.PrintableAreaHeight
    docFuente.PageWidth = DlgImprimir.PrintableAreaWidth
    docFuente.PagePadding = New Thickness(50)

    ' Utilizar dos columnas
    docFuente.ColumnGap = 24
```

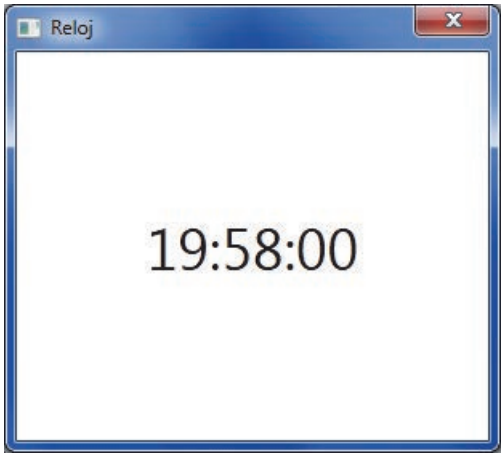
```
    docFuente.ColumnWidth =
      (docFuente.PageWidth - docFuente.ColumnGap -
       docFuente.PagePadding.Left - docFuente.PagePadding.Right) / 2

    Dim docFuentePag As IDocumentPaginatorSource = docFuente
    DlgImprimir.PrintDocument(docFuentePag.DocumentPaginator,
                              descripcionDoc)

    ' Restablecer la configuración inicial
    docFuente.PageWidth = pageWidth
    docFuente.PageHeight = pageHeight
    docFuente.ColumnGap = columnGap
    docFuente.ColumnWidth = columnWidth
    docFuente.PagePadding = pagePadding
  End If
End Sub
```

TEMPORIZADORES Y MODELO DE SUBPROCESOS

Supongamos que deseamos realizar una aplicación que muestre un reloj digital similar al de la figura siguiente. Evidentemente, la ventana mostrará en todo momento la hora actual. Esto nos exigirá utilizar un temporizador.



Un temporizador se corresponde con un objeto que notifica periódicamente a la aplicación que lo crea cuándo ha transcurrido un período predeterminado de tiempo. Cada vez que transcurre ese período de tiempo, especificado cuando se creó el temporizador, el objeto genera un evento "tiempo transcurrido" y espera una respuesta por parte de la aplicación que lo creó.

Hay dos temporizadores en WPF: **System.Timers.Timer** y **System.Windows.Threading.DispatcherTimer**.

El temporizador **Timer** basado en servidor genera un evento **Elapsed** cada vez que transcurre el intervalo de tiempo especificado por su propiedad **Interval**. Está diseñado para ser utilizado con subprocesos de trabajo en un entorno multiproceso y se ejecuta en un subproceso diferente al subproceso de la interfaz de usuario (IU). Pero WPF requiere que la mayor parte de sus objetos estén vinculados al subproceso de la IU. Esto se conoce como afinidad de subprocesos, lo que significa que un objeto WPF sólo puede ser accedido en el subproceso en el que fue creado.

La afinidad de subprocesos se controla mediante un objeto de la clase **Dispatcher**: un bucle de mensajes con prioridad para aplicaciones WPF. Normalmente, los proyectos de WPF tienen un único objeto **Dispatcher** (y por lo tanto un único subproceso de interfaz de usuario) a través del cual se canaliza todo el trabajo de la interfaz de usuario.

Según dijimos en el capítulo *Introducción a WPF*, la mayoría de los objetos de WPF se derivan de la clase **DispatcherObject** que proporciona las estructuras básicas para tratar con la simultaneidad y el subprocesamiento. Pues bien, en WPF, sólo se puede tener acceso al objeto **DispatcherObject** mediante el objeto **Dispatcher** asociado con el objeto WPF. Por ejemplo, un subproceso secundario (piense en el subproceso en el que se ejecuta un **Timer**) no puede actualizar el contenido de un objeto **Label** asociado a **Dispatcher** en el subproceso de la IU. Para que el subproceso secundario tenga acceso a la propiedad **Content** de **Label**, dicho subproceso debe delegar el trabajo en el objeto **Dispatcher** asociado al subproceso de la IU, lo que exige utilizar **Invoke** o **BeginInvoke**. La operación se agregaría a la cola de **Dispatcher**.

En cambio, el temporizador **DispatcherTimer** se integra en la cola del objeto **Dispatcher** de un objeto **DispatcherObject**, con lo que no presenta la problemática del temporizador **Timer**; no obstante, no es tan preciso como éste. Al colocar las operaciones de **DispatcherTimer** en la cola del objeto **Dispatcher** no se puede garantizar que el temporizador se ejecute exactamente cuando finaliza el intervalo de tiempo, porque depende de las otras operaciones de la cola y de sus prioridades, aunque nunca se ejecutarán antes de que finalice dicho intervalo. Este temporizador genera un evento **Tick** cada vez que transcurre el intervalo de tiempo especificado por la propiedad **Interval**.

Timer

A continuación, vamos a resolver el problema planteado anteriormente de realizar una aplicación que muestre un reloj digital. En principio, mostraremos la solución utilizando un objeto **Timer** y después, utilizando un objeto **DispatcherTimer**.

Suponiendo que ya ha creado un nuevo proyecto de tipo *Aplicación WPF*, para implementar un temporizador **Timer** en una aplicación, siga los pasos especificados a continuación:

1. Cree un objeto de la clase **Timer**. Para ello, defina en la clase *MainWindow* un atributo *temporizador* de tipo **Timer**. A continuación añada un método *CrearTemporizador* que cree el objeto **Timer** y establezca sus propiedades según indica el código mostrado a continuación. Finalmente, invoque a este método desde el constructor *MainWindow*:

```
Class MainWindow
  Private temporizador As System.Timers.Timer

  Public Sub New()
    InitializeComponent()
    CrearTemporizador()
  End Sub

  Private Sub CrearTemporizador()
    temporizador = New System.Timers.Timer()
    temporizador.Interval = 1000
    AddHandler temporizador.Elapsed, AddressOf Temporizador_Elapsed
    temporizador.Enabled = True
  End Sub
  ...

End Class
```

   La propiedad **Interval** es el intervalo de tiempo en milisegundos que tiene que transcurrir para que el objeto genere un evento **Elapsed** (el tiempo pasó) independientemente de las acciones del usuario. La propiedad **Enabled** puesta a **True** hace que se generen eventos **Elapsed** cada vez que transcurre el intervalo de tiempo establecido. Finalmente, el evento **Elapsed** hace referencia al manejador de este tipo de eventos; en nuestro ejemplo, cada vez que se genere el evento **Elapsed** se invocará al método *Temporizador_Elapsed*.

   Establecer la propiedad **Enabled** a **True** es lo mismo que llamar al método **Start**, y establecerla a **False** es lo mismo que llamar al método **Stop**.

   Si la propiedad **Enabled** es **True** y **AutoReset** es **False**, el evento **Elapsed** sólo se genera una vez, la primera vez que transcurre el intervalo.

2. Agregue el método *Temporizador_Elapsed* para manejar el evento **Elapsed**. Su firma coincidirá con la especificada por el delegado **ElapsedEventHandler**:

```
Private Sub Temporizador_Elapsed(sender As Object,
e As System.Timers.ElapsedEventArgs)
  ...
End Sub
```

3. Añada a la ventana una etiqueta *etHora* para que muestre la hora actual:

```
<Label Name="etHora" Content="00:00:00"
       Margin="68,86,68,86" FontSize="36"
       HorizontalContentAlignment="Center"
       VerticalContentAlignment="Center" />
```

La aplicación está finalizada. Sólo queda escribir en el método *Temporizador_Elapsed* el código que mostrará la hora. Pruebe a escribir lo siguiente:

```
etHora.Content = DateTime.Now.ToLongTimeString()
```

Si ahora ejecuta la aplicación, observará que el reloj no funciona y si la ejecuta en modo depuración (*F5*), podrá observar que cada vez que se ejecuta la línea anterior se lanza una excepción debido a que se está ejecutando una operación no válida. ¿Por qué? Por lo explicado anteriormente: en WPF, un subproceso secundario (en nuestro caso el **Timer**) no puede actualizar el contenido del objeto *etHora* en el subproceso de la IU. La solución es que este subproceso secundario delegue en el objeto **Dispatcher** asociado al subproceso de la IU, para lo cual utilizaremos la llamada asíncrona **BeginInvoke**:

```
Private Sub Temporizador_Elapsed(sender As Object,
e As System.Timers.ElapsedEventArgs)
  Dispatcher.BeginInvoke(New Action(AddressOf ActualizarReloj))
End Sub

Private Sub ActualizarReloj()
  etHora.Content = DateTime.Now.ToLongTimeString() ' hora actual
End Sub
```

Action es un delegado que encapsula un método que no tiene parámetros y no devuelve un valor.

Resolución del temporizador

Tenga en cuenta que en órdenes de unos pocos milisegundos (o microsegundos, dependiendo de su sistema hardware, sistema operativo, máquina virtual de .NET, API de .NET, etc.) y en ámbitos normales de ejecución, el intervalo programado para el temporizador podrá no cumplirse si está por debajo de lo que su sistema en ese momento puede soportar como valor mínimo. Para conocer este valor puede realizar un test a su máquina añadiendo a la clase *MainWindow* el código siguiente:

```
Class MainWindow
  Private temporizador As System.Timers.Timer
  Private tAntes As TimeSpan
  Private tDespues As TimeSpan

  Public Sub New()
    InitializeComponent()
    CrearTemporizador()
  End Sub

  Private Sub CrearTemporizador()
    ' ...
  End Sub

  Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    tAntes = DateTime.Now.TimeOfDay
    temporizador.Interval = 1 ' milisegundo
  End Sub

  Private Sub Temporizador_Elapsed(sender As Object,
  e As System.Timers.ElapsedEventArgs)
    Dispatcher.BeginInvoke( _
                New Action(AddressOf ActualizarReloj))
    tDespues = DateTime.Now.TimeOfDay
    System.Diagnostics.Debug.WriteLine( _
     Convert.ToString((tDespues - tAntes).Milliseconds) & " mseg.")
    tAntes = tDespues
  End Sub

  Private Sub ActualizarReloj()
    etHora.Content = DateTime.Now.ToLongTimeString() ' hora actual
  End Sub
End Class
```

En el código anterior se puede observar que el método que responde al evento **Loaded** de la ventana toma la hora actual (con una precisión de microsegundos) almacenándola en un objeto **TimeSpan** (un objeto **TimeSpan** representa un intervalo de tiempo) e inicia la propiedad **Interval** a 1 milisegundo, tiempo que tiene que transcurrir para que el temporizador genere un evento **Elapsed**. Después, cada vez que *Temporizador_Elapsed* responde a este evento, se muestra el tiempo en milisegundos que ha transcurrido, que lógicamente estará por encima del programado. Ese valor será su resolución máxima.

Cuando una aplicación termine con el temporizador, es bueno llamar a su método **Stop** para detener la generación de los eventos de acción.

Tenga también presente que, si su aplicación u otra aplicación está realizando una tarea que mantiene ocupados los recursos del ordenador por un espacio largo

de tiempo, tal como un bucle largo, cálculos intensivos, acceso a los puertos, etc., puede ser que la interfaz de usuario no responda de acuerdo con los intervalos de tiempo programados; la solución sería utilizar para realizar estas tareas pesadas otro subproceso.

DispatcherTimer

Utilizando un temporizador de la clase **DispatcherTimer**, el código del ejemplo anterior lo modificaríamos así:

```
Imports System.Windows.Threading

Class MainWindow
  Private temporizador As DispatcherTimer
  Private tAntes As TimeSpan
  Private tDespues As TimeSpan

  Public Sub New()
    InitializeComponent()
    CrearTemporizador()
  End Sub

  Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    tAntes = DateTime.Now.TimeOfDay
    temporizador.Interval = New TimeSpan(10000) ' ticks
  End Sub

  Private Sub CrearTemporizador()
    temporizador = New DispatcherTimer()
    temporizador.Interval = New TimeSpan(0, 0, 1)
    AddHandler temporizador.Tick, AddressOf Temporizador_Tick
    temporizador.Start()
  End Sub

  Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
    etHora.Content = DateTime.Now.ToLongTimeString()
    tDespues = DateTime.Now.TimeOfDay
    System.Diagnostics.Debug.WriteLine( _
     Convert.ToString((tDespues - tAntes).Milliseconds) & " mseg.")
    tAntes = tDespues
  End Sub
End Class
```

Un *tick* son 100 nanosegundos. Compile la aplicación, ejecútela y pruebe los resultados.

RESUMEN

Este capítulo empieza exponiendo lo que son las cajas de diálogo modales y no modales, para después pasar de las generalidades a lo concreto, explicando cómo construir una caja de diálogo, cómo mostrarla, cómo gestionar sus botones *Aceptar* y *Cancelar* y cómo recuperar los datos solicitados a través de la misma. También se ha explicado cómo construir una caja de diálogo muy común: el diálogo *Acerca de*. Una aplicación puede mostrar varios diálogos y, quizás, algunos de ellos necesiten comunicarse con su ventana propietaria, por ejemplo para intercambiar datos, cuestión que también se ha expuesto.

Evidentemente, la interfaz de una caja de diálogo, en general de una ventana, la forman los controles con los que interactúa el usuario. Por ello, hemos dedicado una parte importante de este capítulo a estudiar los controles más comunes como son los distintos tipos de botones y listas, los controles de rango definido, un control con pestañas, un control para la gestión de fechas y los controles ListView, TreeView y DataGrid.

Los controles pueden ser utilizados para solicitar datos al usuario o para tomarlos de un origen de datos y mostrárselos al usuario para que simplemente los vea o para que pueda interactuar con ellos; esto nos ha llevado a introducir, en los ejemplos que se han ido desarrollando para ver trabajar a los distintos controles, los conceptos de orígenes de datos y de enlace con los mismos. También, en los diseños, en ocasiones hemos necesitado diseñar plantillas de contenido para que otros controles pudieran mostrar su contenido bajo un diseño concreto.

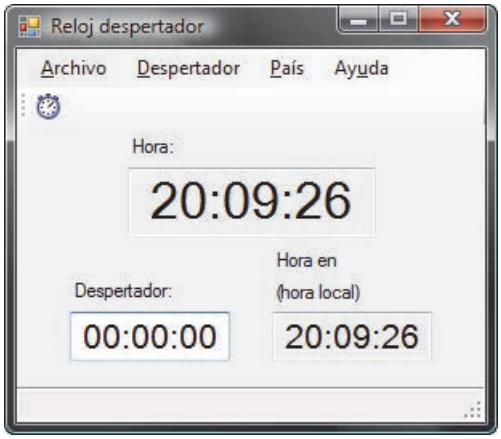
Se supone que con todo lo expuesto hasta ahora estamos en condiciones de construir múltiples y variadas cajas de diálogo personalizadas, pero sin olvidarnos de las cajas de diálogo de uso diario y que están predefinidas en WPF, listas para usar, como son los diálogos *Abrir* y *Guardar*.

Algo también muy común es el trabajo con documentos. De ahí que no podíamos finalizar este capítulo sin hablar de los controles de documentos WPF.

Finalmente, terminamos este capítulo realizando un ejemplo para introducirnos en el uso de los temporizadores con el objeto de proponer el diseño de una aplicación que ayude al lector a afianzar todo lo estudiado.

EJERCICIOS PROPUESTOS

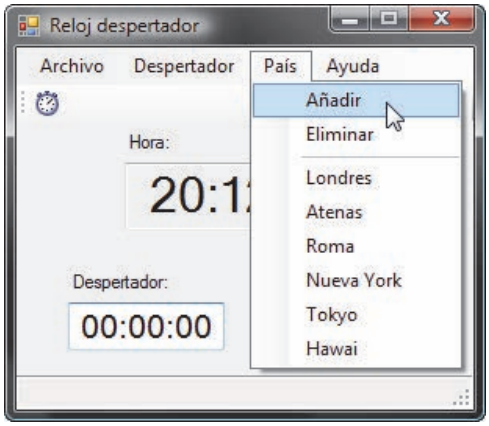
1. Se propone completar el diseño del reloj despertador digital anterior para que ahora presente un aspecto como el que se muestra en la figura siguiente:



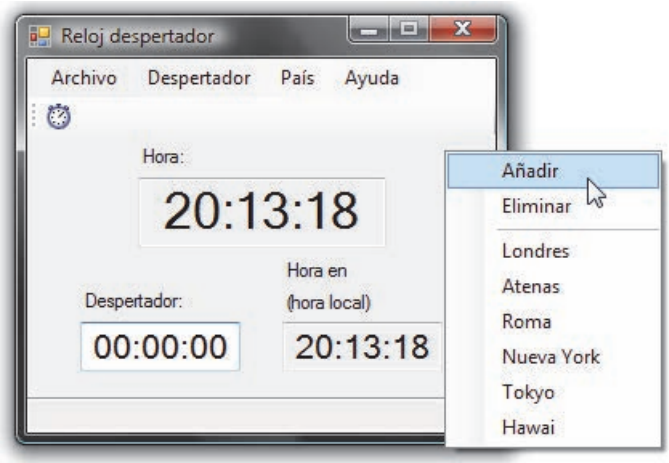
El reloj tiene una pantalla, *ctHora*, para visualizar la hora y una caja de texto denominada *ctDespertador* donde el usuario puede escribir la hora a la que quiere ser avisado. Para activar o desactivar el despertador, el usuario dispone de un menú denominado *Despertador*.

La pantalla para visualizar la hora será una caja de texto de sólo lectura con el fin de que el usuario no pueda modificarla. Para que la hora varíe segundo a segundo, debe actualizarse el contenido de la caja de texto que representa la hora, a intervalos iguales o inferiores a un segundo.

Hay otro menú, denominado *País*, que permitirá al usuario añadir países al propio menú, para después, haciendo clic sobre cualquiera de ellos, obtener en una tercera caja de texto, *ctPais*, la hora actual en ese país. Inicialmente, estos datos pueden ser recuperados de un fichero y almacenados en una matriz de objetos (país, diferencia horaria). Según esto, podemos posponer el diseño de las órdenes *Añadir* y *Eliminar* país para cuando hayamos estudiado cajas de diálogo en el siguiente capítulo. Ídem con respecto a la orden *Ayuda*.



Finalmente, queremos que se visualice un menú flotante idéntico al mostrado por *País*, según se puede observar en la figura siguiente:



Para la realización del ejercicio se recomienda seguir los siguientes pasos:

1. Diseñar un formulario con: barra de menús, barra de herramientas, barra de estado (si lo desea), tres etiquetas, dos cajas de texto, una caja de texto para el despertador con una regla de validación personalizada que verifique que la hora introducida es correcta y un temporizador.

   El menú *Archivo* tendrá un elemento *Salir* que cerrará el formulario. El menú *Despertador* tendrá un elemento que alternará entre *Activar* y *Desactivar*. El menú *País* tendrá, inicialmente, un elemento *Añadir*, otro *Eliminar* y un separador; éstos dos últimos permanecerán ocultos cuando no haya países. El menú *Ayuda* tendrá un elemento *Acerca de...*

   La barra de herramientas tendrá un botón que alternará entre *Activar* y *Desactivar* el despertador.

2. Presentar la hora en la caja de texto *ctHora*, lo que supone añadir un controlador asociado con *Temporizador*. Presentarla también en *ctHoraPais*.

3. Limitar los caracteres en la caja de texto *ctDespertador* a dígitos y dos puntos.

4. Añadir a la ventana principal un método, *ObtenerHoraDespertador*, que devuelva la hora del despertador en *ticks* (pasos; 1 paso = 100 nanosegundos).

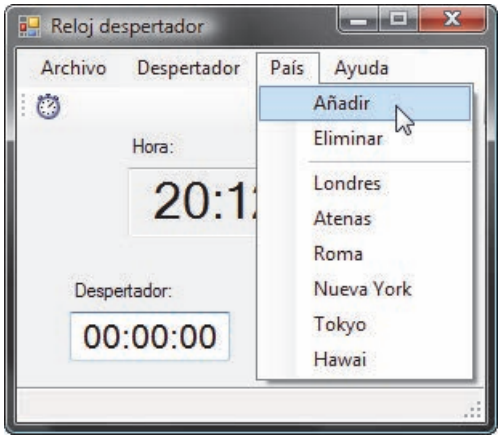
5. Modificar el controlador asociado con *Temporizador* para que haga sonar la alarma, si procede. Cuando suene la alarma y no sea desactivada, lo hará como máximo durante cinco minutos.

6. Añadir el controlador de la orden *Activar* del menú *Despertador*. Esta orden alternará entre *Activar* y *Desactivar* la alarma. Habilitar también, en el mismo sentido, el botón de la barra de herramientas.

7. Añadir un menú contextual, inicialmente con una orden *Añadir*, otra *Eliminar* y un separador; estos dos últimos permanecerán ocultos cuando no haya países.

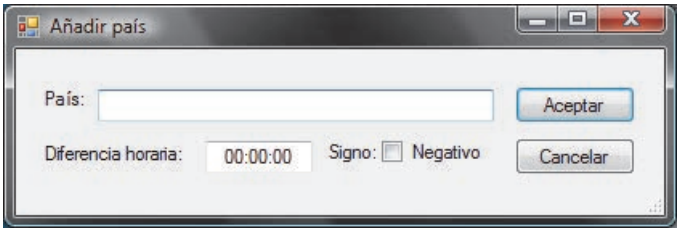
8. El proceso de añadir países lo posponemos hasta el capítulo siguiente. Ahora podemos realizar una simulación análoga a la que fue expuesta anteriormente en este mismo capítulo, tanto para el menú *País* como para el contextual.

2. Se propone completar el diseño del reloj despertador digital realizado en el ejercicio 1 de este apartado. El reloj tenía el aspecto mostrado en la figura siguiente:



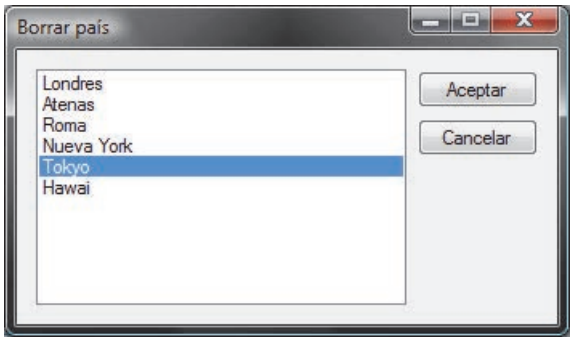
Allí pospusimos la inclusión de las órdenes *Añadir* país y *Eliminar* país para cuando estudiáramos cajas de diálogo, tema que ahora ya conocemos.

Cuando el usuario ejecute la orden *Añadir* país se visualizará una caja de diálogo como la siguiente:



Esta caja permitirá al usuario introducir el nombre de un país y la diferencia horaria existente indicando si es negativa o positiva. Estos datos serán almacenados en una colección de objetos de tipo **List** y, además, el nombre del país dará lugar a un elemento nuevo en los menús *País* y contextual.

Cuando el usuario ejecute la orden *Eliminar* país es porque desea quitar un país de los menús *País* y contextual. Esta operación requiere también actualizar la colección de objetos. Para ello, se visualizará la caja de diálogo *Borrar país* mostrada a continuación, la cual permitirá seleccionar de una lista el nombre del país que desea eliminar.



Para que los datos persistan de una ejecución a otra, guarde en un fichero el par de datos *país-diferencia horaria* cuando se cierre la aplicación y recupérelos cuando se inicie. Estas operaciones resultarán sencillas si encapsula esos datos en objetos de una clase. Añada también al menú *Archivo* dos nuevas órdenes, *Abrir* y *Guardar*, que permitan cargar y guardar, respectivamente, un fichero con esos datos.

Finalmente, la orden *Ayuda* visualizará otra caja de diálogo con una breve explicación acerca de la aplicación.

Para la realización del ejercicio se recomienda seguir los siguientes pasos:

9. Diseñar un formulario *Form2* con: tres etiquetas, una caja de texto para el nombre del país, un control **TextBox** para la diferencia horaria, una casilla de

verificación para el signo de la diferencia horaria (positivo sin marcar, negativo marcada) y dos botones: *Aceptar* y *Cancelar*.

10. Añadir una clase *Pais* cuyos objetos se puedan seriar, con los atributos *m_Pais* de tipo **String**, *m_DifHoraria* de tipo **long** (la diferencia horaria la almacenaremos en *ticks* o pasos; 1 paso = 100 nanosegundos).
11. Añadir a *Form1* un atributo privado que se corresponda con una colección **List** de elementos *Pais* denominada *ListaPaises* y otro *DiferenciaHoraria* de tipo **long** que almacenará la diferencia horaria con respecto a nuestro país del país del cual se esté mostrando su hora local.
12. Implementar un controlador vinculado con la orden *Añadir* para que visualice el diálogo *Añadir país* y añada los datos correspondientes a *ListaPaises* y a los menús *País* y contextual.
13. Implementar un controlador vinculado con la orden *Eliminar* para que visualice el diálogo *Borrar país* y borre el país seleccionado de *ListaPaises* y de los menús *País* y contextual.
14. Implementar el controlador vinculado con la orden correspondiente al país seleccionado del menú *País* o del menú contextual, para que la etiqueta y la caja de texto correspondientes muestren el país y la hora en el mismo (*Hora en*).
15. El reloj deberá presentar la hora actual en nuestro país en *ctHora* y la hora en otro país en *ctHoraPais*. Cuando no se haya seleccionado otro país ambas cajas mostrarán la misma hora.
16. Si existe un fichero predeterminado que almacene países y sus diferencias horarias, por ejemplo, "DifsHorsPredeterminado.bin", cuando se inicie la aplicación (evento **Loaded**) hay que añadir esos datos a los menús *País* y contextual y al objeto **List**.
17. Cuando se cierre la aplicación (evento **FormClosing**) hay que guardar los datos almacenados en el objeto **List** en un fichero predeterminado (por ejemplo, en "DifsHorsPredeterminado.bin").
18. Añadir el menú *Archivo* con las órdenes *Abrir* y *Guardar* y escribir sus controladores. Éstos deben permitir abrir o guardar cualquier otro fichero análogo a "DifsHorsPredeterminado.bin" (por ejemplo, "DifsHorsEuropa.bin").
19. Añada el controlador de la orden *Acerca de...*, del menú *Ayuda*, para que muestre un diálogo con los créditos de la aplicación.

CAPÍTULO 5

© F.J.Ceballos/RA-MA

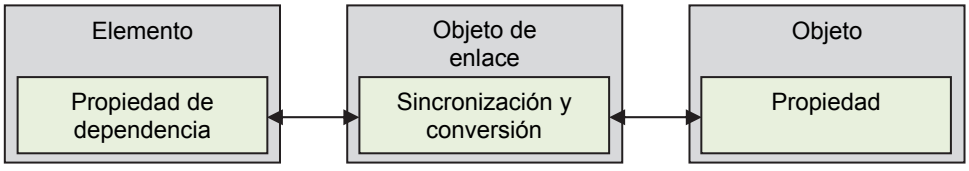
ENLACE DE DATOS EN WPF

Con lo que hemos aprendido en los capítulos anteriores, posiblemente ya hayamos llegado a la conclusión de que WPF facilita el diseño de interfaces de usuario sólidas y que también proporciona unas capacidades muy eficaces de enlace de datos. Esto es, con WPF, podemos realizar la manipulación de datos mediante código de un lenguaje .Net, por ejemplo Visual Basic, XAML o una combinación de ambos, y podemos también realizar el enlace de datos a controles, a propiedades públicas, a XML y a objetos, convirtiendo las operaciones de enlace de datos en tareas rápidas, flexibles y realmente fáciles. Así pues, en este capítulo vamos a profundizar en cómo podemos enlazar los controles a diversos orígenes de datos.

ASPECTOS BÁSICOS

El enlace de datos es un proceso que establece una conexión entre la interfaz gráfica del usuario (IGU) de la aplicación y la lógica de negocio, para que cuando los datos cambien su valor, los elementos de la IGU que estén enlazados a ellos reflejen los cambios automáticamente.

Los aspectos básicos del enlace de datos fueron ya expuestos en el capítulo *Introducción a WPF*. En ese capítulo dijimos que para utilizar el enlace de datos de WPF, debe disponer siempre de un destino y un origen. Como se muestra en la figura siguiente, el objeto enlace de datos es, en esencia, un puente entre el destino del enlace (*elemento*) y el origen del mismo (*objeto*).



El destino del enlace puede ser cualquier elemento o propiedad accesible que se derive de **DependencyProperty**, un ejemplo es la propiedad **Text** del control **TextBox**. El origen del enlace puede ser cualquier propiedad **Public**, incluidas propiedades de otros controles, objetos del CLR, elementos XAML, **DataSets** de ADO.NET, fragmentos XML, etc. Para facilitar la conexión con estas fuentes de datos, WPF proporciona dos proveedores especiales: **XmlDataProvider**, que ya hemos utilizado en algunos ejemplos del capítulo anterior, y **ObjectDataProvider**, que estudiaremos en este capítulo.

Como se mencionó en el capítulo *Introducción a WPF*, el flujo de datos de un enlace puede ir desde el destino al origen (por ejemplo, el valor del origen cambia cuando un usuario modifica el valor del **TextBox** enlazado), desde el origen al destino si el origen del enlace proporciona las notificaciones correspondientes (por ejemplo, el contenido del **TextBox** se actualiza con los cambios en el origen del enlace), o en ambas direcciones.

Un enlace de datos en WPF viene definido por un objeto de la clase **Binding**. La ruta de acceso a los datos se especifica por medio de su propiedad **Path** (o **XPath** si los datos están en XML), la dirección del flujo de datos queda especificada por su propiedad **Mode**, **UpdateSourceTrigger** especifica cuándo ocurre la actualización del origen (por ejemplo, un valor *PropertyChanged* para esta propiedad indica que el origen del enlace se actualizará en cuanto cambie la propiedad de destino), **ElementName** indica el nombre del origen de datos cuando éste es otro elemento de la interfaz y **Source** especifica, de forma explícita, el objeto que se va a utilizar como origen de los datos, invalidando el contexto de datos predeterminado, que es definido implícitamente por la propiedad **DataContext** del elemento seleccionado de la IGU.

El aspecto más importante de un contexto de datos es hacer referencia al origen de datos que se utiliza para el enlace y permitir a los elementos secundarios del árbol de elementos que define la interfaz gráfica heredar esa información cuando haya sido definida en un elemento primario. Por ejemplo, en el código siguiente, el contexto de datos del **Grid** hace referencia al objeto origen de datos, *objGrados*, que podrá ser utilizado por todos sus elementos secundarios.

```
<Window ...
        xmlns:local="clr-namespace:ConverTemps.Clases" >
  <Window.Resources>
    <local:CGrados x:Key="objGrados" GradosC="0.00" GradosF="32.00" />
  </Window.Resources>
  <Grid Name="gridVentPpal"
        DataContext="{StaticResource objGrados}">
    ...
  </Grid>
</Window>
```

Y este otro código que mostramos a continuación enlaza la propiedad **Text** de una caja de texto con la propiedad *GradosF* del objeto origen de datos *objGrados*, y la propiedad **Foreground** de esa caja de texto con la propiedad **Foreground** de otro elemento denominado *ctGradosC* de la interfaz gráfica de usuario.

```
<TextBox Name="ctGradosF"...
  Text="{Binding Path=GradosF,...}"
  Foreground="{Binding Path=Foreground, ElementName=ctGradosC}"
... />
```

Para que el elemento de la IGU vinculado con el objeto origen de datos pueda actualizarse automáticamente cuando éste cambie, dicho objeto origen de datos debe implementar la interfaz **INotifyPropertyChanged**, para notificar los cambios cuando éstos sucedan. Precisamente, el objeto de enlace (**Binding**) está siempre a la escucha de estas notificaciones, para comunicar al destino cuándo debe actualizarse (para más detalles, eche una ojeada al capítulo *Introducción a WPF*).

ENLACE A COLECCIONES DE OBJETOS

El origen de un enlace puede ser un objeto único, cuyas propiedades contienen los datos, o una colección de objetos resultado, por ejemplo, de una consulta a una base de datos. Hasta ahora hemos explicado el enlace a objetos individuales e indirectamente, hemos trabajado también con colecciones de objetos cuando los datos han sido obtenidos de un fichero XML (por ejemplo *alumnos.xml*). Como veremos a continuación, es bastante habitual utilizar un **ItemsControl**, como por ejemplo **ListBox**, **ListView** o **TreeView**, para mostrar una colección de objetos.

Según vimos en el capítulo anterior, para enlazar un **ItemsControl** a una colección de objetos, la propiedad que se utiliza es **ItemsSource**. Esto es, se puede considerar la propiedad **ItemsSource** como el contenido del **ItemsControl** (esta propiedad admite el enlace *OneWay*, desde el origen al destino, de forma predeterminada).

Cómo implementar colecciones

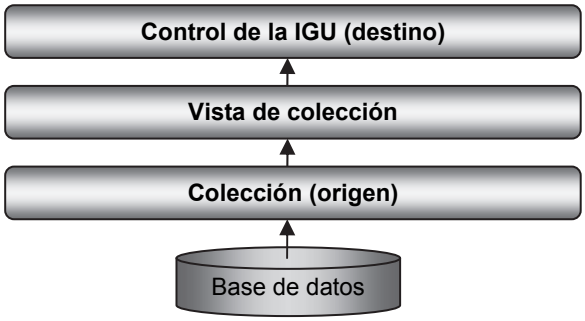
Como sabemos, para configurar enlaces dinámicos que actualicen la interfaz gráfica del usuario de forma automática cuando en la colección origen del enlace se realicen cambios, dicha colección debe implementar la interfaz **INotifyCollectionChanged**. Pues bien, la clase **ObservableCollection(Of** *T***)** de WPF proporciona una colección que implementa esta interfaz. Evidentemente, cada objeto de la colección que publica propiedades enlazables debe implementar también esa interfaz. Otras clases de colección existentes, que también podemos utilizar, son

List(Of *T*), **Collection(Of *T*)** y **BindingList(Of *T*)**, entre otras. Sirva como ejemplo el apartado *ICommand versus Routed Command* del capítulo *Menús y barras de herramientas*, en el que modelamos una base de datos utilizando la colección **List(Of *T*)**.

Vistas de colección

Una vez que un **ItemsControl** está enlazado a una colección de objetos, podemos navegar, filtrar, ordenar o agrupar los datos. Para ello, se utilizan vistas de colección, que son clases que implementan la interfaz **ICollectionView**.

Una *vista de colección* es un objeto situado un nivel por encima de la colección proporcionada por el origen del enlace. Dicha vista permite navegar y mostrar la colección en función de las consultas de ordenación, filtrado y agrupación, sin tener que cambiar la propia colección subyacente en el origen.



Los cambios en la colección se propagarán a las vistas si la colección implementa la interfaz **INotifyCollectionChanged**, pero las vistas no cambian la colección subyacente; por eso, cada colección proporcionada por el origen puede tener varias vistas asociadas. Esto quiere decir que el uso de vistas permitirá mostrar los mismos datos de formas diferentes.

Cada colección proporcionada por el origen del enlace tiene una vista predeterminada. ¿Cómo se obtiene dicha vista? Para crear la vista, se necesita una referencia a la colección subyacente en el origen, que se puede obtener directamente, a través del contexto de datos, de una propiedad del origen u obteniendo una propiedad del enlace. Por ejemplo, supongamos que hemos establecido en el contexto de datos de un **StackPanel** una referencia a un proveedor de datos, *datosAlumnos*, que es una colección **ObservableCollection(Of *T*)** de objetos *alumno*:

```
<StackPanel.DataContext Name="spAlumnos" >
  <Binding Source="{StaticResource datosAlumnos}"/>
</StackPanel.DataContext>
```

En este caso, para obtener la vista de la colección subyacente en el origen utilizaremos la propiedad **DataContext** del control **StackPanel** así:

```
Dim vista As ICollectionView =
  CollectionViewSource.GetDefaultView(spAlumnos.DataContext)
```

La clase **CollectionView** representa una vista para navegar, filtrar, ordenar y agrupar datos en una colección y **CollectionViewSource** es un *proxy* (interfaz de comunicación) para una clase que implemente **ICollectionView**, como ocurre con **CollectionView**, que permite al código comunicarse con la vista. **CollectionViewSource** tiene una propiedad **View** que hace referencia a la vista real y una propiedad **Source** que hace referencia a la colección proporcionada por el origen. Desde el punto de vista de código XAML, la clase **CollectionViewSource** es la representación en XAML de la clase **CollectionView**. Las operaciones que realicemos sobre la vista se reflejarán automáticamente en la ventana. Por ejemplo, el código siguiente obtiene la vista de una colección. Si suponemos que esta colección está enlazada a un **ListBox**, el siguiente código, actuando sobre la vista, desplazaría el cursor al siguiente elemento cada vez que se ejecute:

```
Dim vista As ICollectionView = ObtenerVista()
vista.MoveCurrentToNext()
```

Este desplazamiento se reflejaría automáticamente sobre el **ListBox**. En este caso, si además deseamos que la lista se desplace para mostrar el elemento actualmente seleccionado, tendríamos que añadir una línea como la siguiente:

```
lista.ScrollIntoView(vista.CurrentItem)
```

Este otro ejemplo que se expone a continuación es un método que obtiene la vista de la colección de objetos proporcionada por el proveedor de datos XML *xmlAlumnos* que hemos venido utilizando en los ejemplos de capítulos anteriores:

```
Private Function ObtenerVista() As ICollectionView
  Dim proveedor As DataSourceProvider =
    DirectCast(Me.FindResource("xmlAlumnos"), DataSourceProvider)
  Return CollectionViewSource.GetDefaultView(proveedor.Data)
End Function
```

El tipo de vista utilizada dependerá del tipo del objeto origen de los datos. Todas las vistas derivan de **CollectionView** y concretamente, estas dos implementaciones especializadas: **ListCollectionView** y **BindingListCollectionView**. ¿Cuándo se crea una u otra?

- Si el origen de datos implementa la interfaz **IBindingList**, se crea una colección **BindingListCollectionView**; esto sucede cuando el enlace es con un **DataTable** de ADO.NET.

- Si el origen de datos no implementa **IBindingList** pero implementa la interfaz **IList**, entonces se crea una colección **ListCollectionView**. Esto sucede cuando el enlace es con un **ObservableCollection**. En los dos ejemplos anteriores se devuelve una colección **ListCollectionView**.

- Y si el origen de datos no implementa ni **IBindingList** ni **IList** pero implementa la interfaz **IEnumerable**, entonces se crea una colección **CollectionView**.

No debería crear objetos de la clase **CollectionView**. Para crear una vista de colección para una colección que sólo implementa **IEnumerable**, cree un objeto **CollectionViewSource**, agregue la colección a su propiedad **Source** y obtenga la vista de colección de su propiedad **View**.

Desde un punto de vista práctico, aplicaremos lo aquí explicado al hablar de XML y de colecciones de objetos como fuentes de datos. Antes vamos a profundizar un poco más en las plantillas de datos.

PLANTILLAS DE DATOS

Según hemos podido comprobar en los capítulos anteriores, el modelo de plantillas de datos de WPF proporciona gran flexibilidad para definir la presentación de los datos. Por ello, en este apartado vamos a profundizar un poco más en el tema viendo primero cómo definir un objeto **DataTemplate** para, a continuación, estudiar otras características de las plantillas de datos.

Un objeto **DataTemplate** permite especificar cómo se presentarán los objetos de datos, proceso que resulta particularmente útil al enlazar un control **ItemsControl**, por ejemplo un **ListBox**, a una colección completa de objetos. Como ejemplo, vamos a diseñar una aplicación que muestre una ventana (un diálogo) que presente un objeto **ListBox** enlazado a una lista de objetos *alumno* almacenados en un fichero XML. Trabajaremos con el mismo fichero *alumnos.xml* que utilizamos en el capítulo anterior:

```
<?xml version="1.0" encoding="utf-8" ?>
<alumnos>
  <alumno>
    <foto>Imagenes/foto.jpg</foto>
    <nombre>Alumno 01</nombre>
    <direccion>Dirección 01</direccion>
```

```
    <estudios>Estudios 01</estudios>
    <beca>True</beca>
    ...
  </alumno>
</alumnos>
```

Suponiendo que ya ha creado una nueva aplicación, por ejemplo *EnlaceDeDatosXML*, con una barra de menús con las órdenes *Archivo*, *Diálogos* y *Ayuda*, añada a la sección de recursos del objeto **Application** un proveedor XML que haga referencia al fichero anterior:

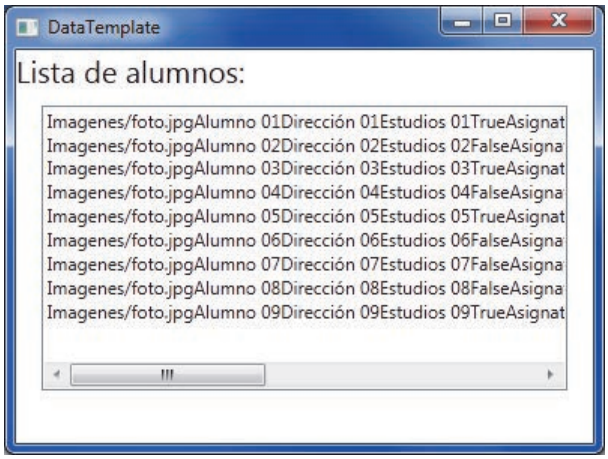
```
<Application.Resources>
  <XmlDataProvider x:Key="xmlAlumnos"
                   Source="datos/alumnos.xml" XPath="alumnos/alumno"/>
</Application.Resources>
```

El valor *alumnos/alumno* del atributo **XPath** indica que se seleccionarán todos los elementos *alumno* que son hijos de *alumnos*.

A continuación añada una nueva ventana *DlgDataTemplate* que muestre un **ListBox**, según el diseño especificado a continuación, y un elemento al menú *Diálogos* de la ventana principal que permita visualizar esta ventana (para más detalles sobre el trabajo con ventanas, véase el capítulo *Controles y cajas de diálogo*):

```
<Window x:Class="DlgDataTemplate" ...
        Title="DlgDataTemplate" Height="300" Width="400">
  <Grid>
    <StackPanel>
      <TextBlock Name="tbTitulo" FontSize="20"
                 Text="Lista de alumnos:"/>
      <ListBox Height="260" Width="400" Margin="10"
        ItemsSource="{Binding Source={StaticResource xmlAlumnos}}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

Observe que la propiedad **ItemsSource** del **ListBox** está enlazada con la fuente de datos proporcionada por el proveedor XML. Esta fuente de datos se corresponde con todos los objetos *alumno* de *alumnos*, según especifica el atributo **XPath** del proveedor de datos. Ejecute ahora la aplicación. Observará que la lista no muestra el resultado esperado:



¿Por qué el resultado anterior? Un control **ListBox** cuyos elementos sean objetos de una colección, sin instrucciones concretas acerca de cómo se deben presentar esos elementos, muestra la representación de cadena de los objetos (como estamos trabajando con elementos XML, la cadena se corresponde con toda la información que incluye un alumno; con clases de objetos CLR la información sería la dada por el método **ToString** de la clase de esos objetos). ¿Cuál es la solución? Pues utilizar un objeto **DataTemplate** para definir cómo deben mostrarse los objetos de datos; esto es, el contenido de **DataTemplate** se convierte en la estructura visual de los objetos de datos.

Definir una plantilla de datos

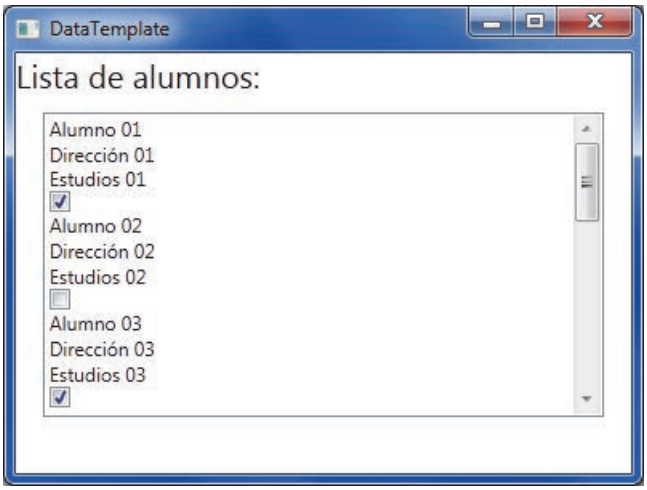
Una forma de definir una plantilla para un control lista es: definir un objeto **DataTemplate** y asignarlo a la propiedad **ItemTemplate** del objeto **ListBox**. El objeto **DataTemplate** definirá los controles de datos adecuados al tipo de los datos a mostrar; según esto, para mostrar los datos *nombre*, *dirección* y *estudios* podemos utilizar elementos **TextBlock** y para el dato *beca*, un elemento **CheckBox**, que podemos colocar dentro de un objeto **StackPanel**. Después, enlazamos cada uno de ellos con el elemento correspondiente del origen de datos.

```
<ListBox Height="210" Width="350" Margin="10"
       ItemsSource="{Binding Source={StaticResource xmlAlumnos}}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding XPath=nombre}" />
        <TextBlock Text="{Binding XPath=direccion}"/>
        <TextBlock Text="{Binding XPath=estudios}"/>
        <CheckBox IsChecked="{Binding XPath=beca}"/>
      </StackPanel>
```

```
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

En general, tenga en cuenta que el objeto **DataTemplate** se aplica a cada uno de los elementos **ListBoxItem** generados en la lista.

Ahora podrá observar que la lista toma el aspecto esperado:



La plantilla puede definirse también como un recurso de la ventana y asignar después ese recurso a la propiedad **ItemTemplate** del **ListBox**. La ventaja de proceder así es que podríamos utilizar este recurso en otros controles, evidentemente siempre y cuando tenga sentido hacerlo. Por ejemplo:

```
<Window x:Class="DlgDataTemplate" ...
        Title="DlgDataTemplate" Height="300" Width="400">
  <Window.Resources>
      <DataTemplate x:Key="plantillaAlumno">
        <StackPanel>
          <TextBlock Text="{Binding XPath=nombre}" />
          <TextBlock Text="{Binding XPath=direccion}"/>
          <TextBlock Text="{Binding XPath=estudios}"/>
          <CheckBox IsChecked="{Binding XPath=beca}" />
        </StackPanel>
      </DataTemplate>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <TextBlock Name="tbTitulo" FontSize="20"
                 Text="Lista de alumnos:"/>
      <ListBox Height="190" Width="350" Margin="10"
```

```
        ItemsSource="{Binding Source={StaticResource xmlAlumnos}}"
        ItemTemplate="{StaticResource plantillaAlumno}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

Mejorar la presentación

Podemos mejorar la presentación agregando un **Border**, elementos **TextBlock** que describan los datos que se muestran y, en lugar del elemento **StackPanel**, un **Grid** con cuatro filas (tantas como datos mostrados) y dos columnas: la primera para las descripciones y la segunda para los datos.

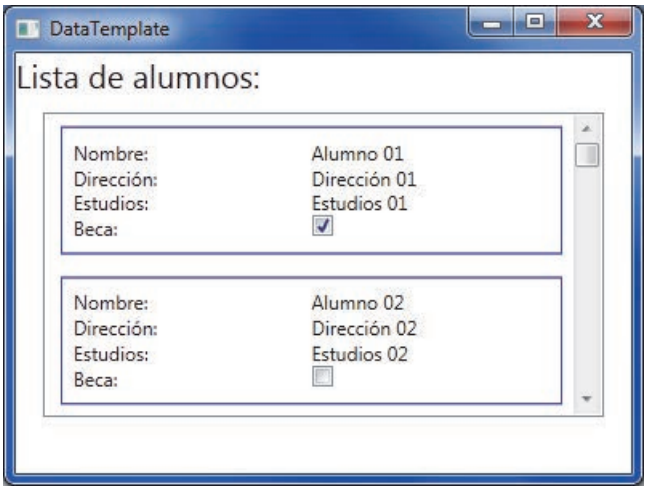
```
<DataTemplate x:Key="plantillaAlumno">
  <Border Name="bordeAlumno" BorderBrush="Blue" BorderThickness="1"
          Padding="7" Margin="7">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Text="Nombre: " />
      <TextBlock Grid.Row="0" Grid.Column="1"
                 Text="{Binding XPath=nombre}" />
      <TextBlock Grid.Row="1" Grid.Column="0" Text="Dirección: " />
      <TextBlock Grid.Row="1" Grid.Column="1"
                 Text="{Binding XPath=direccion}"/>
      <TextBlock Grid.Row="2" Grid.Column="0" Text="Estudios: " />
      <TextBlock Grid.Row="2" Grid.Column="1"
                 Text="{Binding XPath=estudios}"/>
      <TextBlock Grid.Row="3" Grid.Column="0" Text="Beca: " />
      <CheckBox Grid.Row="3" Grid.Column="1"
                IsChecked="{Binding XPath=beca}" />
    </Grid>
  </Border>
</DataTemplate>
```

Para que los elementos de la lista ocupen todo el ancho del **ListBox**, asigne a la propiedad **HorizontalContentAlignment** del **ListBox** el valor *Stretch*:

```
<ListBox Height="190" Width="350" Margin="10"
     ItemsSource="{Binding Source={StaticResource xmlAlumnos}}"
```

```
      ItemTemplate="{StaticResource plantillaAlumno}"
      HorizontalContentAlignment="Stretch">
</ListBox>
```

Ahora, el control **ListBox** mostrará los datos así:



Siguiendo con la mejora de la presentación, podríamos pintar el borde de los alumnos con beca en otro color, por ejemplo en verde. Esto lo podemos hacer utilizando desencadenadores (**DataTrigger**).

Utilizar desencadenadores para aplicar valores de propiedad

En el código siguiente, el objeto **DataTrigger** de la colección **Triggers** del **DataTemplate** establece el valor de la propiedad **BorderBrush** del elemento denominado *bordeAlumno* en *LightGreen* si el atributo *beca* del elemento *alumno* del origen de datos mostrado vale *True*.

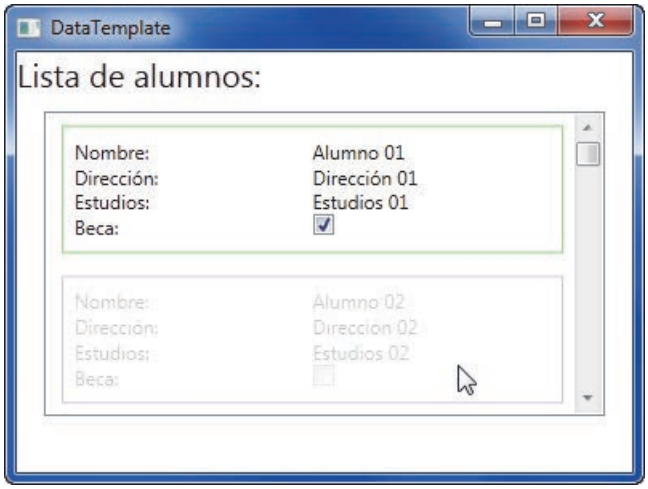
```
<DataTemplate x:Key="plantillaAlumno">
  <Border Name="bordeAlumno" BorderBrush="Blue" BorderThickness="1"
    ...
  </Border>
  <DataTemplate.Triggers>
    <DataTrigger Binding="{Binding XPath=beca}" Value="True">
      <Setter TargetName="bordeAlumno"
              Property="BorderBrush" Value="LightGreen"/>
    </DataTrigger>
  </DataTemplate.Triggers>
</DataTemplate>
```

Otra forma de hacer esto mismo es utilizando un conversor, según vimos en un ejemplo en el capítulo *Introducción a WPF*.

En el ejemplo anterior, el objeto **Setter** del desencadenador establece el valor de una propiedad de un elemento (**Border** en nuestro caso) que se encuentra dentro del **DataTemplate**. Sin embargo, si las propiedades afectadas por **Setter** no corresponden a elementos que estén dentro del **DataTemplate** actual, puede que sea más conveniente establecer las propiedades mediante un objeto **Style** para la clase del elemento al que se desea aplicar el estilo (**ListBoxItem** en nuestro caso). Por ejemplo, si deseamos animar el valor **Opacity** del elemento **ListBoxItem** cuando el ratón señale a un elemento, puede definir en los recursos de la ventana el siguiente estilo:

```
<Style TargetType="ListBoxItem">
  <Style.Triggers>
    <EventTrigger RoutedEvent="ListBoxItem.MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Storyboard.TargetProperty="(ListBoxItem.Opacity)"
            From="1" To="0.1" Duration="0:0:3" AutoReverse="True"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Style.Triggers>
</Style>
```

El objeto **Storyboard** define una escala de tiempo para realizar una animación y **DoubleAnimation** anima el valor de una propiedad **Double** entre dos valores usando la interpolación lineal durante 3 segundos, propiedad **Duration** especificada; **AutoReverse** indica si la escala de tiempo retrocede después de completar una iteración de avance.



XML COMO FUENTE DE DATOS

El proveedor **XmlDataProvider** proporciona una forma rápida y sencilla de extraer datos XML de un fichero, de una dirección web o de los recursos de la aplicación y ponerlo a disposición de los elementos de una aplicación. Este proveedor ha sido diseñado para ser de sólo lectura y la forma de utilizarlo es muy sencilla:

```
<XmlDataProvider x:Key="xmlAlumnos"
                 Source="datos/alumnos.xml" XPath="/alumnos/alumno"/>
```

Source es una propiedad de tipo **Uri** (identificador de recursos uniforme); esta propiedad permite obtener o establecer el objeto **Uri** correspondiente al nombre del fichero de datos XML que se va a usar como origen del enlace, y **XPath** es una propiedad de tipo **String** que hace referencia a la ruta de acceso a los datos sobre la que se realizará la consulta para obtener el conjunto de datos especificado. Esta propiedad **Source** también se puede establecer desde el código subyacente, por ejemplo, si aún no conocemos el fichero XML que necesitamos usar.

El proveedor **XmlDataProvider** carga el contenido XML asíncronamente, excepto si asignamos a su propiedad **IsAsynchronous** el valor **False**.

En los ejemplos realizados anteriormente tomando como fuente de datos un fichero XML, hemos podido comprobar que .Net proporciona una amplia biblioteca para trabajar con este tipo de información (veremos algún ejemplo un poco más adelante). Así que cuando necesitemos la capacidad de modificar XML o convertir los datos XML en objetos para realizar una programación orientada a objetos, es mejor olvidarnos del proveedor de datos y utilizar el soporte que proporciona .Net.

Como ejemplo, volvamos a la aplicación anterior, a la ventana *DlgDataTemplate*, para añadir a los elementos mostrados por la lista la información restante, esto es, las asignaturas de las que está matriculado cada alumno, información de la cual mostramos un ejemplo a continuación:

```
<?xml version="1.0" encoding="utf-8" ?>
<alumnos>
  <alumno>
    <foto>Imagenes/foto.jpg</foto>
    <nombre>Alumno 01</nombre>
    <direccion>Dirección 01</direccion>
    <estudios>Estudios 01</estudios>
    <beca>True</beca>
    <asignaturas tipo="Obligatorias">
      <asignatura>
        <nombre>Asignatura ob01</nombre>
        <nota>8,5</nota>
```

```
      </asignatura>
      <asignatura>
        <nombre>Asignatura ob02</nombre>
        <nota>5,5</nota>
      </asignatura>
      <asignatura>
        <nombre>Asignatura ob03</nombre>
        <nota>7,0</nota>
      </asignatura>
    </asignaturas>
    <asignaturas tipo="Optativas">
      <asignatura>
        <nombre>Asignatura op01</nombre>
        <nota>6,0</nota>
      </asignatura>
      <asignatura>
        <nombre>Asignatura op02</nombre>
        <nota>4,5</nota>
      </asignatura>
    </asignaturas>
  </alumno>
  ...
</alumnos>
```

La colección de datos presentada está organizada jerárquicamente, por lo tanto, para mostrarla lo más fácil será utilizar controles que admitan este tipo de organización, según veremos a continuación.

Datos jerárquicos

En ocasiones tendremos colecciones que contengan otras colecciones. Por ejemplo, en la fuente de datos que acabamos de presentar en el apartado anterior, un alumno tiene una colección de asignaturas. El tratamiento de datos jerárquicos puede resultar sencillo si se utiliza la plantilla **HierarchicalDataTemplate**, que representa un **DataTemplate** que admite controles **HeaderedItemsControl**, como **TreeViewItem** o **MenuItem**.

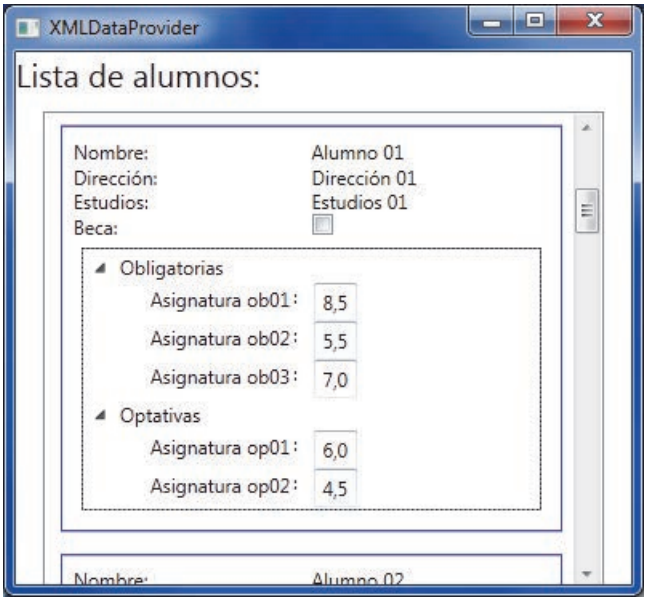
Según esto, para añadir a cada elemento de la lista las asignaturas de las que está matriculado cada alumno, utilizaremos un **TreeView**. Para ello, vamos a modificar la plantilla de la lista, *plantillaAlumno*, para añadir este nuevo elemento a continuación de los existentes, según se puede ver en el código XAML siguiente:

```
<DataTemplate x:Key="plantillaAlumno">
  <Border Name="bordeAlumno" ...>
    <Grid>
      ...
```

```
      <TreeView Grid.Row="4" Grid.ColumnSpan="2"
                Name="trvAlumnos" Margin="5"
              ItemsSource="{Binding XPath=asignaturas}"
              ItemTemplate="{StaticResource kAsignaturas}">
      </TreeView>
    </Grid>
  </Border>
  ...
</DataTemplate>
```

El origen de datos para **TreeView**, propiedad **ItemsSource**, es proporcionado por un enlace que indica por medio de su atributo **XPath** dónde se encuentra la colección que representa el nivel siguiente en la jerarquía de datos (en el ejemplo, se encuentra en *asignaturas*), y la plantilla que indica cómo se va a mostrar cada elemento de esa colección queda especificada por la propiedad **ItemTemplate** (en el ejemplo, se mostrarán según indique el recurso *kAsignaturas*).

El resultado que obtendremos, después de implementar el recurso *kAsignaturas* será análogo al siguiente (para facilitar el estudio, implementaremos este ejemplo en otro diálogo *DlgXmlDataProvider* como el mostrado a continuación):



El recurso *kAsignaturas* también es de tipo **HierarchicalDataTemplate**, ya que un elemento *asignaturas* contiene elementos *asignatura*. Cada uno de estos elementos irá encabezado por el **TextBlock** "Obligatorias" u "Optativas". Este valor lo proporciona el atributo *tipo* del elemento *asignaturas*.

```
<!-- PLANTILLA ASIGNATURAS -->
```

```
<HierarchicalDataTemplate x:Key="kAsignaturas"
                  ItemsSource="{Binding XPath=asignatura}"
                  ItemTemplate="{StaticResource kAsignatura}">
  <TextBlock Text="{Binding XPath=@tipo}" Padding="2" />
</HierarchicalDataTemplate>
```

Observe la sintaxis de **XPath**, cuando el identificador del dato asociado es un atributo de un elemento XML, por ejemplo *XPath=@tipo* va precedido por @, no sucediendo esto cuando el identificador del dato se corresponde con un elemento XML, por ejemplo *XPath=asignatura*.

Ahora, el origen de datos para esta plantilla es *asignatura*, elemento donde se encuentran los datos para el siguiente nivel, y la plantilla que indica cómo se van mostrar estará definida por el recurso *kAsignatura*.

El recurso *kAsignatura* será de tipo **DataTemplate** (no jerárquico) por tratarse de nodos terminales del árbol. Esto es, un elemento *asignatura* contiene elementos *nombre* y *nota*, que son datos, no colecciones de datos.

```
<!-- PLANTILLA ASIGNATURA -->
<DataTemplate x:Key="kAsignatura">
  <WrapPanel>
    <TextBlock Text="{Binding XPath=nombre}" Padding="2" />
    <TextBlock Text=":   " />
    <TextBox Text="{Binding XPath=nota}" Padding="2" />
  </WrapPanel>
</DataTemplate>
```

En este caso, la plantilla define los controles utilizados para mostrar los datos. La propiedad que se utilice de estos controles para mostrar el dato irá enlazada al elemento XML correspondiente.

Otras expresiones **XPath** que hemos utilizado o que son muy comunes:

`/alumno`. Selecciona el elemento raíz *alumno*.

`alumno`. Selecciona todos los nodos hijo del elemento *alumno*.

`. (punto)`. Selecciona el nodo actual.

`.. (dos puntos)`. Selecciona el padre del nodo actual.

`@`. Selecciona atributos de un elemento.

`asignaturas/asignatura`. Selecciona todos los elementos *asignatura* que son hijos de *asignaturas*.

`/asignaturas/*`. Selecciona todos los nodos hijo del elemento *asignaturas*.

`asignaturas[1]`. Selecciona todos los nodos hijo del primer elemento *asignaturas*.

`asignaturas[last()]`. Selecciona todos los nodos hijo del último elemento *asignaturas*.

Para más detalles sobre la sintaxis de **XPath**, puede visitar la dirección *http://www.w3schools.com/xpath*.

Islas de datos

Una isla de datos XML es como un objeto secundario de proveedor de datos XML que permite insertar un documento XML directamente en el código XAML de la ventana. Por ejemplo:

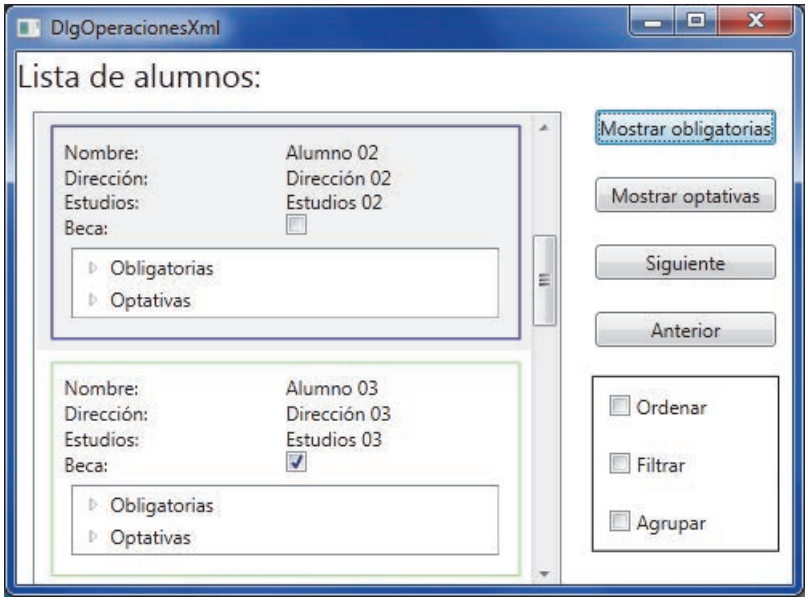
```
<Window x:Class="DlgDataTemplate"
        ...
        Title="DlgDataTemplate" Height="370" Width="400">
  <Window.Resources>
    <XmlDataProvider x:Key="xmlAlumnos" XPath="/alumnos/alumno">
      <x:XData>
        <alumnos xmlns="">
          <alumno>
            ...
          </alumno>
          ...
        </alumnos>
      </x:XData>
    </XmlDataProvider>
    ...
  </Window.Resources>
  <Grid>
    ...
  </Grid>
</Window>
```

En este ejemplo podemos observar cómo el objeto **x:XData** se utiliza como un objeto secundario de **XmlDataProvider**.

Soporte .Net para trabajar con XML

Además de la clase **XmlDataProvider**, la biblioteca .Net proporciona el espacio de nombres **System.Xml** que contiene clases diseñadas para trabajar con datos XML. Veamos entonces algunas operaciones que podemos realizar sobre el documento XML mostrado por la ventana de nuestro ejemplo (para facilitar el estu-

dio, implementaremos este ejemplo en otro diálogo *DlgOperacionesXml* según muestra la figura siguiente).



Por otra parte, tengamos en cuenta que, según explicamos anteriormente en el apartado *Vistas de colección*, la clase **CollectionView** representa una vista para navegar, filtrar, ordenar y agrupar datos en una colección. Por lo tanto, tendremos que obtener esta vista para poder trabajar con la colección de objetos del documento XML.

Continuando con el ejemplo, partiendo de los diálogos ya construidos, diseñe el diálogo de la figura anterior y añada los controladores de eventos para controlar el evento **Click** de cada uno de los botones de la interfaz gráfica, incluidas las casillas de verificación. En los siguientes apartados explicaremos cómo escribir cada uno de estos controladores.

En este diseño, establezca la propiedad **IsSynchronizedWithCurrentItem** de la lista (objeto **ListBox**) a **True**:

```
<ListBox Name="listaAlumnos" Height="300" Width="330" Margin="10"
         ItemsSource="{Binding Source={StaticResource xmlAlumnos}}"
         ItemTemplate="{StaticResource plantillaAlumno}"
         HorizontalContentAlignment="Stretch"
         IsSynchronizedWithCurrentItem="True">
```

Esto nos asegura que el elemento seleccionado siempre se corresponda con la propiedad **CurrentItem** de la colección **ItemCollection** del **ListBox**. De esta

forma, la propiedad **SelectedItem** siempre estará sincronizada con el elemento actual de **ItemCollection**; si fuera **False**, no estaría sincronizada y podría darse el caso de que para dos controles **ListBox** con el mismo origen de datos, el elemento seleccionado en cada **ListBox** no fuera el mismo. Éste no es nuestro caso, pero como veremos cuando implementemos la navegación, con la sincronización conseguiremos que, además de que **CurrentItem** y **SelectedItem** estén sincronizados, el elemento actualmente seleccionado aparezca diferenciado de los demás (con un color gris).

Obtener la vista

Cuando en WPF se realiza un enlace a una colección, en realidad ese enlace se hace con la vista predeterminada de esa colección; esto es así porque todas las colecciones tienen un objeto **CollectionView** predeterminado. Si hay varios enlaces a una colección, todos comparten su vista predeterminada, lo que hace que todos compartan las características de ordenación, filtrado, agrupación y elemento actual de esa vista.

Por otra parte, sabemos que la funcionalidad de la clase **CollectionViewSource** permite al código comunicarse con la vista. Por ejemplo, esta clase proporciona un método **Shared**, denominado **GetDefaultView**, que, como su nombre indica, devuelve la vista predeterminada de la colección especificada como argumento. Por ejemplo, para obtener la vista predeterminada de la colección de datos proporcionada por el proveedor XML de nuestra aplicación, podríamos añadir a la clase de la ventana (en este caso, a *DlgOperacionesXml*) el método siguiente:

```
Private Function ObtenerVista() As ICollectionView
  Dim proveedor As DataSourceProvider =
    DirectCast(Me.FindResource("xmlAlumnos"), DataSourceProvider)
  Return CollectionViewSource.GetDefaultView(proveedor.Data)
End Function
```

La clase **DataSourceProvider** es la clase base (se trata de una clase abstracta) de las clases **XmlDataProvider** y **ObjectDataProvider**. Y el método **GetDefaultView** devuelve un objeto vista que implementa la interfaz **ICollectionView**. En este caso, este objeto es de tipo **ListCollectionView**, según explicamos en el apartado *Vistas de colección* anteriormente en este mismo capítulo.

Elemento actual

Las vistas de colección tienen la capacidad de realizar el seguimiento del elemento actualmente seleccionado mediante un puntero de elemento. Dicho elemento estará siempre referenciado por la propiedad **CurrentItem** de la vista.

Como ejemplo, vamos a escribir el controlador del evento **Click** del botón "Mostrar obligatorias" para que muestre en una ventana de mensaje las asignaturas obligatorias de las que está matriculado el alumno actualmente seleccionado.

```
Private Sub btMostrarOb_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.IsEmpty Then Return
  ' Cada "alumno" es un XmlElement
  Dim alumnoActual As XmlElement =
                DirectCast(vista.CurrentItem, XmlElement)
  Dim list As XmlNodeList =
                alumnoActual.GetElementsByTagName("asignaturas")
  For Each asigs As XmlElement In list
    If asigs.GetAttribute("tipo") = "Obligatorias" Then
      list = asigs.GetElementsByTagName("asignatura")
      Dim s As New StringBuilder()
      For Each asig As XmlElement In list
        s.Append(asig.FirstChild.InnerText + vbLf)
      Next
      MessageBox.Show(s.ToString())
    End If
  Next
End Sub
```

Este método obtiene la vista de la colección de datos, obtiene el elemento actualmente seleccionado en la vista (un *alumno*) que es de tipo **XmlElement** (no olvide que los datos proceden de un fichero XML), desciende al nodo asignaturas obligatorias y extrae de éste todos los nombres de las asignaturas, para después visualizarlos. Observe que para obtener la lista de los nodos secundarios indicados para un determinado elemento, éste invoca a su método **GetElementsByTagName**, y que para obtener el valor de un atributo de un elemento, éste invoca a su método **GetAttribute**. Como cada elemento *asignatura* tiene, a su vez, dos hijos (*nombre* y *nota*), su propiedad **FirstChild** devuelve el primero (el *nombre*) y la propiedad **InnerText** de éste, su contenido.

Como ejercicio, puede escribir el controlador del evento **Click** del botón "Mostrar optativas" de forma análoga.

Navegar

Otra de las operaciones admitidas por la vista es la navegación por sus elementos. Para ello, la clase **CollectionView** proporciona una serie de métodos que permiten realizar un desplazamiento al primer elemento de la vista (**MoveCurrentToFirst**), al último (**MoveCurrentToLast**), al siguiente (**MoveCurrentToNext**), al anterior (**MoveCurrentToPrevious**) o a una posición determinada (**MoveCu-**

rrentToPosition). Estos métodos lo que hacen es establecer el elemento de la vista al que hacen referencia como elemento actual de la vista (**CurrentItem**).

Como ejemplo, vamos a escribir el controlador del evento **Click** del botón "Siguiente" para que, cada vez que el usuario haga clic sobre él, el puntero de elemento avance al siguiente elemento al actualmente seleccionado.

```
Private Sub btSiguiente_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToNext()
  If vista.IsCurrentAfterLast Then
    vista.MoveCurrentToLast()
  End If
  listaAlumnos.ScrollIntoView(vista.CurrentItem)
End Sub
```

Este método primero obtiene la vista de la colección de datos y después, para avanzar al siguiente elemento, la vista invoca a su método **MoveCurrentToNext**. Si el movimiento solicitado va más allá del último elemento, éste pasará a ser el elemento actual. Observe también que después de cada movimiento, el **ListBox** invoca a su método **ScrollIntoView** pasando como argumento el elemento actualmente seleccionado para forzar a la lista a realizar el desplazamiento requerido para que se muestre dicho elemento (dicho de otra forma, lo que este método hace es que el objeto se desplace en la vista).

Como ejercicio, puede escribir el controlador del evento **Click** del botón "Anterior" de forma análoga.

Ordenar

Otra de las operaciones admitidas por la vista es la ordenación de sus elementos (recuerde que la vista siempre se sitúa un nivel por encima de la colección). Pues bien, la forma más sencilla de ordenar los elementos que mostrará la vista es a través de su propiedad **SortDescriptions**. Esta propiedad da acceso a una colección de estructuras de tipo **SortDescription** que describe los criterios bajo los que se ordenan los elementos de la colección en la vista.

Como ejemplo, vamos a escribir el controlador del evento **Click** de la casilla de verificación "Ordenar" para que, si es posible, ordene los elementos de la vista primero por el valor *beca* descendentemente (así agrupamos todos los alumnos que tienen beca en el orden "True", "False") y después por el *nombre* del alumno ascendentemente.

```
Private Sub cvOrdenar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
```

```
  If vista.CanSort AndAlso cvOrdenar.IsChecked = True Then
    If vista.SortDescriptions.Count = 0 Then
      vista.SortDescriptions.Add(
        New SortDescription("beca", ListSortDirection.Descending))
      vista.SortDescriptions.Add(
        New SortDescription("nombre", ListSortDirection.Ascending))
    End If
  Else
    vista.SortDescriptions.Clear()
  End If
End Sub
```

Observe que este método alterna entre dos vistas: si la casilla de verificación está marcada, se presentará la vista ordenada y si no, se mostrará en su estado inicial. Esto último se consigue eliminando de la colección **SortDescriptions** todos los criterios de ordenación (método **Clear**).

Filtrar

Una vista también tiene la capacidad de filtrar los elementos provenientes de la colección subyacente. Esto es, se puede crear una vista con los elementos de la colección que cumplan unos determinados criterios. Para especificar tales condiciones necesitamos vincular la propiedad **Filter** de la vista con el método que se utilice para determinar qué elementos pertenecerán a la misma. Como esta propiedad es de tipo **Predicate(Of Object)** podemos implementar un método anónimo acorde al delegado **Predicate**: *Delegate Function Predicate(Of In T) (obj As T) As Boolean*.

Como ejemplo, vamos a escribir el controlador del evento **Click** de la casilla "Filtrar" para que la vista muestre sólo los alumnos que tienen beca.

```
Private Sub cvFiltrar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()

  If vista.CanFilter AndAlso cvFiltrar.IsChecked = True Then
    If vista.Filter Is Nothing Then
      vista.Filter = Function(elemento As Object) _
        DirectCast(elemento, XmlElement).ChildNodes(4).InnerText = "True"
    End If
  Else
    vista.Filter = Nothing
  End If
End Sub
```

Observe que la condición impuesta para que un *alumno* se incluya en la vista es que el valor del elemento *beca* sea **True**.

Agrupar

Otra característica que proporciona la vista es su capacidad para agrupar sus elementos según unos criterios. La forma de hacer esto es a través de su propiedad **GroupDescriptions**. Esta propiedad da acceso a una colección de estructuras de tipo **PropertyGroupDescription** que describe los criterios bajo los que se agruparán los elementos de la colección en la vista.

Como ejemplo, vamos a escribir el controlador del evento **Click** de la casilla de verificación "Agrupar" para que, si es posible, agrupe los elementos de la vista por el valor *beca*.

```
Private Sub cvAgrupar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CanGroup AndAlso cvAgrupar.IsChecked = True Then
    If vista.GroupDescriptions.Count = 0 Then
      ' Agrupar por beca
      vista.GroupDescriptions.Add(New PropertyGroupDescription("beca"))
    End If
  Else
    vista.GroupDescriptions.Clear()
  End If
End Sub
```

Observe que, igual que en los casos anteriores, este método alterna entre dos vistas: si la casilla de verificación está marcada, se presentará la vista agrupada y si no, se mostrará en su estado inicial.

Fuente de datos XML sin el proveedor

Supongamos que estamos realizando un desarrollo que, como en el ejemplo anterior, tiene un origen de datos XML, pero no permite utilizar el proveedor de datos XML. En un caso como éste, una alternativa al desarrollo que hemos realizado en el apartado anterior es cargar el fichero XML origen de los datos en un documento XML (objeto **XmlDocument**) y utilizar este documento como origen de datos.

Según lo expuesto veamos a continuación las modificaciones que tendríamos que hacer al desarrollo realizado en el apartado anterior (el resultado puede verlo en el diálogo *DlgSinXMLDataProvider*).

Cuando se inicie la aplicación y se muestre el diálogo *DlgSinXMLDataProvider*, desde el constructor de esta clase, creamos un documento **XmlDocument**, de nombre *xmlDoc*, y lo cargamos con la información del fichero *alumnos.xml*. Este documento XML será el origen de datos que utilizaremos.

A continuación asignamos al contexto de datos del **Grid** (propiedad **DataContext** de *gridAlumnos*) este origen de datos. Para ello, creamos un objeto **Binding** que actuará de enlace entre el **Grid**, que hemos denominado *gridAlumnos*, y el origen de datos *xmlDoc*.

Finalmente, modificamos el método *ObtenerVista* para que nos devuelva la vista del origen de datos definido por el contexto de datos del **Grid**.

```
Imports System.ComponentModel
Imports System.Xml
Imports System.Text

Public Class DlgSinXMLDataProvider
  Public Sub New()
    InitializeComponent()
    CargarXmlDoc()
  End Sub

  Private Sub CargarXmlDoc()
    ' Construir un XmlDocument
    Dim xmlDoc As New XmlDocument()
    xmlDoc.Load("../../Datos/alumnos.xml")
    ' Establecer el contexto de datos del Grid
    Dim enlace As New Binding()
    enlace.XPath = "/alumnos/alumno"
    enlace.Source = xmlDoc
    gridAlumnos.SetBinding(Grid.DataContextProperty, enlace)
  End Sub

  Private Function ObtenerVista() As ICollectionView
    Return CollectionViewSource.GetDefaultView(gridAlumnos.DataContext)
  End Function
  ...
End Class
```

Sólo queda enlazar el control **ListBox** con la colección de objetos que tiene que mostrar; esto es, enlazar su propiedad **ItemsSource** con el contexto de datos que acabamos de definir, según se muestra a continuación:

```
<Grid Name="gridAlumnos">
  ...
    <TextBlock Name="tbTitulo" FontSize="20" Text="Lista de alumnos:"/>
    <ListBox Name="listaAlumnos"
             Height="300" Width="330" Margin="10"
             ItemsSource="{Binding}"
             ItemTemplate="{StaticResource plantillaAlumno}"
             HorizontalContentAlignment="Stretch"
             IsSynchronizedWithCurrentItem="True">
    </ListBox>
```

```
    ...
</Grid>
```

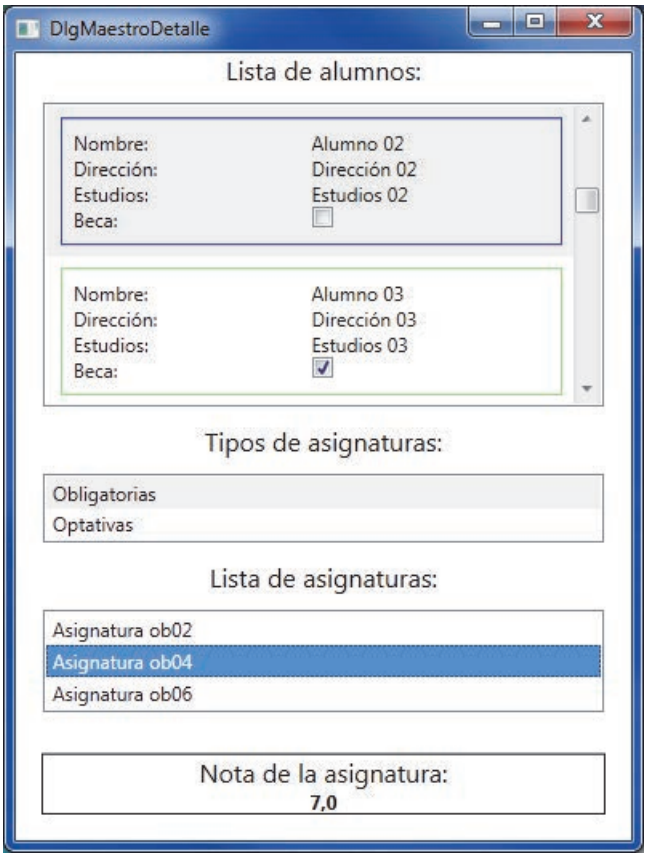
ItemsSource="{Binding}" especifica que se utilizará como origen de datos el contexto actual; en este caso, según el atributo **XPath** del enlace entre *gridAlumnos* y *xmlDoc*, el origen de datos es la colección de elementos *alumno* cargada en *xmlDoc* desde el fichero *alumnos.xml*.

Vinculación maestro-detalle

Un diseño de tipo maestro-detalle incluye dos partes: una vista que muestra una lista de elementos, normalmente una colección de datos, y una vista de detalles que muestra los detalles acerca del elemento que se selecciona en la lista anterior. Por ejemplo, este libro es un ejemplo de diseño de tipo maestro-detalle, donde la tabla de contenido es la vista que muestra una lista de elementos y el tema/apartado escrito es la vista de detalles.

La siguiente figura muestra con claridad lo que se pretende conseguir con este tipo de diseño. Observe: la ventana, inicialmente, presenta la lista de alumnos, en este caso con sus datos, el usuario selecciona un alumno y automáticamente se completa la lista de tipos de asignaturas en los que ese alumno participa, el usuario selecciona un tipo de asignaturas y se completa la lista de asignaturas (de las que el alumno está matriculado en cuanto a ese tipo se refiere) y, finalmente, el usuario selecciona la asignatura de la cual desea conocer la nota. Como habrá deducido, cada paso en la secuencia del proceso es un detalle del anterior.

Vamos a implementar este ejemplo en nuestra aplicación. Para ello, partiendo del diálogo *DlgDataTemplate* vamos a diseñar un nuevo diálogo, *DlgMaestroDetalle*, como el que muestra la figura siguiente. Entonces, partiendo de que la lista de alumnos ya la tenemos, lo que tendremos que hacer básicamente será añadir una nueva lista para los tipos de asignaturas correspondientes al alumno seleccionado, otra para las asignaturas correspondientes al tipo de asignaturas seleccionado y un control de texto para mostrar la nota de la asignatura seleccionada.



Partimos de que disponemos de un recurso definido en los recursos de la aplicación que pone a nuestra disposición el proveedor de datos XML.

```
<XmlDataProvider x:Key="xmlAlumnos"
      Source="datos/alumnos.xml" XPath="/alumnos/alumno"/>
```

En este nuevo diseño, hemos establecido como contexto de datos del **Grid** principal el origen de datos definido por el proveedor de datos XML. Este contexto de datos (colección de alumnos) será ahora el origen de datos para la lista de alumnos *lstAlumnos*, con lo cual esta lista mostrará la lista de alumnos basándose en la plantilla *plantillaAlumno*.

```
<Window x:Class="DlgMaestroDetalle"
        ...
        Title="DlgMaestroDetalle" Height="530" Width="400">
  <Window.Resources>
    <DataTemplate x:Key="plantillaAlumno">
      ...
    </DataTemplate>
```

```
    ...
  </Window.Resources>

  <Grid DataContext="{StaticResource xmlAlumnos}">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>
    <StackPanel Grid.Row="0">
      <TextBlock FontSize="16" Text="Lista de alumnos:"
                 HorizontalAlignment="Center"/>
      <ListBox Name="lstAlumnos"
               Height="190" Width="350" Margin="10"
               ItemsSource="{Binding}"
               ItemTemplate="{StaticResource plantillaAlumno}"
               HorizontalContentAlignment="Stretch">
      </ListBox>
    </StackPanel>
    <StackPanel Grid.Row="1">
      <!-- Lista de tipos de asignaturas -->
    </StackPanel>
    <StackPanel Grid.Row="2">
      <!-- Lista de asignaturas -->
    </StackPanel>
    <Border Grid.Row="3" BorderBrush="Black" BorderThickness="1"
            Margin="16">
      <StackPanel>
        <!-- Nota -->
      </StackPanel>
    </Border>
  </Grid>
</Window>
```

Observe que a la lista de alumnos le siguen otros tres objetos **StackPanel** para mostrar los detalles anteriormente descritos.

El siguiente paso es mostrar los tipos de asignaturas del alumno seleccionado en la lista anterior. Para ello establecemos como contexto de datos del contenedor de esta lista el elemento (*alumno*) seleccionado en la lista anterior. Un *alumno* contiene elementos *asignaturas*, los que tiene que mostrar esta lista; por lo tanto, el origen de datos de esta lista será *asignaturas*. Estos elementos se mostrarán de acuerdo con la plantilla *plantillaAsignaturas*.

```
<StackPanel Grid.Row="1"
      DataContext="{Binding SelectedItem, ElementName=lstAlumnos}">
  <TextBlock FontSize="16" Text="Tipos de asignaturas:"
             HorizontalAlignment="Center"/>
```

```
  <ListBox Name="lstTAsigs" Width="350" Margin="10"
           ItemsSource="{Binding XPath=asignaturas}"
           ItemTemplate="{StaticResource plantillaAsignaturas}">
  </ListBox>
</StackPanel>
```

La plantilla *plantillaAsignaturas* para mostrar los nombres de los tipos de asignaturas la implementaremos como un recurso de la ventana así:

```
<DataTemplate x:Key="plantillaAsignaturas">
  <TextBlock Text="{Binding XPath=@tipo}" Padding="2" />
</DataTemplate>
```

El siguiente paso es mostrar las asignaturas correspondientes al tipo de asignaturas seleccionado en la lista anterior. Para ello establecemos como contexto de datos del contenedor de esta lista el elemento (*asignaturas*) seleccionado en la lista anterior. Un elemento *asignaturas* contiene elementos *asignatura*, los que tiene que mostrar esta lista; por lo tanto, el origen de datos de esta lista será *asignatura*. Estos elementos se mostrarán de acuerdo con la plantilla *plantillaAsignatura*.

```
<StackPanel Grid.Row="2"
      DataContext="{Binding SelectedItem, ElementName=lstTAsigs}">
  <TextBlock FontSize="16" Text="Lista de asignaturas:"
             HorizontalAlignment="Center"/>
  <ListBox Name="lstAsigs" Width="350" Margin="10"
           ItemsSource="{Binding XPath=asignatura}"
           ItemTemplate="{StaticResource plantillaAsignatura}">
  </ListBox>
</StackPanel>
```

La plantilla *plantillaAsignatura* para mostrar los nombres de las asignaturas la implementaremos como un recurso de la ventana así:

```
<DataTemplate x:Key="plantillaAsignatura">
  <TextBlock Text="{Binding XPath=nombre}" Padding="2" />
</DataTemplate>
```

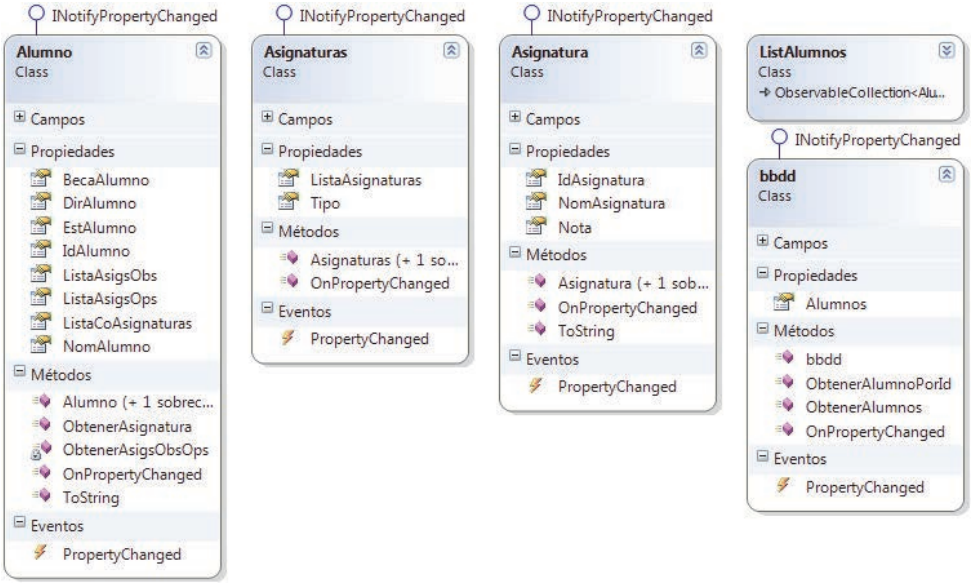
Finalmente, hay que mostrar la nota de la asignatura seleccionada en la lista anterior. Para ello establecemos como contexto de datos del contenedor de esta lista el elemento (*asignatura*) seleccionado en la lista anterior. Un elemento *asignatura* contiene los elementos *nombre* y *nota*, de los cuales sólo mostraremos la nota en un control **TextBlock**.

```
<Border Grid.Row="3" BorderBrush="Black" BorderThickness="1"
        Margin="16">
  <StackPanel
      DataContext="{Binding SelectedItem, ElementName=lstAsigs}">
```

```
    <TextBlock FontSize="16" Text="Nota de la asignatura:"
               HorizontalAlignment="Center"/>
    <TextBlock Text="{Binding XPath=nota}"
               HorizontalAlignment="Center" FontWeight="Bold" />
  </StackPanel>
</Border>
```

OBJETOS COMO FUENTE DE DATOS

Por lo estudiado hasta ahora, hemos comprobado que el enlace de datos de WPF proporciona un método simple y coherente para que las aplicaciones interactúen con datos. También hemos visto que los orígenes de datos pueden estar basados en elementos XML o en objetos CLR (objetos de clases propias o de clases de alguna biblioteca). Pues bien, en este apartado vamos a trabajar con orígenes de datos que sean colecciones de objetos personalizados. Los datos serán tomados de una base de datos modelada por medio de las clases siguientes:



Realizamos una breve descripción de este modelo de datos. Como hemos indicado anteriormente vamos a trabajar con orígenes de datos que sean colecciones de clases de objetos; en nuestro caso, estamos trabajando con una colección de alumnos (clase *ListAlumnos*: **ObservableCollection(Of** *Alumno***)**) que son objetos de la clase *Alumno*. Cada alumno tiene una colección de colecciones de asignaturas, *ListaCoAsignaturas*, que son objetos de la clase *Asignaturas* y cada objeto *Asignaturas* contiene una colección de asignaturas, *ListaAsignaturas*, obligatorias u optativas, que son objetos de la clase *Asignatura*.

La clase *Alumno* representa un alumno y proporciona varias propiedades para acceder a cada uno de los datos del mismo, más la propiedad *ListaCoAsignaturas* que permite acceder a la colección **ObservableCollection(Of** *Asignaturas***)** correspondiente a la lista de colecciones de asignaturas, obligatorias y optativas, de las que se ha matriculado el alumno, la propiedad *ObtenerAsigsObs* que permite acceder a la colección **ObservableCollection(Of** *Asignatura***)** de asignaturas obligatorias, la propiedad *ObtenerAsigsOps* que permite acceder a la colección **ObservableCollection(Of** *Asignatura***)** de asignaturas optativas y el método *ObtenerAsignatura* que devuelve el objeto *Asignatura* correspondiente al identificador de la asignatura pasado como argumento.

La clase *Asignaturas* representa una colección de asignaturas, *ListaAsignaturas*, obligatorias u optativas, característica especificada por su propiedad *Tipo*.

La clase *Asignatura*, que representa una asignatura, básicamente presenta tres propiedades: el identificador de la asignatura, el nombre y la nota que se obtenga en la misma.

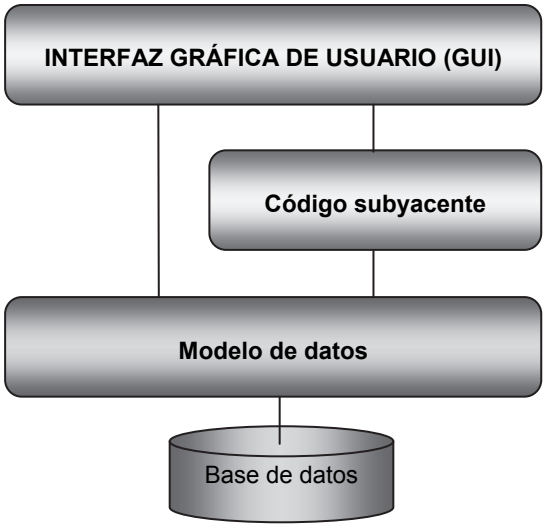
La clase *bbdd* representa la base de datos. Define un objeto de la clase *ListAlumnos* derivada de la colección **ObservableCollection(Of** *Alumno***)**; un objeto de esta clase será el que almacene todos los alumnos matriculados; esta colección será creada por el método privado *ObtenerAlumnos* cuando se cree un objeto de la clase *bbdd* y se accederá a la misma por medio de su propiedad *Alumnos*. También proporciona el método *ObtenerAlumnoPorId* que devuelve el objeto *Alumno* correspondiente al identificador pasado como argumento.

Todas las colecciones son de la clase **ObservableCollection(Of** *T***)** ya que ésta implementa la interfaz **INotifyCollectionChanged**.

Puede echar un vistazo a la figura anterior y observar de una forma más amplia todas las propiedades y métodos que proporciona cada clase. Todas las clases implementan la interfaz **INotifyPropertyChanged** para que los objetos de estas clases puedan notificar a la interfaz gráfica un cambio en el valor de cualquiera de sus propiedades, con el fin de que los elementos de dicha interfaz que estén enlazados con estos objetos puedan ser actualizados automáticamente (véase el apartado *Notificar cuándo cambia una propiedad* del capítulo *Introducción a WPF*). La notificación la realizan los objetos de esas clases generando el evento **PropertyChanged** a través del método **OnPropertyChanged** que lleva asociado como datos el nombre de la propiedad cuyo valor ha cambiado. Recuerde que el motor de enlace a datos de WPF utiliza esta interfaz para mantener la interfaz gráfica sincronizada con el objeto origen del enlace.

Para crear este modelo de datos, simplemente hemos creado un proyecto *BaseDeDatos* de tipo *Biblioteca de clases* y, a continuación, hemos añadido cada

una de las clases que hemos descrito anteriormente. De esta forma, cualquier otro proyecto que quiera utilizar este modelo, simplemente tendrá que añadir una referencia a la biblioteca generada: *BaseDeDatos.dll* (también puede añadir este proyecto a cualquier otra solución, lo que le permitirá realizar modificaciones en esta biblioteca mientras construye el proyecto que la utiliza o, simplemente, tener el código de la misma presente). La base de datos la hemos simulado generando los datos desde el código. Puede ver el código de la biblioteca *BaseDeDatos* en el CD del libro.



Enlace a una colección de objetos

Al igual que Windows Forms, WPF permite implementar enlaces de datos para obtener información desde casi cualquier propiedad de cualquier objeto y colocarla en casi cualquier propiedad de cualquier elemento. WPF, como hemos visto anteriormente, también incluye una serie de controles de lista que permiten trabajar con colecciones enteras de objetos así como navegar a través de ellos. No obstante, comparado con Windows Forms, existen importantes cambios en la forma en como se implementan y como trabajan estos enlaces, conceptos que también hemos ido viendo en lo que hemos estudiado hasta ahora y en los que seguiremos profundizando.

Un objeto de la clase *bbdd* encapsula una colección de objetos, en nuestro caso, una colección de objetos de la clase *Alumno*. Entonces, ¿qué tenemos que hacer para poner esa colección a disposición de una aplicación? Pues tendremos que crear un objeto de la clase *bbdd* y ponerlo a disposición de la aplicación en el con-

texto de datos adecuado. Esto lo podemos hacer desde el código subyacente o desde el código XAML de la ventana que lo vaya a utilizar en cada caso.

Por ejemplo, el siguiente código asigna al contexto de datos del control *gridAlumnos* la colección de alumnos devuelta por la propiedad *Alumnos* de *bbdd*:

```
Imports BaseDeDatos
Public Class MiDialogo
  Private objBBDD As New bbdd()
  Public Sub New()
    InitializeComponent()
    gridAlumnos.DataContext = objBBDD.Alumnos
  End Sub
End Class
```

Este otro ejemplo define el objeto de la clase *bbdd* como un recurso de la aplicación. Observe que es necesario especificar el espacio de nombres, así como el ensamblado, al que pertenece la clase *bbdd*:

```
<Application x:Class="MiAplicacion"
  ...
  xmlns:local="clr-namespace:BaseDeDatos;assembly=BaseDeDatos">
  <Application.Resources>
    <local:bbdd x:Key="objBBDD" />
  </Application.Resources>
</Application>
```

Ahora, para asignar al contexto de datos del control *gridAlumnos* el objeto de la clase *bbdd* referenciado por el recurso *objBBDD*, procederíamos así:

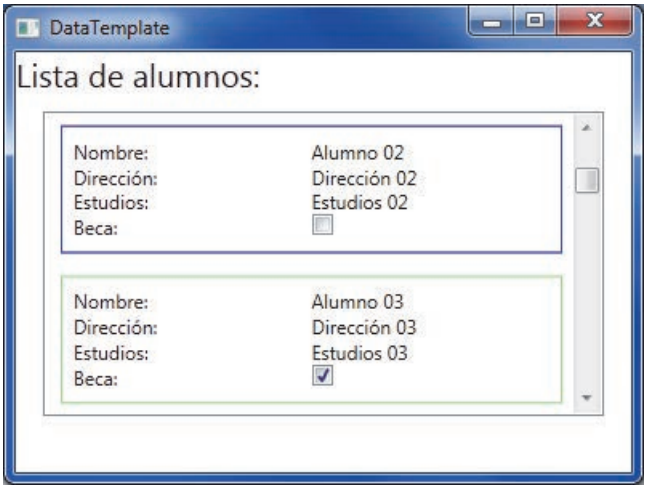
```
<Window x:Class="MiDialogo"
   ...
  <Grid Name="gridAlumnos" DataContext="{StaticResource objBBDD}">

  </Grid>
</Window>
```

Un control que quiera utilizar la colección de alumnos proporcionada por el objeto *bbdd* del contexto anterior establecería un enlace con su propiedad *Alumnos*. Por ejemplo, el siguiente código hará que se muestre la colección de alumnos en la lista *lstAlumnos*:

```
<ListBox Name="lstAlumnos" Height="190" Width="350" Margin="10"
          ItemsSource="{Binding Path=Alumnos}"
          ItemTemplate="{StaticResource plantillaAlumno}"
          HorizontalContentAlignment="Stretch">
```

Por ejemplo, si aplica estos cambios al diálogo que implementamos en el apartado *Plantillas de datos*, el resultado será el mismo, lógicamente:



Vamos a ponerlo en práctica. Cree un nuevo proyecto *EnlaceDeDatosObjetos*. Añada una referencia a la biblioteca *BaseDeDatos.dll*. Al igual que hizo en el proyecto *EnlaceDeDatosXML*, añada a la ventana principal una barra de menús con las órdenes *Archivo*, *Diálogos* y *Ayuda* y dóteles de la misma funcionalidad (el menú *Diálogos* mostrará los diálogos que vayamos creando).

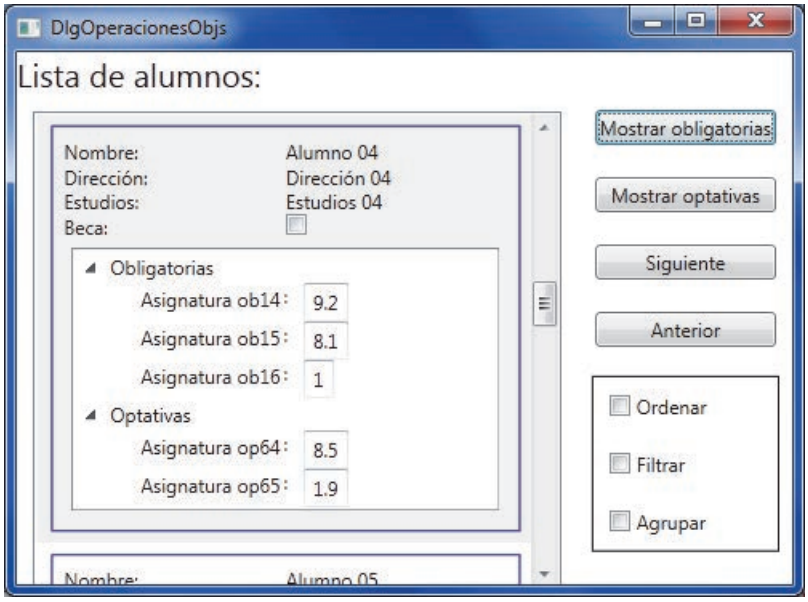
Añada una nueva ventana que dé lugar al diálogo de la figura anterior (copie el código del mismo diálogo del proyecto *EnlaceDeDatosXML*). Respecto a la versión de este diálogo en el proyecto *EnlaceDeDatosXML*, ¿qué cambios tenemos que realizar para que obtenga los datos de nuestra base de datos? En los recursos de la aplicación, quite el proveedor de datos XML y añada el recurso *objBBDD* (un objeto de la clase *bbdd*), según explicamos anteriormente en este mismo apartado. En la plantilla *plantillaAlumno* utilice en los enlaces los nombres que hemos dado a las propiedades en la clase *Alumno*: *NomAlumno*, *DirAlumno*, *EstAlumno* y *BecaAlumno*. Finalmente, como indicamos anteriormente, asigne al contexto de datos del **Grid** principal (aquí lo hemos denominado *gridAlumnos*) el recurso *objBBDD* y a la propiedad **ItemsSource** del **ListBox**, el valor *"{Binding Path=Alumnos}"* (*Alumnos* es la propiedad del objeto *bbdd* que hace referencia a la colección de objetos *Alumno*). Pruebe la aplicación.

Vistas de colección de objetos

Como explicamos al principio de este capítulo, una *vista de colección* es un objeto situado un nivel por encima de la colección proporcionada por el origen del enlace. Dicha vista permite navegar y mostrar la colección en función de las consultas

de ordenación, filtrado y agrupación, sin tener que cambiar la propia colección subyacente en el origen.

Los cambios en la colección se propagarán a las vistas si la colección y los objetos que contiene implementan la interfaz **INotifyCollectionChanged**, cosa que ocurre en nuestro modelo de datos, pero las vistas no cambian la colección subyacente; por eso, cada colección proporcionada por el origen puede tener varias vistas asociadas. Esto quiere decir que el uso de vistas permitirá mostrar los mismos datos de formas diferentes. Veamos un ejemplo que dé como resultado la ventana mostrada a continuación. Recuerde que el origen de datos es proporcionado por el recurso *objBBDD* de la aplicación.



Añada una nueva ventana que dé lugar al diálogo de la figura anterior (copie el código del mismo diálogo *DlgOperacionesXml* del proyecto *EnlaceDeDatosXML* y realice las modificaciones indicadas a continuación). Análogamente a como procedimos en el diálogo anterior, vamos a ver qué cambios tenemos que realizar en este diálogo con respecto a su versión XML. En la plantilla *plantillaAlumno* utilice en los enlaces los nombres que hemos dado a las propiedades en la clase *Alumno*: *NomAlumno*, *DirAlumno*, *EstAlumno* y *BecaAlumno*.

El origen de datos para el control **TreeView** será ahora la propiedad *ListaCoAsignaturas* de la clase *Alumno*:

```
<TreeView Grid.Row="4" Grid.ColumnSpan="2"
          Name="trvAlumnos" Margin="5"
          ItemsSource="{Binding Path=ListaCoAsignaturas}"
```

```
          ItemTemplate="{StaticResource kAsignaturas}">
</TreeView>
```

En la plantilla *kAsignaturas* el origen de datos será la propiedad *ListaAsignaturas* de la clase *Asignaturas* y la propiedad **Text** del control **TextBlock** tenemos que enlazarla con la propiedad *Tipo* de la misma clase:

```
<!-- PLANTILLA ASIGNATURAS -->
<HierarchicalDataTemplate x:Key="kAsignaturas"
    ItemsSource="{Binding Path=ListaAsignaturas}"
    ItemTemplate="{StaticResource kAsignatura}">
  <TextBlock Text="{Binding Path=Tipo}" Padding="2" />
</HierarchicalDataTemplate>
```

En la plantilla *kAsignatura*, enlace los controles **TextBlock** con las propiedades correspondientes de la clase *Asignatura*:

```
<!-- PLANTILLA ASIGNATURA -->
<DataTemplate x:Key="kAsignatura">
  <WrapPanel>
    <TextBlock Text="{Binding Path=NomAsignatura}" Padding="2" />
    <TextBlock Text=":   " />
    <TextBox Text="{Binding Path=Nota}" Padding="2" />
  </WrapPanel>
</DataTemplate>
```

Finalmente, como indicamos anteriormente, asigne al contexto de datos del **Grid** principal el recurso *objBBDD* y a la propiedad **ItemsSource** del **ListBox**, el valor *"{Binding Path=Alumnos}"*.

Ejecute la aplicación y pruebe a modificar las notas. Cierre el diálogo y vuélvalo a abrir. Observará que los cambios persisten, ¿por qué? Pues porque el enlace es *TwoWay* y las clases de objetos que utilizamos en nuestro modelo de datos implementan la interfaz **INotifyPropertyChanged** que, además de permitir notificar a los controles enlazados que una propiedad ha cambiado, es la que utiliza el motor de enlace a datos de WPF (los objetos **Binding**) para mantener la IGU sincronizada con el objeto origen del enlace. Esta sincronización lleva implícito actualizar las propiedades enlazadas de los objetos de datos. Lógicamente, como la base de datos está simulada con una colección que se crea cuando se inicia la aplicación, cuando se cierre ésta los cambios se perderán, pero con una base de datos real los cambios se guardarían, como veremos en el próximo capítulo.

El enlace *TwoWay* permite que los cambios realizados en la propiedad de origen se reflejen automáticamente en la de destino y viceversa. Este tipo de enlace es adecuado para formularios modificables. Sepa que la mayoría de las propiedades tienen el enlace *OneWay* de forma predeterminada, excepto algunas propieda-

des de dependencia que tienen el enlace *TwoWay*, normalmente pertenecientes a controles modificables por el usuario, como la propiedad **Text** de **TextBox** o la propiedad **IsChecked** de **CheckBox**.

Una vez que la ventana es capaz de mostrar los datos, continuemos con las operaciones de navegar y mostrar la colección en función de las consultas de ordenación, filtrado y agrupación. Como ya explicamos anteriormente, estas operaciones son facilitadas por las vistas de colección, que son clases que implementan la interfaz **ICollectionView**.

Obtener la vista

Cada colección proporcionada por el origen del enlace tiene una vista predeterminada. ¿Cómo se obtiene dicha vista? Para crear la vista, se necesita una referencia a la colección subyacente en el origen que, en nuestro caso, podemos obtener a través de la propiedad **ItemsSource** del **ListBox** *listaAlumnos*. Según esto, y continuando con las modificaciones de la versión XML para adaptarla a este nuevo modelo de datos, el método *ObtenerVista* de la clase *DlgOperacionesObjs* será así:

```
Private Function ObtenerVista() As ICollectionView
  Return CollectionViewSource.GetDefaultView(listaAlumnos.ItemsSource)
End Function
```

En este caso, el objeto devuelto es de tipo **ListCollectionView**, según explicamos en el apartado *Vistas de colección* anteriormente en este mismo capítulo, ya que la colección es de tipo **ObservableCollection(Of** *T***)**, que se deriva de **Collection(Of** *T***)**, e implementa la interfaz **IList(Of** *T***)**.

Elemento actual

El elemento actual está siempre referenciado por la propiedad **CurrentItem** de la vista. Como ejemplo, vamos a escribir los controladores del evento **Click** de los botones "Mostrar obligatorias" y "Mostrar optativas" para que muestren en una ventana de mensaje las asignaturas obligatorias y optativas, respectivamente, de las que está matriculado el alumno actualmente seleccionado.

```
Private Sub btMostrarOb_Click(sender As Object, e As RoutedEventArgs)
  MostrarAsigs("Obligatorias")
End Sub

Private Sub btMostrarOp_Click(sender As Object, e As RoutedEventArgs)
  MostrarAsigs("Optativas")
End Sub
```

```
Private Sub MostrarAsigs(tipo As String)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.IsEmpty Then Return
  ' Cada "alumno" es un objeto de la clase Alumno
  Dim alumnoActual As Alumno = DirectCast(vista.CurrentItem, Alumno)
  Dim list As ObservableCollection(Of Asignatura) = Nothing
  If tipo = "Obligatorias" Then
    list = alumnoActual.ListaAsigsObs
  ElseIf tipo = "Optativas" Then
    list = alumnoActual.ListaAsigsOps
  Else
    Return
  End If
  Dim s As New StringBuilder()
  For Each asig As Asignatura In list
    s.Append(asig.NomAsignatura + vbLf)
  Next
  MessageBox.Show(s.ToString())
End Sub
```

Este método obtiene la vista de la colección de datos, obtiene el elemento actualmente seleccionado en la vista (un objeto *Alumno*), extrae la colección de asignaturas obligatorias u optativas invocando al método correspondiente de la clase *Alumno*, en función del botón pulsado, y obtiene de la colección los nombres de las asignaturas, para después visualizarlos.

Navegar

El sistema de navegación no varía con respecto a lo que ya vimos. Simplemente, hay que utilizar los métodos **MoveCurrentToFirst**, **MoveCurrentToLast**, **MoveCurrentToNext** o **MoveCurrentToPrevious** de la clase **CollectionView**. Por ejemplo:

```
Private Sub btSiguiente_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToNext()
  If vista.IsCurrentAfterLast Then
    vista.MoveCurrentToLast()
  End If
  listaAlumnos.ScrollIntoView(vista.CurrentItem)
End Sub
```

Este método primero obtiene la vista de la colección de datos y después, para avanzar al siguiente elemento, la vista invoca a su método **MoveCurrentToNext**. Si el movimiento solicitado va más allá del último elemento, éste pasará a ser el elemento actual. Observe también que después de cada movimiento, el **ListBox** invoca a su método **ScrollIntoView** pasando como argumento el elemento ac-

tualmente seleccionado para forzar a la lista a realizar el desplazamiento requerido para que se muestre dicho elemento. El controlador del evento **Click** del botón "Anterior" es análogo.

Ordenar

La ordenación de los elementos de la vista (recuerde que la vista siempre se sitúa un nivel por encima de la colección) tampoco varía con respecto a la versión anterior, basta con agregar a la colección **SortDescriptions** los criterios **SortDescription** de ordenación, como puede comprobar en el método mostrado a continuación, correspondiente al controlador del evento **Click** de la casilla de verificación "Ordenar":

```
Private Sub cvOrdenar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CanSort AndAlso cvOrdenar.IsChecked = True Then
    If vista.SortDescriptions.Count = 0 Then
      vista.SortDescriptions.Add(New SortDescription("BecaAlumno",
        ListSortDirection.Descending))
      vista.SortDescriptions.Add(New SortDescription("NombAlumno",
        ListSortDirection.Ascending))
    End If
  Else
    vista.SortDescriptions.Clear()
  End If
End Sub
```

Observe que este método alterna entre dos vistas: si la casilla de verificación está marcada, se presentará la vista ordenada y si no, se mostrará en su estado inicial. Esto último se consigue eliminando de la colección **SortDescriptions** todos los criterios de ordenación.

Filtrar

Se trata de filtrar los elementos provenientes de la colección subyacente para crear una vista con los elementos de la colección que cumplan unos determinados criterios. Para especificar tales condiciones necesitamos vincular la propiedad de la vista **Filter**, de tipo **Predicate(Of Object)**, con el método que se utilice para determinar qué elementos pertenecerán a la misma. Según el tipo de esta propiedad, podemos asignarle un método anónimo acorde al delegado **Predicate**. Según esto, el controlador del evento **Click** de la casilla "Filtrar", para que la vista muestre sólo los alumnos que tienen beca, será así:

```
Private Sub cvFiltrar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CanFilter AndAlso cvFiltrar.IsChecked = True Then
```

```
    If vista.Filter Is Nothing Then
      vista.Filter = Function(elemento As Object) _
        DirectCast(elemento, Alumno).BecaAlumno = True
    End If
  Else
    vista.Filter = Nothing
  End If
End Sub
```

Observe que la condición impuesta para que un *Alumno* se incluya en la vista es que el valor del elemento *BecaAlumno* sea **True**.

Agrupar

Para agrupar los elementos de la vista basta con agregar a la colección **GroupDescriptions** los criterios **PropertyGroupDescription** bajo los que se agruparán en la vista los elementos de la colección. De acuerdo a lo expuesto, el controlador del evento **Click** de la casilla de verificación "Agrupar" para que, si es posible, agrupe los elementos de la vista por el valor *BecaAlumno*, será así:

```
Private Sub cvAgrupar_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CanGroup AndAlso cvAgrupar.IsChecked = True Then
    If vista.GroupDescriptions.Count = 0 Then
      ' Agrupar por BecaAlumno
      vista.GroupDescriptions.Add(New PropertyGroupDescription("BecaAlumno"))
    End If
  Else
    vista.GroupDescriptions.Clear()
  End If
End Sub
```

Observe que, igual que en los casos anteriores, este método alterna entre dos vistas: si la casilla de verificación está marcada, se presentará la vista agrupada y si no, se mostrará en su estado inicial.

Insertar y borrar elementos de la colección

Supongamos que añadimos a nuestra ventana un botón "Borrar" con la intención de eliminar el elemento actualmente seleccionado de la lista. Para realizar tal operación, el controlador para el evento **Click** de este botón podría escribirse así:

```
Private Sub btBorrar_Click(sender As Object, e As RoutedEventArgs)
  ' Recuperar el recurso objBBDD de Application
  Dim objBBDD As bbdd = TryCast(Me.FindResource("objBBDD"), bbdd)
  objBBDD.Alumnos.Remove(TryCast(listaAlumnos.SelectedItem, Alumno))
End Sub
```

Ejecutamos la aplicación, visualizamos el diálogo *DlgOperacionesObjs*, se muestra la lista de alumnos, seleccionamos un alumno, hacemos clic en el botón "Borrar" y observamos que todo ha funcionado correctamente. Pero, para que esto suceda la colección tiene que implementar la interfaz **INotifyCollectionChanged**, lo cual quiere decir que nuestra colección, que es de tipo **ObservableCollection(Of *T*)**, implementa esta interfaz.

WPF proporciona la clase **ObservableCollection(Of *T*)** que implementa la interfaz **INotifyCollectionChanged**, lo que permite configurar enlaces dinámicos de modo que las inserciones o eliminaciones que se realicen en la colección se reflejen de forma automática en la vista. Esta interfaz expone el evento **CollectionChanged**, que debe generarse siempre que se realicen cambios en la colección subyacente. Evidentemente, los objetos de datos individuales de la colección también deben cumplir el requisito que ya hemos descrito: implementar la interfaz **INotifyPropertyChanged**.

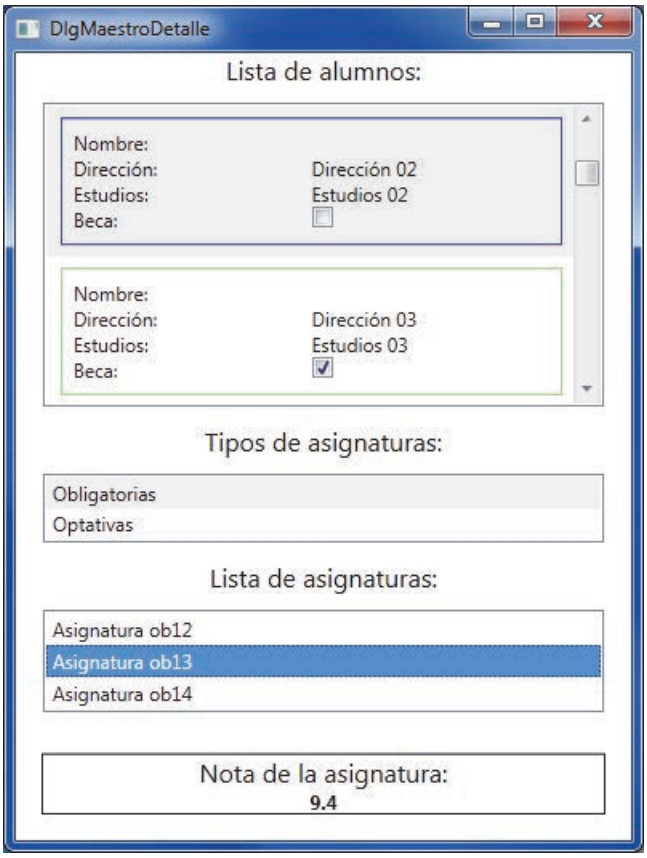
Análogamente, podríamos añadir un elemento a la colección y el cambio se reflejaría también automáticamente en la vista. Por ejemplo:

```
Dim alum As New Alumno(123, "Nombre", "Dirección",
                       "Estudios", False, Nothing)
objBBDD.Alumnos.Add(alum)
```

Veremos un ejemplo completo un poco más adelante, concretamente en el apartado *Datos introducidos por el usuario*.

Vinculación maestro-detalle

Como ya explicamos anteriormente, un diseño de tipo maestro-detalle incluye dos partes: una vista que muestra una lista de elementos, normalmente una colección de datos, y una vista de detalles que muestra los detalles acerca del elemento que se selecciona en la lista anterior. Por ejemplo, la figura siguiente muestra con claridad lo que se pretende conseguir con este tipo de diseño. Vamos a construir esta ventana apoyándonos en la versión XML que ya construimos anteriormente.



Añada una nueva ventana que dé lugar al diálogo de la figura anterior (copie el código del mismo diálogo *DlgMaestroDetalle* del proyecto *EnlaceDeDatosXML* y realice las modificaciones indicadas a continuación). Análogamente a como procedimos en el diálogo anterior, vamos a ver qué cambios tenemos que realizar en este diálogo con respecto a su versión XML. En la plantilla *plantillaAlumno* utilice en los enlaces los nombres que hemos dado a las propiedades en la clase *Alumno*: *NomAlumno*, *DirAlumno*, *EstAlumno* y *BecaAlumno*.

Partimos de que disponemos de un recurso *objBBDD* definido en los recursos de la aplicación que pone a nuestra disposición la colección *Alumnos*:

```
<Application x:Class="MiAplicacion"
  ...
  xmlns:local="clr-namespace:BaseDeDatos;assembly=BaseDeDatos">
  <Application.Resources>
    <local:bbdd x:Key="objBBDD" />
  </Application.Resources>
</Application>
```

Asignamos al contexto de datos del control *gridAlumnos* el recurso anterior para que los controles secundarios puedan acceder a la colección *Alumnos*:

```
<Window x:Class="MiDialogo"
   ...
  <Grid Name="gridAlumnos" DataContext="{StaticResource objBBDD}">

  </Grid>
</Window>
```

La colección *Alumnos* de *objBBDD* será el origen de datos de la lista de alumnos *lstAlumnos* para que muestre la lista de alumnos de acuerdo con la plantilla *plantillaAlumno*:

```
<ListBox Name="lstAlumnos" Height="190" Width="350" Margin="10"
          ItemsSource="{Binding Path=Alumnos}"
          ItemTemplate="{StaticResource plantillaAlumno}"
          HorizontalContentAlignment="Stretch">
```

A la lista de alumnos le siguen otros tres controles para mostrar los detalles que observamos en la figura anterior: tipos de asignaturas del alumno seleccionado, asignaturas del tipo seleccionado y nota de la asignatura seleccionada.

Para mostrar los tipos de asignaturas del alumno seleccionado en la lista anterior, establecemos como contexto de datos del contenedor de esta nueva lista el objeto *Alumno* seleccionado en la lista anterior, que tiene una propiedad *ListaCoAsignaturas* que contiene objetos *Asignaturas* cuya propiedad *Tipo* queremos mostrar en esta lista de acuerdo con la plantilla *plantillaAsignaturas*:

```
<StackPanel Grid.Row="1"
      DataContext="{Binding SelectedItem, ElementName=lstAlumnos}">
  <TextBlock FontSize="16" Text="Tipos de asignaturas:"
             HorizontalAlignment="Center"/>
  <ListBox Name="lstTAsigs" Width="350" Margin="10"
            ItemsSource="{Binding Path=ListaCoAsignaturas}"
            ItemTemplate="{StaticResource plantillaAsignaturas}">
  </ListBox>
</StackPanel>
```

La plantilla *plantillaAsignaturas* para mostrar los nombres de los tipos de asignaturas la implementaremos como un recurso de la ventana así:

```
<DataTemplate x:Key="plantillaAsignaturas">
  <TextBlock Text="{Binding Path=Tipo}" Padding="2" />
</DataTemplate>
```

Para mostrar las asignaturas correspondientes al tipo de asignaturas seleccionado en la lista anterior, establecemos como contexto de datos del contenedor de esta nueva lista el objeto *Asignaturas* seleccionado en la lista anterior, que tiene una propiedad *ListaAsignaturas* que contiene objetos *Asignatura* cuya propiedad *NomAsignatura* queremos mostrar de acuerdo con la plantilla *plantillaAsignatura*:

```
<StackPanel Grid.Row="2"
        DataContext="{Binding SelectedItem, ElementName=lstTAsigs}">
  <TextBlock FontSize="16" Text="Lista de asignaturas:"
             HorizontalAlignment="Center"/>
  <ListBox Name="lstAsigs" Width="350" Margin="10"
           ItemsSource="{Binding Path=ListaAsignaturas}"
           ItemTemplate="{StaticResource plantillaAsignatura}">
  </ListBox>
</StackPanel>
```

La plantilla *plantillaAsignatura* para mostrar los nombres de las asignaturas la implementaremos como un recurso de la ventana así:

```
<DataTemplate x:Key="plantillaAsignatura">
  <TextBlock Text="{Binding Path=NomAsignatura}" Padding="2" />
</DataTemplate>
```

Finalmente, para mostrar la nota de la asignatura seleccionada en la lista anterior, establecemos como contexto de datos del contenedor de esta lista el objeto *Asignatura* seleccionado en la lista anterior, que tiene una propiedad *Nota* que contiene el valor que queremos mostrar en un control **TextBlock**:

```
<Border Grid.Row="3" BorderBrush="Black" BorderThickness="1"
        Margin="16">
  <StackPanel
        DataContext="{Binding SelectedItem, ElementName=lstAsigs}">
    <TextBlock FontSize="16" Text="Nota de la asignatura:"
               HorizontalAlignment="Center"/>
    <TextBlock Text="{Binding Path=Nota}"
               HorizontalAlignment="Center" FontWeight="Bold" />
  </StackPanel>
</Border>
```

Proveedor de datos de objetos

Anteriormente hemos visto dos formas de crear un objeto que se va a usar como origen de enlace: una, crear el objeto en el código subyacente y establecerlo en el **DataContext** de la ventana, y otra, crear el objeto en XAML en la sección de recursos de la aplicación (o de la ventana) y establecerlo en el **DataContext** de la

ventana. Una extensión de estos mecanismos es el proveedor de datos **ObjectDataProvider**.

El proveedor de datos **ObjectDataProvider** permite crear un objeto en el código XAML y ponerlo a disposición de un origen de datos. Así mismo, proporciona una serie de propiedades que permiten su configuración para ejecutar una consulta sobre el objeto origen de los datos y poner el resultado a disposición de un enlace. Por ejemplo:

```
<Window.Resources>
  <ObjectDataProvider x:Key="ColAlumnos"
                      ObjectType="{x:Type local:bbdd}"
                      MethodName="ObtenerAlumnos">
  </ObjectDataProvider>
<Window.Resources>
```

Este código crea un proveedor de datos de objetos como un recurso referenciado por *ColAlumnos*. La propiedad **ObjectType** hace referencia a la clase del objeto que se va a crear y **MethodName** hace referencia al nombre del método del objeto que se va a llamar para obtener el objeto origen del enlace. Si el método tuviera parámetros, utilizaríamos la propiedad **MethodParameters** para pasárselos. Así mismo, si tuviéramos que pasar parámetros al constructor del objeto, utilizaríamos la propiedad **ConstructorParameters**. El código siguiente muestra cómo utilizamos el resultado obtenido, esto es, el recurso *ColAlumnos*:

```
<Grid Name="gridAlumnos" DataContext="{StaticResource ColAlumnos}">
    ...
    <ListBox Name="listaAlumnos"
             Height="300" Width="330" Margin="10"
              ItemsSource="{Binding}"
              ItemTemplate="{StaticResource plantillaAlumno}"
              HorizontalContentAlignment="Stretch"
              IsSynchronizedWithCurrentItem="True">
    </ListBox>
    ...
</Grid>
```

También podemos utilizar la propiedad **IsAsynchronous** de **ObjectDataProvider** para especificar si la creación de objetos se realiza en el contexto activo (**False**, valor predeterminado) o en un subproceso de forma asíncrona. Cuando a esta propiedad se le asigna el valor **True**, el proveedor de datos hace su trabajo en un subproceso de forma asíncrona. Durante este tiempo los controles de la IGU no estarán enlazados a los datos. Cuando el método que obtiene los datos finaliza y devuelve el resultado, el proveedor de datos pone dicho resultado a disposición de todos los controles enlazados.

Una limitación que presenta el proveedor de datos de objetos cuando se define en el código XAML (en general, las llamadas a métodos desde XAML cuando de sus resultados depende la construcción de la ventana) es el tratamiento de las excepciones. Esto es, ¿qué sucede si el método *ObtenerAlumnos* lanza una excepción? Pues que la excepción se propaga hacia el método *InitializeComponent* que se invoca desde el constructor de la ventana. El código que muestra esta ventana necesita atrapar esta excepción para poder manipularla y continuar mostrando la ventana. Pero aun capturándola en el constructor, el resto de la ventana no se iniciaría correctamente. ¿Cuál es la solución? Una puede ser crear el proveedor de datos en el código subyacente de la ventana y otra, configurar el proveedor de datos para que haga su trabajo asíncronamente; en esta última situación, la excepción no interfiere en la creación de la ventana (decimos que la excepción es silenciosa, pero podríamos ver el mensaje de error en la ventana de depuración).

Virtualización

Cuando una aplicación trabaja con grandes cantidades de datos, miles y miles de registros, la técnica empleada debe ser capaz de hacerlo sin disminuir la velocidad y sin consumir una cantidad excesiva de memoria. Afortunadamente, los controles de lista de WPF, como **ListBox**, **ListView**, **DataGrid** y **TreeView**, están optimizados para ello.

El término "virtualizar" se refiere a una técnica mediante la cual se genera un subconjunto de elementos de la IGU a partir de un número mayor de elementos de datos y en función de los elementos que pueda visualizar el control. Por ejemplo, si una ventana tiene un control **ListBox** con 100.000 registros, pero el área visible puede mostrar sólo 10 registros, el **ListBox** creará los 10 elementos **ListBoxItem** y unos pocos más para asegurar un buen rendimiento en el desplazamiento. Si la lista no soportara la virtualización de la IGU, sería necesario generar los de 100.000 elementos **ListBoxItem**, con el consiguiente consumo de memoria y la cantidad de tiempo que sería necesario invertir para la asignación de estos elementos en el momento de establecer la propiedad **ItemsSource** de la lista.

El soporte para la virtualización de la IGU no está realmente integrado en los controles de lista o en sus hijos, sino que es proporcionado por el contenedor **VirtualizingStackPanel**, que funciona como un **StackPanel** pero dando soporte a la virtualización. Los controles ListBox, ListView y DataGrid utilizan de forma automática el contenedor **VirtualizingStackPanel** para contener sus elementos, con lo que no es necesario por parte del desarrollador tomar medidas adicionales.

El control **TreeView** admite la virtualización pero la tiene desactivada de forma predeterminada por compatibilidad con versiones anteriores. No obstante,

activarla es muy sencillo según muestra el código siguiente. Se recomienda activarla sólo en los árboles que contienen grandes cantidades de datos:

```
<TreeView VirtualizingStackPanel.IsVirtualizing="True"... >
```

Para los otros controles de lista mencionados, esta propiedad está establecida de forma predeterminada en **True**.

Ahora, en el caso de utilizar un control de lista que no soporte la virtualización, como sucede con el **ComboBox**, habría que implementarla explícitamente añadiendo una nueva plantilla **ItemsPanelTemplate**. Por ejemplo:

```
<ComboBox>
  <ComboBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel Width="250"
                              CanHorizontallyScroll="True"/>
    </ItemsPanelTemplate>
  </ComboBox.ItemsPanel>
</ComboBox>
```

Datos introducidos por el usuario

El fin de una aplicación puede ser simplemente procesar datos procedentes de un origen de datos y mostrar los datos y los resultados al usuario. Esto, como hemos podido comprobar, supone un trabajo importante en el desarrollo de una aplicación. Pero, una aplicación puede también solicitar datos al usuario, lo que requerirá analizar, aceptar o rechazar tales datos, simplemente porque los usuarios cometen errores tipográficos, olvidan especificar los valores necesarios, escriben valores en el sitio equivocado, eliminan o agregan registros que no deberían y, por lo general, cumplen la ley de Murphy siempre que pueden: "Si algo puede salir mal, saldrá mal". Evitar estas entradas erróneas o malintencionadas es otra parte importante en el desarrollo de una aplicación. ¿Cómo? Implementando reglas de validación para cada caso que se pueda presentar.

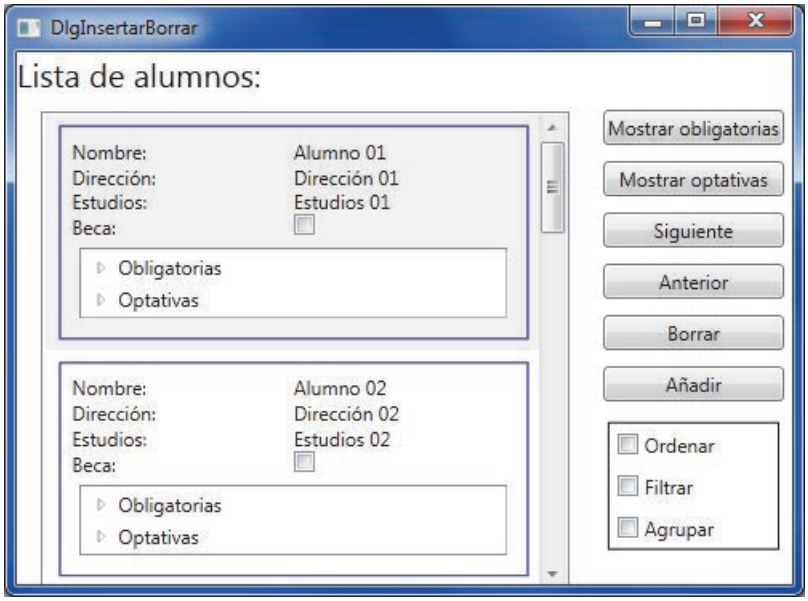
En el capítulo *Introducción a WPF* vimos la teoría, y algunos ejemplos, relacionada con la validación y la conversión de datos. Allí vimos que la clase **Binding** cuenta con una propiedad **ValidationRules**, que hace referencia a una colección que puede almacenar cualquier número de reglas de validación: clases derivadas de **ValidationRule**. Todas esas reglas contendrán la lógica que intenta comprobar si el valor enlazado es válido.

Vamos a realizar un ejemplo que nos permita, por una parte, estudiar cómo construir un diálogo que solicite datos al usuario y por otra, estudiar cómo validar

esos datos. La primera parte tendrá como finalidad añadir un objeto *Alumno* a la colección de alumnos de nuestra base de datos y la segunda, verificar que el identificador del alumno es correcto, que no se dejan cajas de texto vacías y que la lista de asignaturas obligatorias no está vacía.

Solicitar datos al usuario

Añada una nueva ventana que dé lugar al diálogo de la clase *DlgInsertarBorrar* de la figura siguiente (copie el código del diálogo *DlgOperacionesObjs* del proyecto *EnlaceDeDatosObjetos* que estamos desarrollando; la creación de un nuevo diálogo casi igual que el mencionado se hace con fines didácticos). Añada a la IGU de esta nueva ventana el botón "Borrar", si aún no lo ha añadido, y después, el botón "Añadir".




```
<Button Name="btBorrar" Content="Borrar" Margin="5"
        Click="btBorrar_Click" />
<Button Name="btAñadir" Content="Añadir" Margin="5"
        Click="btAñadir_Click" />
```

El controlador del evento **Click** del botón "Borrar" tiene como fin eliminar el elemento seleccionado de la lista, operación que ya fue expuesta anteriormente en el apartado *Insertar y borrar elementos de la colección*.

Y el controlador del evento **Click** del botón "Añadir" tiene como fin añadir un nuevo elemento (un nuevo alumno con sus datos personales y con las asignatu-

ras de las cuales se ha matriculado). Para ello, cuando el usuario haga clic sobre este botón, se abrirá un nuevo diálogo como el mostrado a continuación:



Entonces, como primer paso vamos a diseñar este diálogo. Para ello, agregue un nuevo diálogo a la aplicación, por ejemplo *DlgAñadir*, y realice el diseño mostrado en la figura. Obsérvese que tiene cuatro cajas de texto, una casilla de verificación, dos listas fijas y dos botones, más las etiquetas explicativas de los datos que hay que introducir o seleccionar en cada uno de los controles. El diseño de las cajas de diálogo personalizadas fue abordado en el capítulo *Controles y cajas de diálogo*, concretamente en el apartado *Cajas de diálogo personalizadas*, por lo que no vamos aquí a pararnos en ello. A continuación mostramos un resumen del código XAML que da lugar a este diseño:

```
<Window x:Class="DlgAñadir"...>
  <Border ...>
    <Grid Name="gridAlumno">
      ...

        <TextBlock Grid.Row="0" Grid.Column="0" Text="ID: "
                   Margin="7" />
        <TextBox Grid.Row="0" Grid.Column="1" Name="ctIdAlumno"
                 Margin="5" Text="{Binding Path=IdAlumno}" />
        <TextBlock Grid.Row="1" Grid.Column="0" Text="Nombre: "
                   Margin="7" />
        <TextBox Grid.Row="1" Grid.Column="1" Name="ctNomAlumno"
                 Margin="5" Text="{Binding Path=NomAlumno}" />
        <TextBlock Grid.Row="2" Grid.Column="0" Text="Dirección: "
                   Margin="7" />
```

```
        <TextBox Grid.Row="2" Grid.Column="1" Name="ctDirAlumno"
                 Margin="5" Text="{Binding Path=DirAlumno}" />
        <TextBlock Grid.Row="3" Grid.Column="0" Text="Estudios: "
                   Margin="7" />
        <TextBox Grid.Row="3" Grid.Column="1" Name="ctEstAlumno"
                 Margin="5" Text="{Binding Path=EstAlumno}" />
        <TextBlock Grid.Row="4" Grid.Column="0" Text="Beca: "
                   Margin="7" />
        <CheckBox Grid.Row="4" Grid.Column="1" Name="ctBecaAlumno"
                  Margin="5" IsChecked="{Binding Path=BecaAlumno}"/>
      </Grid>
      <Grid Grid.Row="1" >
        ...
        <TextBlock Grid.Row="0" Grid.Column="0"
                   Text="As. Obligatorias: " Margin="7,7,7,1" />
        <TextBlock Grid.Row="0" Grid.Column="1"
                   Text="As. Optativas: " Margin="7,7,7,1" />
        <ListBox Grid.Row="1" Name="lsListaObs"
                 Margin="5" SelectionMode="Extended" />
        <ListBox Grid.Row="1" Grid.Column="1" Name="lsListaOps"
                 Margin="5" SelectionMode="Extended" />
      </Grid>
      <StackPanel Grid.Row="2" Orientation="Horizontal"
                  HorizontalAlignment="Right ">
        <Button Name="btAceptar" Content="_Aceptar" Width="72"
                Margin="7" IsDefault="True"
                Click="btAceptar_Click" />
        <Button Name="btCancelar" Content="_Cancelar" Width="72"
                Margin="7" IsCancel="True" />
      </StackPanel>
    </Grid>
  </Border>
</Window>
```

Observe que las cajas de texto están enlazadas con las propiedades correspondientes del nuevo objeto *Alumno* que queremos añadir a la colección, y que las listas permiten realizar una selección múltiple.

Esta caja de diálogo será mostrada cuando el usuario haga clic en el botón "Añadir". Por lo tanto, como siguiente paso, añada este controlador a la clase *DlgInsertarBorrar* a la que pertenece dicho botón. ¿Qué proceso tiene que realizar dicho controlador cuando se ejecute?:

1. Crear un nuevo objeto *Alumno*, el que queremos añadir. Algunas propiedades de este objeto, *IdAlumno*, *NomAlumno*, *DirAlumno*, *EstAlumno* y *BecaAlumno*, se rellenarán automáticamente a través de los enlaces establecidos en el diseño anterior y otras, *ListaCoAsignaturas*, *ListaAsigsObs* y *ListaAsigsOps*, se rellenarán escribiendo el código adecuado.

2. Crear un objeto de la clase del diálogo *DlgAñadir*. Esta clase tendrá un constructor que permita pasar como argumento el nuevo objeto *Alumno* para que este diálogo pueda acceder a sus propiedades.

3. Mostrar el diálogo y esperar a que el usuario introduzca los datos y cierre el diálogo.

4. Si el diálogo se cerró haciendo clic en su botón "Aceptar", se verificará si ya hay un alumno con ese identificador en la colección. Si lo hay, se notifica y se abandona el proceso iniciado. Si no lo hay, se recuperarán las asignaturas seleccionadas por el usuario en las listas fijas y se asignarán al objeto *Alumno* (observe que cada elemento de las listas presenta tanto el código de la asignatura como el nombre de la misma); después se añadirá el objeto *Alumno* a la colección de alumnos de la aplicación.

5. Si se cerró haciendo clic en su botón "Cancelar", no se realizará ninguna acción.

El código que ejecute cada uno de los puntos descritos puede ser el siguiente:

```
Private Sub btAñadir_Click(sender As Object, e As RoutedEventArgs)
  Dim nuevo As Alumno = CrearNuevoAlumno()
  Dim dlg As New DlgAñadir(nuevo)
  If dlg.ShowDialog() = True Then
    ' Verificar si el IdAlumno existe en la colección
    Dim objBBDD As bbdd = TryCast(Me.FindResource("objBBDD"), bbdd)
    Dim id As Integer = Integer.Parse(dlg.ctIdAlumno.Text)
    If objBBDD.ObtenerAlumnoPorId(id) IsNot Nothing Then
      MessageBox.Show(String.Format("El Id {0} ya existe", id))
      Return
    End If
    ' Añadir las asignaturas seleccionadas
    AñadirAsigs(nuevo, dlg)
    ' Añadir el nuevo alumno a la colección
    objBBDD.Alumnos.Add(nuevo)
  End If
End Sub

Private Function CrearNuevoAlumno() As Alumno
  ' Construir la colección de asignaturas
  Dim listaAsigs As New ObservableCollection(Of Asignaturas)()
  listaAsigs.Add(New Asignaturas("Obligatorias",
    New ObservableCollection(Of Asignatura)()))
  listaAsigs.Add(New Asignaturas("Optativas",
    New ObservableCollection(Of Asignatura)()))
  ' Construir el objeto Alumno
  Dim nuevo As New Alumno()
  nuevo.ListaCoAsignaturas = listaAsigs
```

```
  Return nuevo
End Function

Private Sub AñadirAsigs(nuevo As Alumno, dlg As DlgAñadir)
  ' Añadir las asignaturas seleccionadas en las listas
  ' a las colecciones correspondientes
  Dim id As Integer
  Dim nombre As String
  For Each s As String In dlg.lsListaObs.SelectedItems
    id = Integer.Parse(s.Substring(0, 2))
    nombre = s.Substring(3)
    nuevo.ListaAsigsObs.Add(New Asignatura(id, nombre, -1))
  Next
  For Each s As String In dlg.lsListaOps.SelectedItems
    id = Integer.Parse(s.Substring(0, 2))
    nombre = s.Substring(3)
    nuevo.ListaAsigsOps.Add(New Asignatura(id, nombre, -1))
  Next
End Sub
```

Según lo explicado anteriormente, el código subyacente de la clase *DlgAñadir* será el siguiente:

```
Imports BaseDeDatos

Public Class DlgAñadir
  Public Sub New(nuevo As Alumno)
    InitializeComponent()
    LlenarListasAsigs()
    gridAlumno.DataContext = nuevo
  End Sub

  Private Sub LlenarListasAsigs()
    ' Obligatorias
    lsListaObs.Items.Add("11 Asignatura ob11")
    lsListaObs.Items.Add("12 Asignatura ob12")
    lsListaObs.Items.Add("13 Asignatura ob13")
    lsListaObs.Items.Add("14 Asignatura ob14")
    ' Optativas
    lsListaOps.Items.Add("61 Asignatura op61")
    lsListaOps.Items.Add("62 Asignatura op62")
  End Sub

  Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = True
  End Sub
End Class
```

Como dijimos anteriormente, esta clase tiene un constructor con un parámetro: el objeto *Alumno* nuevo que deseamos añadir a la colección de alumnos. Este constructor construye el diálogo invocando al método *InitializeComponent*, llena las listas con las asignaturas elegibles y asigna al contexto de datos de *gridAlumno* el nuevo objeto *Alumno* a cuyas propiedades hacen referencia los enlaces establecidos.

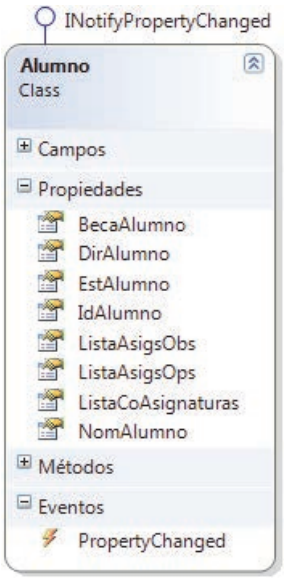
El controlador del botón "Aceptar" simplemente cerrará el diálogo.

Pruebe ahora la aplicación y observe los resultados. Podrá comprobar cómo los nuevos alumnos se añaden al final de la colección. Las notas las puede introducir cuando se visualiza la colección.

Validación

Según estudiamos en el capítulo *Introducción a WPF*, WPF incluye una clase **ExceptionValidationRule**, derivada de **ValidationRule**, que define una regla que permite comprobar las excepciones que se producen durante las actualizaciones de una propiedad del origen de enlace; esto es, un objeto de la misma puede ser añadido a la colección **ValidationRules** de un enlace, con el fin de capturar las excepciones producidas durante las actualizaciones de las propiedades del origen de datos del enlace, lo que permite a la interfaz del usuario mostrar el mensaje de error correspondiente si se lanza una excepción.

Pongamos nuestra atención en la clase *Alumno* y veamos las propiedades que expone en la figura siguiente:



Cuando diseñamos esta clase pensamos en que el usuario pudiera modificar las propiedades de la misma (de ahí que la clase implemente la interfaz **INotifyPropertyChanged**), por eso en el diseño del diálogo *DlgAñadir* hemos incluido controles **TextBox** cuya propiedad **Text** está enlazada a la propiedad correspondiente de un objeto *Alumno* (el que deseamos añadir). Dado que el usuario puede escribir el texto que desee en un **TextBox**, no podemos asegurar que el texto introducido en la caja de texto *ctIdAlumno* se pueda convertir en un valor del tipo **Integer**.

De forma predeterminada el enlace ya habilita una comprobación del tipo de dato introducido. Por ejemplo, si prueba a mostrar la caja de diálogo *DlgAñadir* y deja vacía la caja de texto correspondiente al identificador del alumno (un **Integer**), observará, cuando el cursor abandone dicha caja, que aparece un borde rojo, indicando que el valor introducido no es correcto. Lo mismo ocurrirá si introduce un valor que no se corresponde con un dato de tipo **Integer**:



En este escenario, puede usar **ExceptionValidationRule** para notificar errores de conversión de datos y, a continuación, mostrar dichos errores de conversión en la interfaz de usuario. Por ejemplo, el siguiente código XAML muestra las dos formas de hacerlo:

```
<TextBox Grid.Row="0" Grid.Column="1" Name="ctIdAlumno" Margin="5">
  <TextBox.Text>
    <Binding Path="IdAlumno" ConverterCulture="es-ES"
             UpdateSourceTrigger="PropertyChanged" >
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

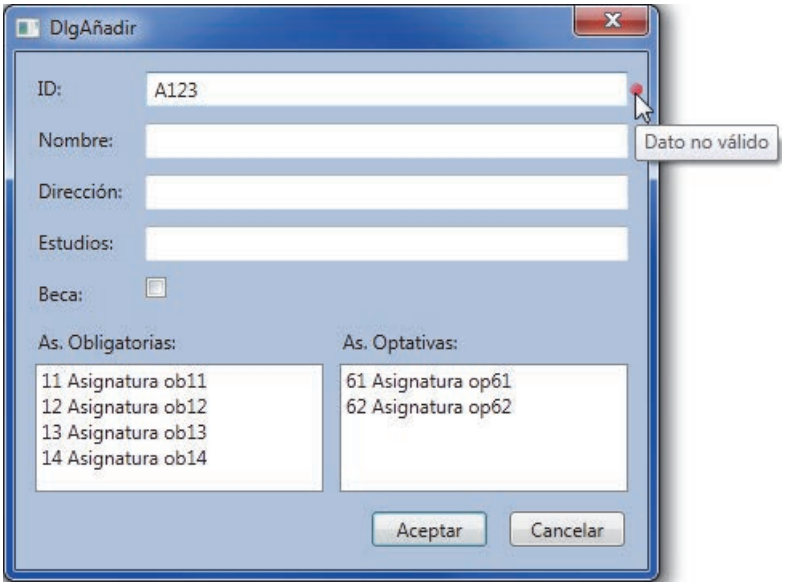
O bien,

```
<TextBox Grid.Row="0" Grid.Column="1" Name="ctIdAlumno" Margin="5"
         Text="{Binding Path=IdAlumno,
                        ConverterCulture=es-ES,
                        UpdateSourceTrigger=PropertyChanged,
                        ValidatesOnExceptions=True}"
  />
```

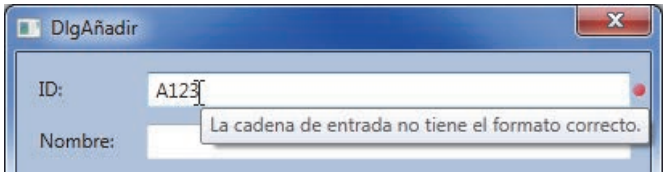
El código anterior muestra dos formas de agregar la regla **ExceptionValidationRule** a la colección **ValidationRules** de un enlace en XAML. La primera, que utiliza la sintaxis detallada de las propiedades como elementos, permite agregar la regla de forma explícita y la segunda, que utiliza la sintaxis abreviada, propiedades como atributos, establece la propiedad **ValidatesOnExceptions** del enenlace en **True**. Las dos formas asignan a la propiedad **UpdateSourceTrigger** del enlace el valor **PropertyChanged** para que la entrada se valide cada vez que la propiedad **Text** de la caja de texto reciba un valor nuevo, en lugar de esperar hasta que el control pierda el foco (acción predeterminada).

Visualización de los errores de validación

Cuando la caja de texto contiene un valor no válido, porque no se puede convertir en el tipo de datos esperado, podemos programar que aparezca un mensaje de error e incluso, añadir un pequeño icono rojo de error a la derecha del control para identificar más fácilmente dónde se ha producido el error, según muestra la figura siguiente.



Puede elegir una u otra forma de señalización, o bien dejar ambas, opción que hemos adoptado aquí:



Este comportamiento, según se indica a continuación, es muy fácil de implementar y personalizar (para más detalles, vea el apartado *Estilos y plantillas* en el capítulo *Introducción a WPF*). El código mostrado a continuación, primero implementa la plantilla que se aplica al **TextBox** cuando su contenido no es válido y a continuación, implementa el estilo que se aplicará al **TextBox** para que, además de utilizar la plantilla anterior, muestre los mensajes deseados:

```
<Window.Resources>
  <!-- Plantilla que se aplica a TextBox
  cuando su contenido no es válido -->
  <ControlTemplate x:Key="PlantillaErrorEnTextBox">
    <DockPanel>
      <Ellipse DockPanel.Dock="Right"
               Margin="1,0" Width="8" Height="8"
               ToolTip="Dato no válido" >
        <Ellipse.Fill>
          <LinearGradientBrush>
            <GradientStop Color="#11FF1133" Offset="0" />
            <GradientStop Color="#FFFF0000" Offset="1" />
          </LinearGradientBrush>
        </Ellipse.Fill>
      </Ellipse>
      <!-- Lugar que ocupará el TextBox -->
      <AdornedElementPlaceholder />
    </DockPanel>
  </ControlTemplate>
  <!-- Estilo que se aplica al TextBox -->
  <Style TargetType="TextBox" x:Key="EstiloTextBox">
    <Setter Property="Margin" Value="3" />
    <Setter Property="Validation.ErrorTemplate"
            Value="{StaticResource PlantillaErrorEnTextBox}" />
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="True">
        <Setter Property="ToolTip">
          <Setter.Value>
            <Binding Path="(Validation.Errors)[0].ErrorContent"
               RelativeSource="{x:Static RelativeSource.Self}" />
          </Setter.Value>
        </Setter>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

La clase estática **Validation** que hemos utilizado en este código permite establecer una relación entre un control y todos los errores de validación que contiene debido al uso de algunas propiedades adjuntas y métodos estáticos. Veamos cómo hemos utilizado esas propiedades adjuntas en XAML para especificar cómo la IGU debe presentar al usuario los errores de validación de la entrada del mismo.

Observe el objeto **Style** en el código anterior: aplica tres parámetros a una caja de texto. El primer **Setter** se aplica a la propiedad **Margin** de la caja de texto con la intención de proporcionar suficiente espacio para mostrar el icono de error a la derecha de la caja. El segundo **Setter** asigna la plantilla de tipo **ControlTemplate** utilizada para representar la caja de texto cuando ésta contiene datos no válidos; esto es, establece la propiedad adjunta **Validation.ErrorTemplate** con la plantilla que se utilizará cuando la clase **Validation** notifica que la caja de texto tiene uno o más errores de validación. Aquí es donde entra en escena el icono rojo de error, tal como se muestra en la figura anterior. El tercer **Setter** está condicionado por un desencadenador que supervisa la propiedad adjunta **Validation.HasError** de la caja de texto. Cuando la clase **Validation** establece la propiedad adjunta **HasError** en **True** para la caja de texto, el desencadenador de **Style** se activa y asigna información a la propiedad **ToolTip** de la caja de texto. El valor de la propiedad **ToolTip** está enlazado al mensaje de error correspondiente a la excepción que se lanza cuando el contenido de la caja de texto no se corresponde con el tipo de datos de la propiedad del origen con la que está enlazada.

Para aplicar el estilo al **TextBox** sólo hay que establecer su propiedad **Style** con el recurso *EstiloTextBox*:

```
<TextBox Grid.Row="0" Grid.Column="1" Name="ctIdAlumno" Margin="5"
         Style="{StaticResource EstiloTextBox}">
  <TextBox.Text>
    ...
  </TextBox.Text>
</TextBox>
```

Regla de validación personalizada

Continuando con la aplicación anterior, supongamos que los identificadores de los alumnos están entre los valores 111 y 999. Podemos verificar que el valor introducido está en este rango creando una regla de validación personalizada. Para ello, tenemos que definir una clase derivada de la clase **ValidationRule** y redefinir su método **Validate**. Según esto, agregue una carpeta *Validacion* al proyecto y añada a la misma una nueva clase *ReglaIdValido* derivada de **ValidationRule** que redefina el método **Validate**. Después, edite este método como se indica a continuación:

```
Class ReglaIdValido
  Inherits ValidationRule
  Private _idMin As Double
  Private _idMax As Double

  Public Property IdMin() As Double
    Get
      Return _idMin
```

```
    End Get
    Set(value As Double)
      _idMin = value
    End Set
  End Property

  Public Property IdMax() As Double
    Get
      Return _idMax
    End Get
    Set(value As Double)
      _idMax = value
    End Set
  End Property

  Public Overrides Function Validate(value As Object, cultureInfo _
  As System.Globalization.CultureInfo) As ValidationResult
    Dim id As Integer
    If Not Integer.TryParse(value.ToString(), id) Then
      Return New ValidationResult(False, "Identificador no válido.")
    End If

    If id < IdMin OrElse id > IdMax Then
      Return New ValidationResult(False, String.Format( _
        "Valor fuera del rango {0} a {1}", IdMin, IdMax))
    End If

    Return ValidationResult.ValidResult
  End Function
End Class
```

La clase *ReglaIdValido* proporciona dos propiedades públicas que definen el rango de identificadores válido y el método **Validate** que define dos condiciones para pasar la validación: una para verificar el formato del dato y otra para verificar si el dato está dentro del rango definido. Si el formato o el rango no son los adecuados, se construye y se devuelve un objeto **ValidationResult**. El constructor de **ValidationResult** tiene un primer parámetro que indica si el dato no es válido (**False**) o si es válido (**True**) y un segundo parámetro que da información acerca de lo ocurrido. Si todas las reglas pasan, se devuelve el valor de la propiedad **ValidationResult.ValidResult** que equivale a un objeto **ValidationResult** construido con los parámetros **True** y **Nothing**.

El proceso de validación tiene lugar durante la transferencia de un valor del destino al origen del enlace. Precisamente, el valor a validar se corresponde con el parámetro *value* del método **Validate** y sólo será asignado al origen del enlace si pasa todas las reglas de validación; en otro caso, el objeto origen permanecerá con el dato inicial, esto es, la propiedad correspondiente no se modificará.

Para aplicar esta regla de validación procedemos igual que hicimos con la regla **ExceptionValidationRule**:

```
<TextBox Grid.Row="0" Grid.Column="1" Name="ctIdAlumno" Margin="5"
         Style="{StaticResource EstiloTextBox}">
  <TextBox.Text>
    <Binding Path="IdAlumno" ConverterCulture="es-ES"
             UpdateSourceTrigger="PropertyChanged" >
      <Binding.ValidationRules>
        <local:ReglaIdValido IdMin="111" IdMax="999" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Recuerde que el prefijo *local* hace referencia al espacio de nombres *EnlaceDeDatosObjetos* al que pertenece la clase *ReglaIdValido*.

```
<Window x:Class="DlgAñadir"
        ...
        xmlns:local="clr-namespace:EnlaceDeDatosObjetos"
        ...
```

Análogamente, puede añadir otra regla de validación para que las otras tres cajas de texto no queden vacías o no contengan un valor numérico:

```
Class ReglaCajaTextoVacia
  Inherits ValidationRule
  Public Overrides Function Validate(value As Object, cultureInfo _
  As System.Globalization.CultureInfo) As ValidationResult
    Dim n As Double
    If Double.TryParse(value.ToString(), n) Then
      Return New ValidationResult(False,
        "No puede ser un dato numérico")
    End If

    Dim contenido As String = TryCast(value, String)
    If contenido.Length = 0 Then
      Return New ValidationResult(False, String.Format( _
        "La cadena de entrada está vacía"))
    End If

    Return ValidationResult.ValidResult
  End Function
End Class
```

Para aplicar esta regla de validación procedemos igual que en los casos anteriores. Por ejemplo:

```
<TextBox Grid.Row="1" Grid.Column="1" Name="ctNomAlumno" Margin="5"
         Style="{StaticResource EstiloTextBox}">
  <TextBox.Text>
    <Binding Path="NomAlumno" ConverterCulture="es-ES">
      <Binding.ValidationRules>
        <local:ReglaCajaTextoVacia />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

En este caso, para utilizar esta regla, hemos preferido no modificar el valor predeterminado de la propiedad **UpdateSourceTrigger** que es validar cuando el control pierda el foco.

Permanecer en la caja de diálogo si hay errores

En los apartados anteriores hemos visto cómo establecer las reglas de validación para validar la entrada, pero el usuario podía no hacer caso de las advertencias que se le muestran en la IGU y hacer clic en el botón "Aceptar" del diálogo para continuar, con lo que se añadiría un nuevo elemento a la colección, probablemente no deseado. Para evitar este comportamiento y obligar al usuario a introducir todos los datos exigidos de forma correcta o a pulsar el botón "Cancelar", modifique el controlador del botón "Aceptar" del diálogo *DlgAñadir* como se indica a continuación:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  If Validation.GetHasError(ctIdAlumno) OrElse
    Validation.GetHasError(ctNomAlumno) OrElse
    Validation.GetHasError(ctDirAlumno) OrElse
    Validation.GetHasError(ctEstAlumno) OrElse
    lsListaObs.SelectedItems.Count = 0 Then
    MessageBox.Show("Error: valor no valido o no selección asigs. obs.")
    Return
  Else
    DialogResult = True
  End If
End Sub
```

Cuando el contenido de una caja de texto no haya cumplido todas las reglas de validación, esto es, el dato no sea válido, la clase **Validation** establece su propiedad asociada **HasError** en **True**, valor que se puede comprobar con el método **Shared GetHasError** de la clase **Validation**. Por eso, la condición para salir del diálogo *DlgAñadir* haciendo clic en el botón "Aceptar" es que todos los datos introducidos sean correctos y que al menos se haya seleccionado una asignatura obligatoria. En otro caso, el usuario sólo podrá abandonar el diálogo haciendo clic en el botón "Cancelar", no añadiéndose así ningún elemento a la colección.

Grupos de enlaces

En el ejemplo anterior, la aplicación rellena un objeto de tipo *Alumno* que tiene, entre otras, las propiedades *NomAlumno*, *DirAlumno* y *EstAlumno* con los valores que el usuario proporcionó. Estos valores han sido validados aplicando independientemente a cada caja la misma regla de validación. Nos preguntamos, ¿hay alguna forma de aplicar esa regla de validación conjuntamente a las tres cajas? La respuesta es sí, utilizando un objeto **BindingGroup**.

Un objeto **BindingGroup** permite validar y actualizar varios enlaces de forma conjunta. Para ello, hay que establecer la propiedad **BindingGroup** en un elemento **FrameworkElement** o **FrameworkContentElement** con el objeto **BindingGroup** que especifique las reglas de validación, para que los elementos secundarios puedan heredarlo.

Aplicando lo expuesto a nuestro ejemplo, tendremos que incluir las tres cajas de texto que queremos validar conjuntamente en un panel (por ejemplo, en un objeto **StackPanel**) y establecer su propiedad **BindingGroup**:

```
<StackPanel Grid.Row="1" Grid.Column="1" Grid.RowSpan="3"
            Name="spCajasTexto">
  <StackPanel.BindingGroup>
    <BindingGroup x:Name="bgCajasTexto">
      <BindingGroup.ValidationRules>
        <local:ReglaCajasTextoVacias />
      </BindingGroup.ValidationRules>
    </BindingGroup>
  </StackPanel.BindingGroup>

  <TextBox Name="ctNomAlumno" Margin="5"
           Text="{Binding Path=NomAlumno}" >
  </TextBox>
  <TextBox Name="ctDirAlumno" Margin="5"
           Text="{Binding Path=DirAlumno}">
  </TextBox>
  <TextBox Name="ctEstAlumno" Margin="5"
           Text="{Binding Path=EstAlumno}">
  </TextBox>
</StackPanel>
```

En el código anterior podemos ver que hemos incluido en un panel los tres controles **TextBox**, cada uno de los cuales mantiene un enlace a datos con la propiedad correspondiente del objeto *Alumno*. De esta forma podemos usar una regla **ValidationRule**, en nuestro caso *ReglaCajasTextoVacias*, en el objeto **BindingGroup** para validar esas propiedades; este objeto, en este caso, agrupará los tres enlaces que compartirán la regla de validación.

La pertenencia de un enlace **Binding** a un grupo de enlaces **BindingGroup** puede hacerse a través de la propiedad **BindingGroupName** del enlace; si esta propiedad vale **Nothing** el enlace no pertenece a ningún grupo. Por ejemplo:

```
<TextBox Name="ctNomAlumno" Margin="5"
  Text="{Binding Path=NomAlumno, BindingGroupName=bgCajasTexto}" >
```

Si no se especifica la propiedad **BindingGroupName**, el enlace pertenece al **BindingGroup** sólo si el origen del enlace y la propiedad **DataContext** del elemento primario que tiene el **BindingGroup** hacen referencia al mismo objeto (en nuestro caso hacen referencia al objeto *Alumno*).

Cuando se invoque al método **Validate** de la regla de validación establecida, este método recibirá en su primer parámetro el objeto **BindingGroup**. Entonces, para obtener los valores propuestos por el usuario para el objeto origen y verificarlos, podemos utilizar el método **GetValue** o **TryGetValue** y la propiedad **Items**.

La propiedad **Items** nos da acceso a los orígenes de los objetos **Binding** del objeto **BindingGroup** y el método **GetValue** devuelve el valor propuesto por el usuario para la propiedad especificada del objeto especificado.

Según lo expuesto, la regla de dependencia *ReglaCajasTextoVacias* podríamos escribirla así:

```
Class ReglaCajasTextoVacias
  Inherits ValidationRule
  Public Overrides Function Validate(value As Object, cultureInfo _
  As System.Globalization.CultureInfo) As ValidationResult
    Dim bindingGroup As BindingGroup = DirectCast(value, BindingGroup)
    ' Sólo hay un elemento en la colección Items.
    Dim al As Alumno = TryCast(bindingGroup.Items(0), Alumno)
    ' al contiene los valores originales
    Dim n As Double
    ' Verificar los valores nuevos
    Dim nom As String = bindingGroup.GetValue(al, "NomAlumno").ToString()
    Dim dir As String = bindingGroup.GetValue(al, "DirAlumno").ToString()
    Dim est As String = bindingGroup.GetValue(al, "EstAlumno").ToString()
    If Double.TryParse(nom, n) OrElse
       Double.TryParse(dir, n) OrElse
       Double.TryParse(est, n) Then
      Return New ValidationResult(False,
        "No puede ser un dato numérico")
    End If

    If nom.Length = 0 OrElse dir.Length = 0 OrElse
       est.Length = 0 Then
```

```
      Return New ValidationResult(False,
        String.Format("La cadena de entrada está vacía"))
    End If

    Return ValidationResult.ValidResult
  End Function
End Class
```

Cuando un enlace forma parte de un **BindingGroup**, el origen del enlace no se actualiza hasta que se llame al método **UpdateSources** o **CommitEdit** del objeto **BindingGroup**, a menos que se establezca explícitamente la propiedad **UpdateSourceTrigger**.

Según lo expuesto, el controlador del botón "Aceptar" lo modificaríamos como se muestra a continuación:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  bgCajasTexto.CommitEdit()
  If Validation.GetHasError(ctIdAlumno) OrElse
     Validation.GetHasError(spCajasTexto) OrElse
     lsListaObs.SelectedItems.Count = 0 Then
    MessageBox.Show("Error: valor no valido o no selección asigs. obs.")
    Return
  Else
    DialogResult = True
  End If
End Sub
```

En este método observamos dos cambios: la llamada al método **CommitEdit** del objeto *bgCajasTexto* de tipo **BindingGroup**, para actualizar los orígenes de los enlaces que agrupa, y la llamada a **GetHasError** para verificar el valor de la propiedad **HasError** asociada con el control *spCajasTexto* de tipo **StackPanel** que define el grupo de enlaces. Si todos los valores de los destinos de los enlaces del grupo cumplen lo especificado por la regla de validación, esta propiedad tendrá el valor **False**; en otro caso, tendrá el valor **True**.

DataGrid

Este control, según comentamos en el capítulo anterior, muestra la información en filas y columnas. Para enlazar este control con una colección de objetos, basta asignar dicha colección de objetos a su propiedad **ItemsSource**, colección que deberá implementar la interfaz **IEnumerable**, como sucede con **ObservableCollection**. Cada fila se corresponde con un objeto particular, y cada columna con una propiedad de ese objeto. Además, el tipo de fila que integra incluye también una sección de detalles desplegable que se puede utilizar para mostrar contenido adicional debajo de los valores de celda.

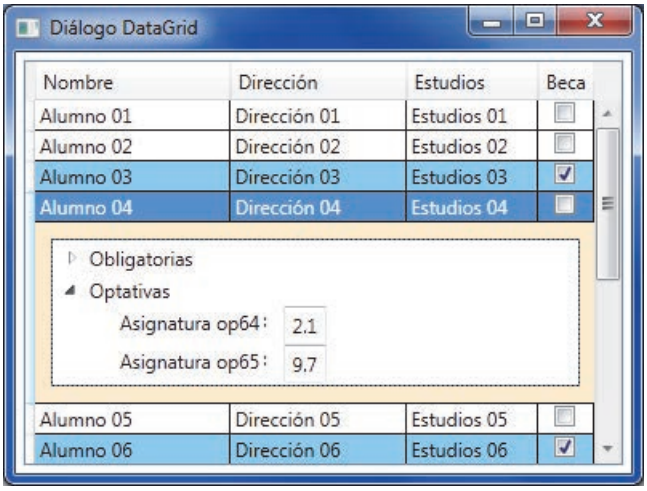
Una de las características importantes del **DataGrid** es su soporte para la edición. Una celda cambia a modo de edición cuando el usuario hace doble clic sobre ella (el primer clic para seleccionarla y el segundo para pasar a modo de edición). La edición será efectiva cuando se cambie el cursor a otra celda. También se puede abandonar la edición de la celda actual pulsando la tecla *Esc*. Sin embargo, podemos restringir esta capacidad de varias maneras: por medio de la propiedad **DataGrid.IsReadOnly**, cuando esta propiedad vale **True** el usuario no puede editar nada, o por medio de la propiedad **DataGridColumn.IsReadOnly**, cuando esta propiedad vale **True** el usuario no puede editar la columna correspondiente.

Para que la IGU se actualice automáticamente cuando se añadan, se eliminen o se modifiquen elementos de la colección origen de los datos, dicha colección debe implementar la interfaz **INotifyCollectionChanged**. Lógicamente, los objetos de la colección deben implementar también esa interfaz.

De manera predeterminada, el control **DataGrid** genera automáticamente las columnas a partir de la información del origen de datos. Si no queremos que esto suceda así, hay que asignar a la propiedad **AutoGenerateColumns** el valor **False**.

Para más detalles, vea el apartado *DataGrid* en el capítulo *Controles y cajas de diálogo*.

Por ejemplo, vamos a añadir a nuestra aplicación *EnlaceDeDatosObjetos* un nuevo diálogo *DlgDataGrid* como el que muestra la figura siguiente:



El origen de datos, como en los ejemplos anteriores, es el objeto *objBBDD* de la clase *bbdd* definido en los recursos de la aplicación:

```
<Application x:Class="Application"
```

```
  ...
  xmlns:local="clr-namespace:BaseDeDatos;assembly=BaseDeDatos">
  <Application.Resources>
    <local:bbdd x:Key="objBBDD" />
  </Application.Resources>
</Application>
```

Los datos mostrados son proporcionados por el contexto de datos del **Grid**, como se puede ver en el código XAML siguiente, y la colección *Alumnos* es asignada al **DataGrid** a través de su propiedad **ItemsSource**:

```
<Window x:Class="DlgDataGrid"
  ...
  <Window.Resources>

  </Window.Resources>

  <Grid DataContext="{StaticResource objBBDD}" >
    <DataGrid Name="dgAlumnos" Margin="5"
              AutoGenerateColumns="False"
              ItemsSource="{Binding Path=Alumnos}">
      <DataGrid.Columns>
        ...
      </DataGrid.Columns>
    </DataGrid>
  </Grid>
</Window>
```

Columnas del DataGrid

Los tipos de columnas que integra el **DataGrid** dependen del tipo de las propiedades de los objetos vinculados. En nuestro caso integra columnas de cajas de texto (**DataGridTextColumn**), para mostrar las propiedades *NomAlumno*, *DirAlumno* y *EstAlumno* del objeto de la clase *Alumno* y una columna de casilla de verificación (**DataGridCheckBoxColumn**) para mostrar la propiedad *BecaAlumno*. Dichas columnas son añadidas a la colección **DataGrid.Columns**. El siguiente código XAML define estas columnas:

```
<DataGrid.Columns>
  <DataGridTextColumn Header="Nombre"
                      Binding="{Binding Path=NomAlumno}"/>
  <DataGridTextColumn Header="Dirección"
                      Binding="{Binding Path=DirAlumno}"/>
  <DataGridTextColumn Header="Estudios"
                      Binding="{Binding Path=EstAlumno}"/>
  <DataGridCheckBoxColumn Header="Beca"
                      Binding="{Binding Path=BecaAlumno}"/>
</DataGrid.Columns>
```

El tamaño de las columnas se ajusta automáticamente en función del tamaño de la cabecera o del contenido de las celdas, también se puede ajustar manualmente, las filas se pueden ordenar por cualquier columna haciendo clic en su cabecera y las celdas se pueden editar.

También pondremos la propiedad **AutoGenerateColumns** del **DataGrid** a **False** para que las columnas no especificadas del objeto *Alumno* no se generen automáticamente a partir del contenido referenciado por su propiedad **ItemsSource**.

Inmovilizar columnas

La propiedad **FrozenColumnCount** del **DataGrid** permite fijar el número de columnas del **DataGrid** que se desea inmovilizar. Por ejemplo:

```
<DataGrid Name="dgAlumnos" FrozenColumnCount="2"...>
```

Las columnas inmovilizadas se muestran siempre y se corresponden con las columnas situadas más a la izquierda en el orden de presentación. El código del ejemplo anterior inmovilizará las dos columnas de la izquierda.

Filas del DataGrid

En muchos casos, es útil señalar filas que contienen datos específicos. Por ejemplo, observando la figura anterior vemos que están resaltadas las filas de los alumnos que tienen beca. Esta forma de proceder requiere aplicar formato a las filas, para lo cual hay que controlar el evento **DataGrid.LoadingRow** que se genera una vez para cada fila cuando ésta aparece en pantalla. Por lo tanto, controlando este evento tendremos acceso al objeto origen de los datos de la fila actual, lo que permitirá escribir código a partir de esos datos, y también tenemos acceso al objeto **DataGridRow** de la fila, lo que permitirá dar formato a la misma, por ejemplo, para aplicarle un color diferente. La ventaja de este enfoque es que no se actúa sobre todas las filas de la rejilla, sino sólo sobre las visibles y, además, el **DataGrid** reutiliza los mismos objetos **DataGridRow** para mostrar las nuevas filas mientras éstas se desplazan, lo que reduce el consumo de memoria.

Esta manera de trabajar del **DataGrid** puede conducir a que el control pueda cargar datos en un **DataGridRow** ya con formato, y que estos datos no se correspondan con ese formato. Para evitar esta situación, se aconseja restaurar el **DataGridRow** a su estado inicial cada vez que éste recargue datos.

Como ejemplo vamos a programar que el diálogo *DataGrid* mostrado anteriormente muestre los alumnos con beca en otro color según mostraba la figura anterior. Para ello, hay que interceptar el evento **LoadingRow** del **DataGrid**:

```
<DataGrid Name="dgAlumnos" Margin="5"
          ItemsSource="{Binding Path=Alumnos}"
          LoadingRow="dgAlumnos_LoadingRow">
  <DataGrid.Columns>

  </DataGrid.Columns>
</DataGrid>
```

y verificar por cada fila que se muestra, objeto **DataGridRow**, si el dato *beca* es **True** o **False**; si es **True**, aplicamos el nuevo formato a la fila (el nuevo color) y si es **False**, restauramos el color predeterminado para asegurarnos de que las filas recargadas con formato son restauradas.

```
Public Class DlgDataGrid
  Private azulClaro As New SolidColorBrush(Colors.LightSkyBlue)
  Private colorPredeterminado As New SolidColorBrush(Colors.White)

  Private Sub dgAlumnos_LoadingRow(sender As Object,
  e As DataGridRowEventArgs)
    ' Comprobar los datos de esta fila
    Dim filaActual As Alumno = TryCast(e.Row.DataContext, Alumno)
    If filaActual Is Nothing Then Return
    If filaActual.BecaAlumno = True Then
      e.Row.Background = azulClaro
    Else
      ' Restaurar el color predeterminado para asegurar que
      ' las filas recargadas con formato son restauradas
      e.Row.Background = colorPredeterminado
    End If
  End Sub
End Class
```

Observe que la propiedad **Row** del objeto **DataGridRowEventArgs** de información del evento nos da acceso al objeto **DataGridRow** correspondiente a la fila para la que se produce el evento. Como este objeto participa de un enlace de datos, será el contexto de datos del mismo, propiedad **DataContext**, el que proporcione los datos del origen, que, en nuestro ejemplo, se corresponde con un objeto *Alumno* cuyas propiedades nos permitirán acceder al contenido deseado: propiedad *BecaAlumno*.

Selección de celdas

Cuando el usuario hace clic en una celda se selecciona toda la fila del **DataGrid**; esto es fijado por la propiedad **SelectionUnit** que tiene como valor predeterminado *FullRow* (los otros valores son *Cell* y *CellOrRowHeader*). También se pueden seleccionar múltiples filas, que es lo que indica el valor predeterminado *Extended* (el otro valor es *Single*) de la propiedad **SelectionMode**. Dependiendo del valor

de **SelectionUnit**, la propiedad **SelectedCells** del **DataGrid** contendrá una colección u otra de celdas seleccionadas.

Cuando se selecciona una celda, se produce el evento **DataGridCell.Selected**, se establece la propiedad **DataGridCell.IsSelected** en **True** y se añade la celda a la colección **SelectedCells**. Análogamente, cuando se selecciona una fila, se produce el evento **DataGridRow.Selected**, se establece la propiedad **DataGridRow.IsSelected** en **True** y se añade la fila a la colección **SelectedItems**. Y cuando la selección cambia, se generan los eventos **SelectedCellsChanged** (cambia la colección **SelectedCells**) y **SelectionChanged** (cambio de fila).

Por ejemplo, cuando modificamos el valor de una celda enlazada con la propiedad *BecaAlumno* esa fila debería cambiar de color, pero sólo lo hará cuando se recargue de nuevo (evento **LoadingRow**). Si queremos que ese cambio ocurra después de que la propiedad haya cambiado de valor, podemos interceptar el evento **SelectedCellsChanged** (el usuario modifica la celda y cambia a otra fila para que se genere este evento después de la modificación) y responder al mismo así:

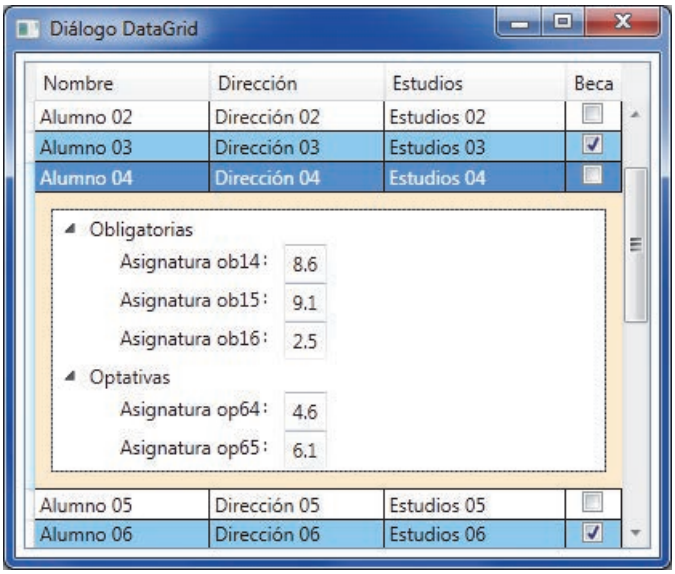
```
Private Sub dgAlumnos_SelectedCellsChanged(sender As Object,
e As SelectedCellsChangedEventArgs)
  CollectionViewSource.GetDefaultView( _
    dgAlumnos.ItemsSource).Refresh()
End Sub
```

Lo que hace el método **Refresh** de **ICollectionView** es volver a crear la vista con los valores actuales de la colección.

Detalles de las filas

Anteriormente dijimos que cada fila de un **DataGrid** se puede expandir para mostrar una sección de detalles. La sección de detalles, que especifica el aspecto y los datos que va a mostrar esta sección, la define la plantilla **DataTemplate** que se implementa en la sección de detalles de la fila definida por la propiedad **RowDetailsTemplate**.

Como ejemplo, veamos cómo implementamos la sección de detalles mostrada por la figura siguiente:



En esta figura observamos que la sección de detalles se corresponde con un **TreeView** que muestra las asignaturas, con sus notas, del alumno seleccionado. El diseño de este objeto ya fue explicado anteriormente, por lo tanto sólo queda ver qué hacer para que pase a ser la sección de detalles de las filas del **DataGrid**. Esto es lo que muestra el código XAML siguiente, en el que se observa que el diseño del **TreeView** está definido por el recurso *kAsignaturas*.

```
<Window x:Class="DlgDataGrid"
  ...
  <Window.Resources>
    <!-- PLANTILLA ASIGNATURA -->
    <DataTemplate x:Key="kAsignatura">
      ...
    </DataTemplate>
    <!-- PLANTILLA ASIGNATURAS -->
    <HierarchicalDataTemplate x:Key="kAsignaturas"
      ...
    </HierarchicalDataTemplate>
  </Window.Resources>

  <Grid DataContext="{StaticResource objBBDD}" >
    <DataGrid Name="dgAlumnos" Margin="5"
      ...
      <!-- Sección de detalles de la fila -->
      <DataGrid.RowDetailsTemplate>
        <DataTemplate>
          <Border BorderThickness="0"
                  Background="BlanchedAlmond" Padding="5">
            <TreeView  Name="trvAlumnos" Margin="5"
```

```
                ItemsSource="{Binding Path=ListaCoAsignaturas}"
                ItemTemplate="{StaticResource kAsignaturas}">
            </TreeView>
          </Border>
        </DataTemplate>
      </DataGrid.RowDetailsTemplate>
    </DataGrid>
  </Grid>
</Window>
```

Filtrado, agrupación y ordenación

Para agrupar, ordenar y filtrar datos en un **DataGrid** proceda sobre la vista, según explicamos en el apartado *Vistas de colección de objetos* anteriormente en este mismo capítulo.

Validación

El control **DataGrid** permite validar los datos a nivel de celda, se validan propiedades individuales de un objeto de datos enlazado, o a nivel de fila, se validan objetos de datos completos. Para proporcionar una validación personalizada hay que establecer la plantilla **RowValidationErrorTemplate** y crear las reglas de validación personalizadas, según estudiamos anteriormente:

```
<DataGrid.RowValidationErrorTemplate>
  <ControlTemplate>
    ...
  </ControlTemplate>
</DataGrid.RowValidationErrorTemplate>
```

En el capítulo siguiente veremos un ejemplo.

RESUMEN

Este capítulo empieza haciendo un breve resumen de los elementos fundamentales que intervienen en un enlace a datos, para después continuar haciendo una exposición sobre las colecciones de objetos, ya que son los orígenes de datos utilizados habitualmente por un enlace, explicando las características que deben tener para que cuando se utilicen implícitamente se produzca una actualización automática de la interfaz de usuario. La interfaz de usuario muestra los datos de la colección a través de una o más vistas de colección, por eso a continuación se explica cómo se trabaja con este tipo de objetos. La forma en la que se muestran los datos es otro aspecto importante, por eso se explica también cómo definir y utilizar plantillas de datos.

El siguiente apartado que trata este capítulo habla sobre las fuentes de datos XML. Partiendo de una colección de datos en XML, se expone cómo acceder y mostrar datos jerárquicos, cómo incrustar los datos XML en el propio código XAML, para después centrarse en las operaciones que se pueden realizar sobre un documento XML como son obtener la vista de la colección, obtener el elemento actualmente seleccionado en la vista, navegar por los elementos de la vista, ordenarlos, filtrarlos y agruparlos. También se explica cómo utilizar un origen de datos XML por medio de un proveedor de datos XML y sin proveedor de datos. Para terminar este tema sobre fuentes de datos XML, se muestra cómo implementar un diseño maestro-detalle tan presente en la presentación de datos.

A continuación se entra en el apartado de las colecciones de objetos del CLR (clases de objetos convencionales). Para tratar este tema, se simula una base de datos de la que se obtiene una colección de objetos jerárquica a imagen y semejanza de la colección de datos XML utilizada anteriormente. A partir de esta colección de objetos se vuelven a tratar, desde el punto de vista de colecciones de objetos, los conceptos explicados cuando la fuente de datos era XML; esto es, se expone cómo utilizar una colección jerárquica de objetos a través de un proveedor de datos y sin proveedor de datos, y también se explica cómo obtener la vista de la colección y cómo realizar sobre la misma las operaciones más frecuentes: acceso al elemento actual, navegación, ordenación, filtrado y agrupación de objetos. Después se muestra el diseño maestro-detalle y se comenta la importancia de la virtualización cuando se trabaja con grandes cantidades de datos.

A continuación, se expone cómo implementar un formulario para solicitar datos al usuario con el fin de actualizar el origen de datos. Lógicamente, este proceso conlleva otros como la validación de los datos propuestos por el usuario, la visualización de los errores de validación y la implementación de reglas de validación personalizadas.

Finalmente, se expone un diseño para mostrar cómo utilizar un **DataGrid** cuya fuente de datos es una colección de objetos CLR.

CAPÍTULO 6

© F.J.Ceballos/RA-MA

ACCESO A UNA BASE DE DATOS

Una base de datos es una colección de datos y un conjunto de programas para acceder a los mismos. Los datos están clasificados y estructurados y son guardados en uno o varios ficheros, pero referenciados como si de un único fichero se tratara. Para crear y manipular bases de datos relacionales, objetivo de este capítulo, existen en el mercado varios sistemas administradores de bases de datos; por ejemplo, *SQL Server*, *Access*, *Oracle* y *DB2*. Otros sistemas administradores de bases de datos de interés y de libre distribución son *PostgreSQL* y *MySQL*.

Los datos de una base de datos relacional se almacenan en tablas lógicamente relacionadas entre sí utilizando campos clave comunes. A su vez, cada tabla dispone los datos en *filas* y *columnas*. Por ejemplo, piense en el listín de teléfonos. Los datos relativos a un teléfono (nombre, dirección, teléfono, etc.) son *columnas* que agrupamos en una *fila*. El conjunto de todas las *filas* de todos los teléfonos forma una tabla de la base de datos.

Nombre	*Dirección*	*Teléfono*
Aguado Rodríguez, Jesús	Las Ramblas 3, Barcelona	932345678
Cuesta Suñer, Ana María	Mayor 22, Madrid	918765432
…	…	…

Como se puede observar, una tabla es una colección de datos presentada en forma de una matriz bidimensional, donde las filas reciben también el nombre de *tuplas* o *registros* y las columnas de *campos*.

Los usuarios de un sistema administrador de bases de datos pueden realizar sobre una determinada base operaciones como insertar, recuperar, modificar y

eliminar datos, así como añadir nuevas tablas o eliminarlas. Estas operaciones se expresan generalmente en un lenguaje denominado SQL.

SQL

SQL es el lenguaje estándar para interactuar con bases de datos relacionales y es soportado prácticamente por todos los sistemas administradores de bases de datos actuales. En él, las unidades básicas son *tablas*, *columnas* y *filas*. La *tabla* proporciona una forma simple de relacionar los datos que componen la misma, una *columna* representa un dato presente en la tabla, mientras que una *fila* representa un registro o entrada de la tabla.

Este apartado introducirá al lector que no conoce SQL en las operaciones más comunes que este lenguaje proporciona para acceso a bases de datos. SQL incluye operaciones tanto de definición, por ejemplo CREATE, como de manipulación de datos, por ejemplo INSERT, UPDATE, DELETE y SELECT.

Crear una base de datos

Para crear una base de datos, SQL proporciona la sentencia CREATE DATABASE cuya sintaxis es:

```
CREATE DATABASE <base de datos>
```

Esta sentencia especifica el nombre de la base de datos que se desea crear. Cuando desee eliminarla, ejecute la sentencia:

```
DROP DATABASE <base de datos>
```

Crear una tabla

Para crear una tabla, SQL proporciona la sentencia CREATE TABLE. Esta sentencia especifica el nombre de la tabla, los nombres y tipos de las columnas de la tabla y las claves primaria y externa de esa tabla (también llamada extranjera, en el sentido de que es importada de otra tabla). Su sintaxis es la siguiente:

```
CREATE TABLE <tabla>(<columna 1> [,<columna 2>]...)
```

donde *columna n* se formula según la sintaxis siguiente:

```
<columna n> <tipo de dato> [DEFAULT <expresión>]
    [<constante 1> [<constante 2>]...]
```

Algunos de los tipos de datos más utilizados son los siguientes:

Tipo SQL	*Tipo SQL de .NET Framework*	*Tipo MS Access*
INTEGER	SqlInt32	Número entero largo
REAL	SqlSingle	Número simple
FLOAT	SqlDouble	Número doble
CHAR	SqlString	Texto
VARCHAR	SqlString	Texto
BINARY	SqlBinary	Binario
DATE	SqlDateTime	Fecha/hora

La cláusula DEFAULT permite especificar un valor por omisión para la columna y, opcionalmente, para indicar la forma o característica de cada columna, se pueden utilizar las constantes: NOT NULL (no se permiten valores nulos: NULL), UNIQUE o PRIMARY KEY.

La cláusula PRIMARY KEY se utiliza para definir la columna como clave principal de la tabla. Esto supone que la columna no puede contener valores nulos ni duplicados; es decir, que dos filas no pueden tener el mismo valor en esa columna. Una tabla puede contener una sola restricción PRIMARY KEY.

La cláusula UNIQUE indica que la columna no permite valores duplicados; es decir, que dos filas no pueden tener el mismo valor en esa columna. Una tabla puede contener varias restricciones UNIQUE. Se suele emplear para que el propio sistema compruebe que no se añaden valores que ya existen.

El ejemplo que se muestra a continuación crea la tabla *telefonos*, en la base de datos con la que estemos trabajando, con las columnas *nombre*, *direccion*, *telefono* y *observaciones* de los tipos especificados. La columna *telefono* es la clave principal; esto implica que en esa columna todos los valores tienen que ser diferentes y no nulos. El resto de las columnas, excepto *observaciones*, tampoco permiten valores nulos:

```
CREATE TABLE telefonos(
                nombre        VARCHAR(30) NOT NULL,
                direccion     VARCHAR(30) NOT NULL,
                telefono      VARCHAR(12) PRIMARY KEY NOT NULL,
                observaciones VARCHAR(240)
  )
```

La diferencia entre los tipos *CHAR (n)* y *VARCHAR (n)* es que en el primer caso, el campo se rellena con espacios hasta *n* caracteres (longitud fija) y en el segundo no (longitud variable).

Escribir datos en la tabla

Para escribir datos en una tabla, SQL proporciona la sentencia INSERT. Esta sentencia agrega una o más filas nuevas a una tabla. Su sintaxis, de forma simplificada, es la siguiente:

```
INSERT [INTO] <tabla> [(<columna 1>[,<columna 2>]...)]
                      VALUES (<expresión 1>[,<expresión 2>]...),...

INSERT [INTO] ... SELECT ... FROM ...
```

donde *tabla* es el nombre de la tabla en la que se desean insertar las filas, argumento que va seguido por una lista con los nombres de las columnas que van a recibir los datos especificados por la lista de valores que siguen a la cláusula VALUES. Las columnas no especificadas en la lista reciben el valor NULL, si lo permiten, o el valor predeterminado, si se especificó. Si todas las columnas reciben datos, se puede omitir la lista con los nombres de las columnas.

Con respecto al segundo formato, un poco más adelante veremos la sentencia SELECT.

El ejemplo que se muestra a continuación añade a la tabla *telefonos* una nueva fila con los valores de las columnas especificados:

```
INSERT INTO telefonos
       VALUES ('Leticia Aguirre Soriano','Madrid',
               '912345671,'Ninguna')
```

Modificar datos de una tabla

Para modificar datos en una tabla, SQL proporciona la sentencia UPDATE. Esta sentencia puede cambiar los valores de filas individuales, grupos de filas o todas las filas de una tabla. Su sintaxis es la siguiente:

```
UPDATE <tabla>
    SET <columna 1 = (<expresión 1> | NULL)
        [, <columna 2 = (<expresión 2> | NULL)]...
    WHERE <condición de búsqueda>
```

La cláusula SET contiene una lista separada por comas de las columnas que deben actualizarse y el nuevo valor de cada columna. El valor suministrado por las expresiones incluye elementos tales como constantes, valores seleccionados de una columna de otra tabla, o valores calculados por una expresión compleja. Y la

cláusula WHERE especifica la condición de búsqueda que define la fila de la tabla cuyas columnas se desea modificar.

El ejemplo que se muestra a continuación modifica en la tabla *telefonos* la dirección de la persona cuyo teléfono se especifica:

```
UPDATE telefonos
    SET direccion='Triana, Sevilla'
    WHERE telefono='91234567'
```

Borrar registros de una tabla

Para borrar registros en una tabla, SQL proporciona la sentencia DELETE. Esta sentencia quita una o varias filas de una tabla. Una forma simplificada de la sintaxis de DELETE es:

```
DELETE FROM <tabla> WHERE <condición de búsqueda>
```

El argumento *tabla* nombra la tabla de la que se van a eliminar las filas. Se eliminan todas las filas que reúnan los requisitos de la condición de búsqueda de la cláusula WHERE. Si no se especifica una cláusula WHERE, se eliminan todas las filas de la tabla.

Cualquier tabla de la que se hayan quitado todas las filas sigue permaneciendo en la base de datos. La instrucción DELETE sólo elimina filas de la tabla; si se quiere quitar la tabla de la base de datos, hay que ejecutar la sentencia:

```
DROP TABLE <tabla>
```

El ejemplo que se muestra a continuación quita de la tabla *telefonos* el alumno que tiene la clave especificada:

```
DELETE FROM telefonos WHERE telefono='958324555'
```

Seleccionar datos de una tabla

Para seleccionar datos de una tabla, SQL proporciona la sentencia SELECT. Las cláusulas principales de esta sentencia se pueden resumir del modo siguiente:

```
SELECT [ALL | DISTINCT] <lista de selección>
    FROM <tablas>
    WHERE <condiciones de selección>
    [ORDER BY <columna 1> [ASC|DESC][, <columna 2> [ASC|DESC]]...]
```

Las cláusulas de una instrucción SELECT deben especificarse en el orden indicado.

El argumento *lista de selección* describe las columnas del conjunto de resultados que se desea obtener. Es una lista de expresiones separadas por comas. Cada expresión de lista de selección suele ser una referencia a una columna de la tabla de la que provienen los datos, aunque puede ser cualquier otra expresión. Al usar la expresión * en una lista de selección se especifica que se devolverán todas las columnas de la tabla de origen.

La cláusula DISTINCT elimina las repeticiones del conjunto de resultados obtenido por SELECT y ALL especifica que pueden aparecer filas duplicadas en el conjunto de resultados; es el valor predeterminado.

La cláusula FROM especifica una lista de las tablas de donde se recuperan los datos del conjunto de resultados.

La cláusula WHERE describe un filtro que define las condiciones que debe cumplir cada fila de las tablas de origen para satisfacer los requisitos de la instrucción SELECT. Sólo las filas que cumplen las condiciones contribuyen con datos al conjunto de resultados. Los datos de las filas que no cumplen las condiciones no se usan.

La cláusula ORDER BY define el orden de las filas del conjunto de resultados y especifica las columnas que intervienen en la clasificación. Las palabras clave ASC y DESC se utilizan para especificar si las filas se ordenan en una secuencia ascendente o descendente. Por omisión se supone ASC.

El ejemplo siguiente lista todas las filas de la tabla *telefonos*:

```
SELECT * FROM telefonos
```

Este otro ejemplo lista todas las filas de la tabla *telefonos* ordenadas ascendentemente por el campo *nombre*:

```
SELECT * FROM telefonos ORDER BY nombre
```

SQL permite utilizar los operadores <, <=, >, >=, <>, AND, OR, NOT, IS NULL, LIKE, BETWEEN, IN, ALL, ANY, etc. El ejemplo siguiente lista los registros de la tabla *telefonos* cuyo teléfono sea mayor que 958000000:

```
SELECT * FROM telefonos WHERE telefono>'958000000'
```

El siguiente ejemplo lista todos los registros de la tabla *telefonos* que empiecen por 91 (91* o 91%):

```
SELECT * FROM telefonos WHERE telefono LIKE '91*'
```

Crear una base de datos

Dependiendo del sistema administrador de bases de datos que tenga, *SQL Server*, *Access*, *Oracle*, *DB2*, *PostgreSQL*, *MySQL*, etc., la operación de crear una base de datos puede variar cuando se realiza desde el entorno de desarrollo aportado por ese sistema, pero no variaría si la creáramos desde la línea de órdenes utilizando el lenguaje SQL, operación que no es muy común (véase en el apéndice A, *SQL Server Express*).

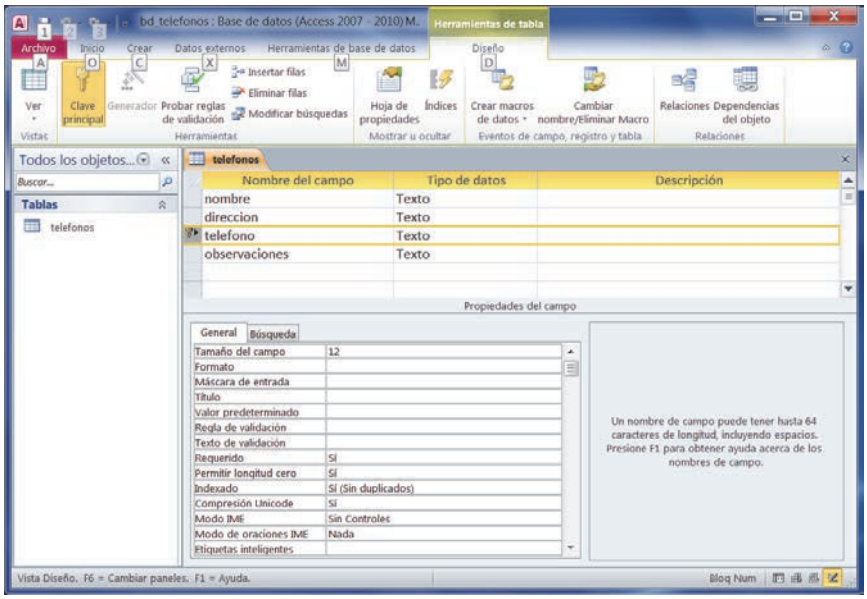
Base de datos Microsoft Access

Por ejemplo, vamos a crear una base de datos con *Microsoft Access*. Desde este entorno de desarrollo, esta operación resulta muy sencilla. Simplemente hay que:

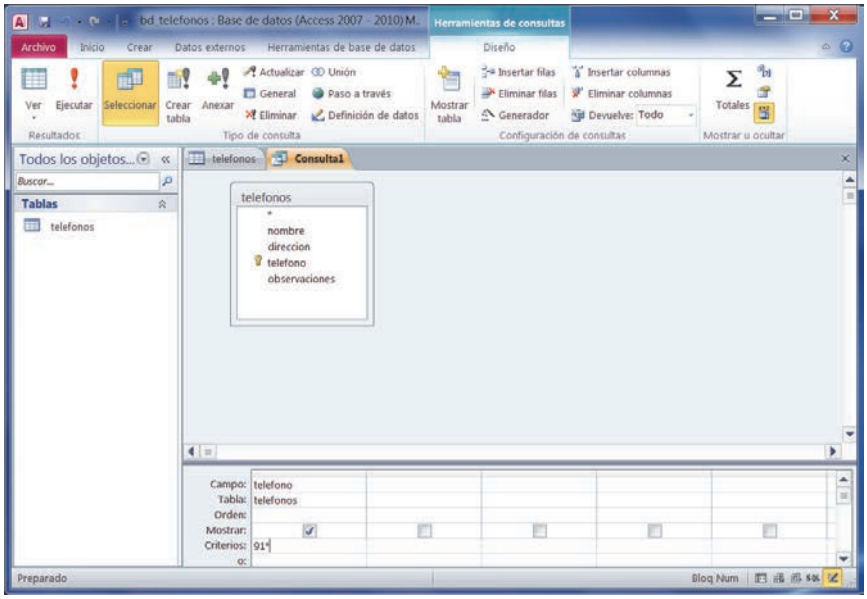
- Abrir *Microsoft Access*.
- Ejecutar la orden *Nuevo* del menú *Archivo*. Seleccionar *Base de datos en blanco*.
- Introducir el nombre de la base de datos. En nuestro caso *bd_telefonos.accdb*. Pulsar el botón *Crear*. Se abre la nueva base de datos, se crea una nueva tabla denominada *Tabla1* y se abre en la vista *Hoja de datos*. Cuando necesite crear una nueva tabla: *Crear* > *Tablas*.
- Cambie a la vista de diseño para editar los campos de la tabla: *Ver* > *Vista de diseño* y asigne un nombre a la tabla; en nuestro caso *telefonos*.
- Introducir el nombre, el tipo y las propiedades para cada uno de los campos de un registro. En nuestro caso:

Campo	Tipo	Descripción
nombre	Texto de 30 caracteres	Nombre y apellidos
direccion	Texto de 30 caracteres	Dirección de la persona
telefono	Texto de 12 caracteres	Número de teléfono
observaciones	Texto de 240 caracteres	Observaciones

- Requerir los tres primeros campos (propiedad *Requerido* = *sí*).
- Para el campo *telefono* establecer la propiedad *Indexado* al valor *sí (sin duplicados)*.
- Definir *telefono* como clave principal (clic con el botón secundario del ratón y seleccionar *Clave principal*).
- Guarde el diseño y cambie a la vista de datos, *Ver* > *Vista hoja de datos*, para introducir los datos.



Una vez creada la base de datos, puede ejecutar manualmente cualquier sentencia SQL sobre la misma. Para ello, haga clic en *Crear* > *Diseño de consulta*, agregue la tabla, seleccione el campo y, por ejemplo, fije los criterios. Finalmente, haga clic en *Ejecutar* (esquina superior izquierda). En la figura siguiente puede observarse la vista de diseño de consultas; si lo prefiere, puede cambiar a la vista SQL haciendo clic en el botón *SQL* situado en la esquina inferior derecha:



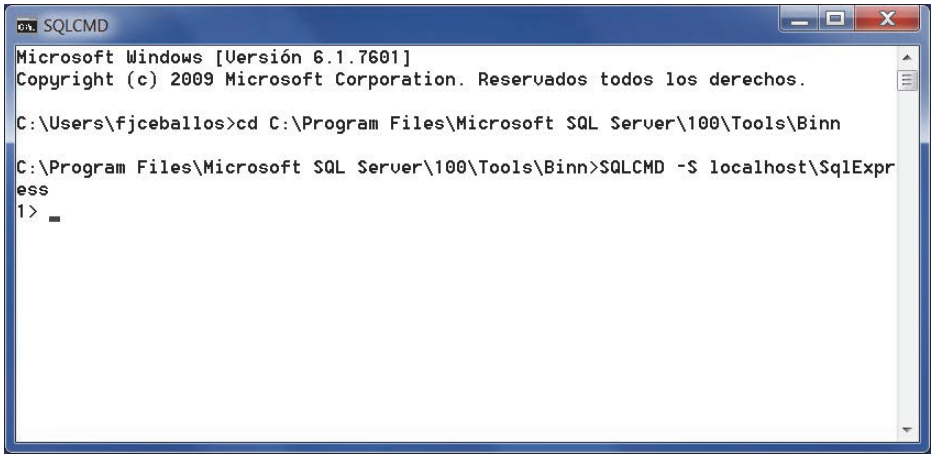
Base de datos Microsoft SQL Server

En este apartado, vamos a crear una base de datos con *Microsoft SQL Server*. Este proceso resulta muy sencillo si tiene instalado el administrador corporativo que incorpora el paquete de SQL Server (en su versión empresarial) o si sólo tiene instalado SQL Server Express, puede instalar cualquier otro administrador de características análogas, como *Microsoft SQL Server Management Studio Express*, o también, otra forma de crear bases de datos es a través del explorador de servidores del EDI de Visual Studio (véase el apéndice A). También, aunque sólo tenga instalado SQL Server Express, dispondrá de la utilidad SQLCMD ejecutable desde la línea de órdenes. Como ejemplo, utilizando SQLCMD, vamos a crear la misma base de datos que creamos en el apartado anterior con *Access* (en este caso la vamos a denominar *bd_telefonos*). Para ello, siga los pasos indicados a continuación:

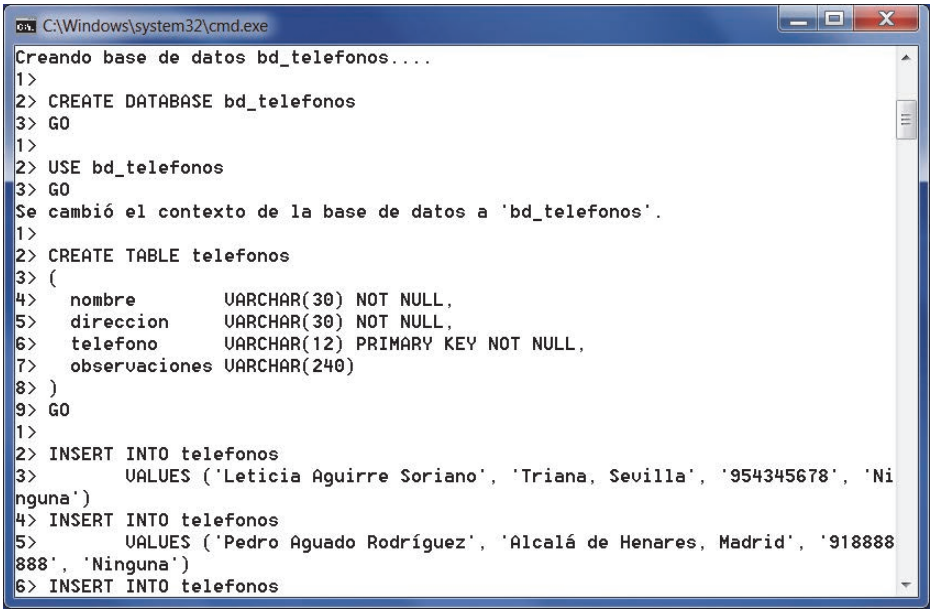
- Ejecute la orden *Inicio* > *Ejecutar* > *cmd* para abrir el intérprete de órdenes (consola del sistema).

- Localice en su instalación el fichero SQLCMD.EXE, cambie a ese directorio y ejecute la orden:

```
SQLCMD -S nombre-del-ordenador\SqlExpress
```



- Ejecute el guión indicado en la figura siguiente. En la carpeta *Cap06* localizada en el CD que acompaña al libro, tiene el fichero *generador_bd_telefonos.sql* que le proporcionará dicho guión; para mayor sencillez, cópielo y péguelo en la línea de órdenes.



La base de datos *bd_telefonos* está creada; tiene una tabla denominada *telefonos*. Si tiene instalada la utilidad *Microsoft SQL Server Management Studio Express* a la que nos hemos referido anteriormente, podrá utilizarla para operar sobre la base de datos, o bien, a falta de esta herramienta con interfaz gráfica, puede seguir utilizando SQLCMD. Finalmente, conocer que el fichero *bd_telefonos.mdf* está localizado en la carpeta *Data* de la instalación de Microsoft SQL Server, aunque este dato no es necesario.

ADO.NET

Muchas de las aplicaciones, distribuidas o no, trabajan sobre bases de datos. Por esta razón, Microsoft decidió crear una tecnología de acceso a datos potente y fácil de utilizar: ADO.NET.

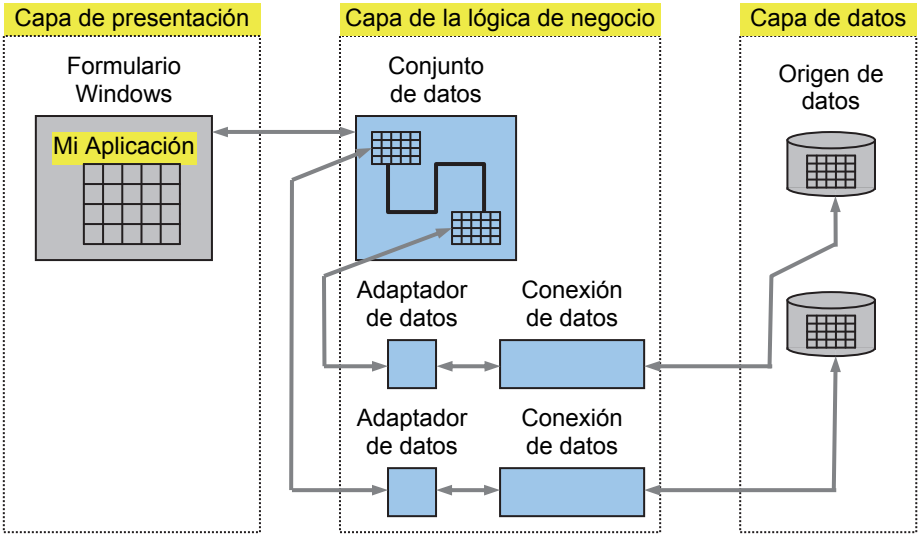
- ADO.NET *no depende de conexiones* continuamente activas, esto es, se diseñó en torno a una arquitectura donde las aplicaciones se conectan a la base de datos sólo durante el tiempo necesario para extraer o actualizar los datos. De esta forma, la base de datos no contiene conexiones que la mayor parte del tiempo permanecen inactivas, lo que se traduce en dar servicio a muchos más usuarios y facilita la escalabilidad.
- Las interacciones con la base de datos se realizan mediante *órdenes para acceso a los datos*, que son objetos que encapsulan las sentencias SQL o los procedimientos almacenados que definen la operación a realizar sobre el origen de datos.

- Los datos requeridos normalmente se almacenan en memoria caché en *conjuntos de datos*, lo que permite trabajar sin conexión sobre una copia temporal de los datos obtenidos. Los conjuntos de datos son independientes de los orígenes de datos. Cuando sea necesario, se puede restablecer la conexión con la base de datos y actualizarla desde el conjunto de datos.
- En ADO.NET, el formato de transferencia de datos es XML. La representación XML de los datos no utiliza información binaria (muchos servidores de seguridad bloquean la información binaria), sino que se basa en texto, lo que permite enviarla mediante cualquier protocolo, como por ejemplo HTTP.

Componentes de ADO.NET

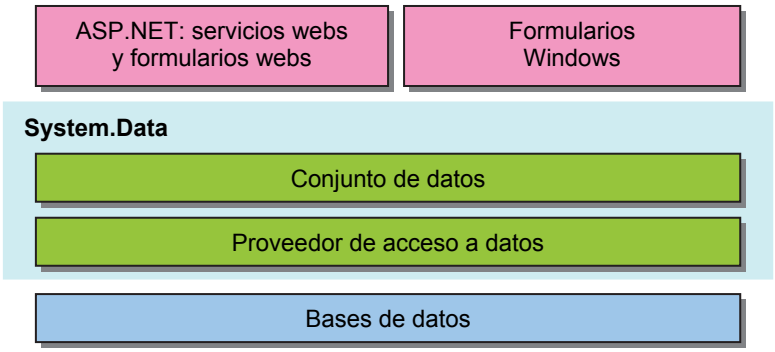
ADO.NET es un conjunto de clases, pertenecientes al espacio de nombres **System.Data**, para acceso a los datos de un origen de datos. Dicho de otra forma, ADO.NET proporciona un conjunto de componentes para crear aplicaciones distribuidas de uso compartido de datos. Dichos componentes están diseñados para separar el acceso a los datos de la manipulación de los mismos y son: **DataSet** y el *proveedor de datos* de .NET Framework, que es un conjunto de componentes entre los que se incluyen los objetos conexión (**Connection**), de órdenes (**Command**), lector de datos (**DataReader**) y adaptador de datos (**DataAdapter**), que describimos a continuación.

Como paso previo a esta descripción, para una mejor comprensión, la figura siguiente muestra cómo trabajan conjuntamente los objetos mencionados entre sí, para que una aplicación pueda interactuar con un origen de datos.



De forma resumida se puede decir que el trabajo de conexión con la base de datos o la ejecución de una sentencia SQL determinada para recuperar datos de la base de datos los realiza el *proveedor de acceso a datos*.

En cambio, recuperar esos datos para tratarlos, manipularlos o volcarlos a un determinado control o dispositivo es una acción ejecutada por una capa superior formada por un *conjunto de datos* agrupados en tablas.



A continuación, estudiaremos ambas capas con más detalle, analizando cada una de las partes que componen el modelo ADO.NET.

Conjunto de datos

Según se observa en las figuras anteriores, un formulario Windows (en nuestro caso una ventana WPF), para acceder a los datos de un origen de datos, lo que hace generalmente es utilizar un adaptador de datos para leer la información de la base de datos y almacenarla en un conjunto de datos. Posteriormente, cuando quiera escribir en el origen de datos, volverá a utilizar el adaptador que tomará los datos del conjunto de datos. Como alternativa, según veremos un poco más adelante, se puede interactuar directamente con la base de datos utilizando un objeto de órdenes para acceso a los datos que incluya una sentencia SQL o una referencia a un procedimiento almacenado.

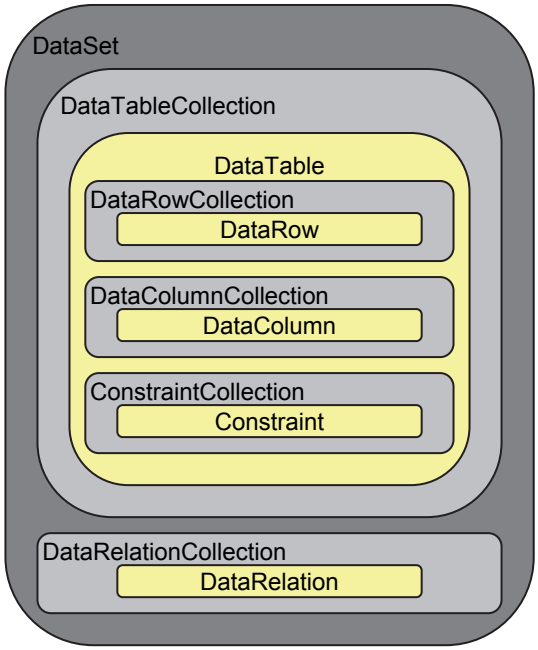
Un conjunto de datos incluye una o más tablas basadas en las tablas del origen de datos y también puede incluir información acerca de las relaciones entre estas tablas y las restricciones para los datos que puede contener cada tabla. Las partes fundamentales de un conjunto de datos, como veremos a continuación, se exponen al programador mediante propiedades y colecciones.

En ADO.NET, el componente central de la arquitectura sin conexión es la clase de objetos **DataSet** (conjunto de datos) perteneciente al espacio de nombres **System.Data**, que se puede utilizar con múltiples y distintos orígenes de datos.

La clase **DataSet** incluye la colección **DataTableCollection** de objetos **DataTable** (tablas de datos), a la que se obtiene acceso a través de la propiedad **Tables** del **DataSet**, y la colección **DataRelationCollection** de objetos **DataRelaDataRelation** (relaciones entre las tablas).

A su vez, la clase **DataTable** incluye la colección **DataRowCollection** de objetos **DataRow** (filas de tabla), la colección **DataColumnCollection** de objetos **DataColumn** (columnas de datos) y la colección **ConstraintCollection** de objetos **Constraint** (restricciones).

Según lo expuesto, un objeto **DataSet** con varias tablas (sólo representamos una) tendría la estructura siguiente:

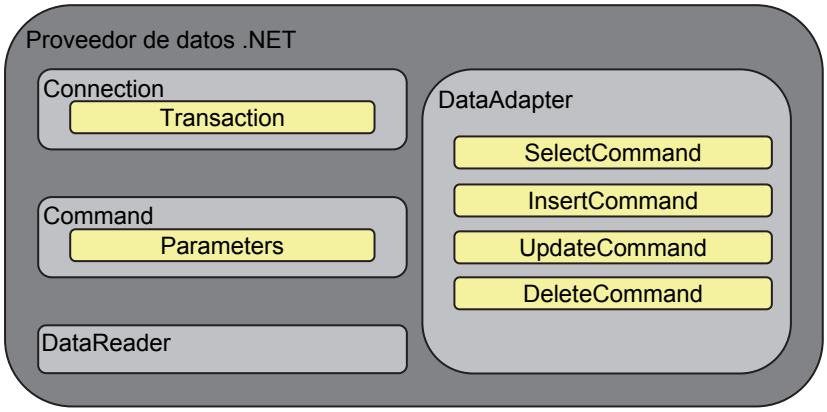


Así mismo, la clase **DataRow** incluye la propiedad **RowState**, que permite saber si la fila cambió y de qué modo, desde que la tabla de datos se cargó por primera vez. Algunos de los valores que esta propiedad puede tomar son: **Added**, **Deleted**, **Modified** y **Unchanged**.

Proveedor de datos

En .NET Framework, un proveedor de datos sirve como puente entre una aplicación y un origen de datos. Se utiliza tanto para recuperar datos de un origen como para actualizarlos. Los componentes principales de un proveedor de datos .NET son los objetos siguientes:

- *Conexión* con el origen de datos (objeto **Connection**). Establece una conexión a un origen de datos determinado.
- *Orden* para acceso a los datos (objeto **Command**). Ejecuta una orden SQL o un procedimiento almacenado en un origen de datos.
- *Lector de datos* (objeto **DataReader**). Proporciona una forma rápida de acceder a los datos recuperados después de una consulta a la base de datos. El acceso permitido es sólo para leer y hacia delante.
- *Adaptador de datos* (objeto **DataAdapter**). Llena un **DataSet** y realiza las actualizaciones necesarias en el origen de datos.



.NET incluye los siguientes proveedores de datos: ODBC, OLE DB, Oracle y SQL Server, que podemos encontrar en los espacios de nombres System.Data.Odbc, System.Data.OleDb, System.Data.OracleClient y System.Data.SqlClient, respectivamente. Cada proveedor de datos tiene una implementación concreta de las clases **Connection**, **Command**, **DataReader** y **DataAdapter** que han sido optimizadas para un determinado sistema de gestión de base de datos (SGBD). Por ejemplo, si usted necesita crear una conexión con una base de datos de SQL Server, utilizará la clase de conexión **SqlConnection**.

El proveedor **Odbc** permite conectar una aplicación a distintos orígenes de datos a través de ODBC. El proveedor **OleDb** permite conectar una aplicación a distintos orígenes de datos a través de OLE DB. El proveedor **OracleClient** es un

proveedor de acceso a datos nativo especialmente diseñado para bases de datos Oracle. Por último, el proveedor **SqlClient** es un proveedor de acceso a datos nativo, que nos permite conectar una aplicación a orígenes de datos Microsoft SQL Server 7 o posterior.

Como vemos, Microsoft proporciona los proveedores de acceso a datos más corrientes, pero éstos no son todos; no obstante, con ODBC y OLE DB, podemos acceder a la inmensa totalidad de ellos.

Sin embargo, hay muchos motores de bases de datos de igual importancia como MySql, AS/400, etc. En estos casos, si queremos utilizar un proveedor de acceso a datos nativo, deberemos acudir al fabricante para que nos proporcione el conjunto de clases que definen ese proveedor en particular, o bien crear nuestro propio proveedor de datos a partir de las clases del espacio de nombres **System.Data.Common** que proporciona clases para la creación de objetos **DbProviderFactory** que permiten trabajar con orígenes de datos específicos.

Es aconsejable, siempre que pueda, utilizar un proveedor de acceso a datos nativo para acceder a bases de datos. Esto permitirá aumentar considerablemente el rendimiento a la hora de establecer la conexión con un determinado origen de datos.

A primera vista, podría parecer que ADO.NET ofrece un modelo fragmentado, ya que no incluye un conjunto genérico de objetos que pueda trabajar con múltiples tipos de bases de datos. Como resultado, si se cambia de un SGBD a otro tendrá que modificar su código de acceso de datos para utilizar un conjunto diferente de clases. Pero a pesar de que los diferentes proveedores utilicen diferentes clases, todos ellos se han estandarizado en la misma forma. Más específicamente, cada proveedor se basa en el mismo conjunto de interfaces y clases base. Por ejemplo, cada objeto **Connection** implementa la interfaz **IDbConnection**, que define los métodos básicos, tales como **Open** y **Close**. Esta estandarización garantiza que todas las clases de conexión funcionarán de la misma manera y expondrán el mismo conjunto de propiedades y métodos.

Objeto conexión

Para establecer una conexión a un origen de datos, ADO.NET proporciona el objeto **Connection**. Por ejemplo, para establecer una conexión a Microsoft SQL Server utilizaremos el objeto **SqlConnection**, para conectarse a un origen de datos OLE DB utilizaremos el objeto **OleDbConnection**, para conectarse a un origen de datos ODBC utilizaremos el objeto **OdbcConnection** y para conectarse a un origen de datos de Oracle, utilizaremos el objeto **OracleConnection**.

La función de un objeto conexión es presentar atributos y métodos para permitir establecer y modificar las propiedades de la conexión (por ejemplo, el identificador de usuario y la contraseña, entre otras).

En el ejemplo siguiente se muestra cómo crear una conexión:

```
Dim conexion As New OleDbConnection(strConexion)
```

El argumento *strConexion* hace referencia a la cadena de conexión. La cadena de conexión está compuesta por una serie de elementos *nombre/valor* separados por punto y coma. El orden de estos elementos no es importante. En conjunto, especifican la información básica necesaria para crear una conexión. Aunque las cadenas de conexión varían según el proveedor de SGBD que se esté utilizando, algunos elementos, como los citados a continuación, son casi siempre requeridos:

- El servidor donde se encuentra la base de datos.
- La base de datos que desea utilizar.
- La autenticación. Oracle y SQL Server darán la opción de suministrar credenciales de identificación como usuario y contraseña.

Por ejemplo, la cadena de conexión siguiente se utiliza para conectarse a la base de datos Sql Server denominada *bd_telefonos* en el equipo actual utilizando la seguridad integrada (esto es, utiliza el usuario actual de Windows para acceder a la base de datos):

```
Dim strConexion As String = "Data Source=.\sqlexpress;" &
  "Initial Catalog=bd_telefonos; Integrated Security=True"
```

Si la seguridad integrada no es compatible con la conexión, deberá indicar un usuario válido junto con una contraseña. Por ejemplo, para una base de datos PostgreSql la cadena de conexión podría ser así:

```
Dim strConexion As String = "SERVER=localhost;" &
  "Database=bd_telefonos; User name=*****; Password=*****"
```

Si está utilizando el proveedor OLE DB, la cadena de conexión seguirá siendo similar, pero ahora hay que proporcionar un nombre de proveedor que identifique el controlador OLE DB que se va a utilizar. Por ejemplo, puede utilizar la siguiente cadena de conexión para conectarse a una base de datos Microsoft Access:

```
Dim strConexion As String = "Provider=Microsoft.ACE.OLEDB.12.0;" &
  "Data Source=C:./../../bd_telefonos.accdb;"
```

Cuando utilizamos OLE DB en un sistema X64 tenemos que saber que hay controladores OLE DB para 32 y para 64 bits. Por ejemplo, el proveedor *Micro-*

soft.Jet.OLEDB.4.0 es para 32 bits y *Microsoft.ACE.OLEDB.12.0*, dependiendo de la versión de *Office*, puede ser para 32 o para 64 bits. Según esto, habrá que seleccionar *X86* o *X64* como *CPU de destino* en la pestaña *Compilar*, *Opciones de compilación avanzadas*, de las propiedades del proyecto.

Objeto orden

Después de establecer una conexión con un origen de datos, puede utilizar un objeto **Command** para ejecutar sentencias SQL y devolver resultados desde ese origen de datos. Para crear un objeto de éstos, invoque a su constructor. Si el objeto ya está creado, la instrucción SQL encapsulada por él podrá ser consultada o modificada a través de su propiedad **CommandText**.

En el caso de orígenes de datos compatibles con OLE DB, utilice **OleDbCommand**, para orígenes de datos compatibles con ODBC, use **OdbcCommand**, para Microsoft SQL Server, utilice **SqlCommand** y para Oracle, **OracleCommand**. En el ejemplo siguiente se muestra cómo crear una orden SQL para acceder a un origen de datos compatible con OLE DB:

```
Dim ordenSQL As New OleDbCommand( _
  "SELECT nombre, telefono FROM telefonos",
  conexion)
```

Para ejecutar esta orden contra la base de datos podemos utilizar, básicamente, alguno de estos métodos: **ExecuteNonQuery** y **ExecuteReader**.

ExecuteNonQuery se utiliza para ejecutar operaciones de manipulación de datos como UPDATE, INSERT o DELETE (SELECT no). El valor devuelto corresponde al número de filas afectadas por la orden ejecutada. Por ejemplo:

```
ordenSQL.ExecuteNonQuery()
```

También podemos utilizar este método para ejecutar operaciones de definición, por ejemplo CREATE, ALTER o DROP.

ExecuteReader se utiliza para ejecutar una consulta SELECT y devuelve un objeto **DataReader**. Por ejemplo:

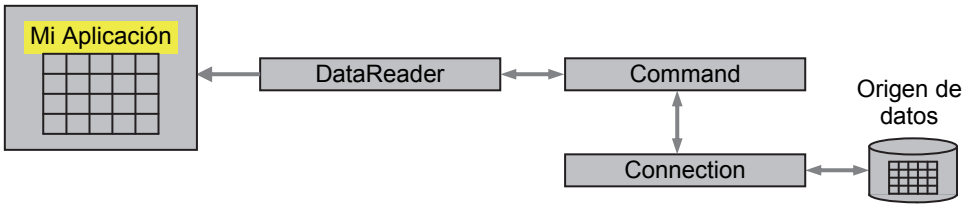
```
Dim lector As OleDbDataReader = ordenSQL.ExecuteReader()
```

Objeto lector de datos

Cuando una aplicación sólo necesite leer datos (no actualizarlos), no será necesario almacenarlos en un conjunto de datos, basta utilizar un objeto lector de datos en su lugar. Un objeto lector de datos obtiene los datos del origen de datos y los

pasa directamente a la aplicación (un adaptador de datos utiliza un objeto lector de datos para llenar un conjunto de datos).

El objeto lector de datos proporcionado por .NET para SQL Server es **SqlDataReader** y para orígenes OLE DB es **OleDbDataReader**. La figura siguiente muestra cómo se utilizan estos objetos:



El ejemplo siguiente muestra cómo utilizar un lector de datos para obtener el resultado proporcionado por la orden SQL SELECT anterior:

```
conexion.Open()
Dim lector As OleDbDataReader = ordenSQL.ExecuteReader()

While lector.Read()
  Console.WriteLine(lector.GetString(0) & ", " &
                    lector.GetString(1))
End While
lector.Close()
conexion.Close()
```

El método **Read** desplaza el cursor del **DataReader**, de sólo lectura y avance hacia delante, hasta el siguiente registro. La posición inicial es antes del primer registro, por lo tanto, se debe llamar a **Read** para iniciar el acceso a los datos. Una vez finalizada la lectura se debe llamar al método **Close** para liberar el **DataReader** de la conexión, objeto **Connection**.

Para recuperar las columnas de la fila actual del conjunto de datos proporcionado por el **DataReader**, utilizaremos la funcionalidad proporcionada por este objeto; por ejemplo, **GetValue** obtiene el valor de la columna especificada en su formato nativo, **GetValues** rellena la matriz de objetos pasada como argumento con los valores de la fila actual, **GetString** obtiene el valor de la columna especificada como una cadena, **GetInt32** obtiene el valor de la columna especificada como un entero de 32 bits con signo, etc.

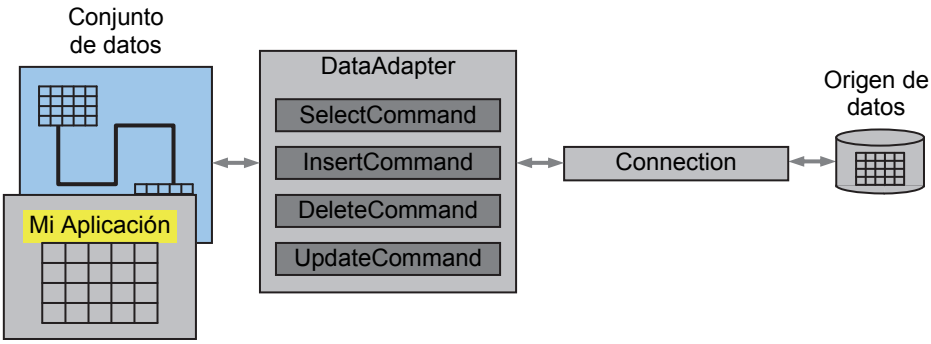
Adaptador de datos

Un adaptador es un conjunto de objetos utilizado para intercambiar datos entre un origen de datos y un conjunto de datos (objeto **DataSet**). Esto significa que una

aplicación leerá datos de una base de datos para un conjunto de datos y, a continuación, manipulará dichos datos. También, en algunas ocasiones, volverá a escribir en la base de datos los datos modificados del conjunto de datos.

En el caso de orígenes de datos compatibles con OLE DB, utilice **OleDbDataAdapter** junto con sus objetos **OleDbCommand** y **OleDbConnection** asociados. En el caso de otros orígenes de datos compatibles con ODBC, utilice **OdbcDataAdapter** junto con sus objetos **OdbcCommand** y **OdbcConnection** asociados. Si se conecta a una base de datos de Microsoft SQL Server, podrá mejorar el rendimiento general utilizando **SqlDataAdapter** junto con sus objetos asociados **SqlCommand** y **SqlConnection**. En el caso de bases de datos de Oracle, utilice **OracleDataAdapter** junto con sus objetos **OracleCommand** y **OracleConnection** asociados.

Generalmente, cada adaptador de datos, como muestra la figura siguiente, intercambia datos entre una sola tabla de un origen de datos y un solo objeto **DataTable** (tabla de datos) del conjunto de datos. Esto quiere decir que lo normal es utilizar tantos adaptadores como tablas tenga el conjunto de datos. De esta forma, cada tabla del conjunto de datos tendrá su correspondiente tabla en el origen de datos. En la siguiente figura se puede observar la utilización de un adaptador de datos para llenar un conjunto de datos (objeto **DataSet**):



Según se observa en la figura anterior, un adaptador contiene también las propiedades **SelectCommand**, **InsertCommand**, **DeleteCommand**, **UpdateCommand** y **TableMappings** para facilitar la lectura y actualización de los datos en un origen de datos.

SelectCommand hace referencia a una orden que recupera filas del origen de datos, entendiendo por orden un objeto **Command** que almacena una instrucción SQL o un nombre de procedimiento almacenado, **InsertCommand** hace referencia a una orden para insertar filas en el origen de datos, **UpdateCommand** hace referencia a una orden para modificar filas en el origen de datos, y **DeleteCommand** hace referencia a una orden para eliminar filas del origen de datos.

Modos de conexión

La acción más pesada cuando realizamos un acceso a una base de datos se encuentra en la conexión con la base de datos. Esa tarea tan simple es la que más recursos del sistema consume. Por ello, es bueno tener presente las siguientes consideraciones:

- La conexión debe realizarse, siempre que se pueda, con los proveedores de acceso a datos nativos, simplemente porque son más rápidos que los proveedores del tipo OLE DB y ODBC.

- La conexión debe abrirse lo más tarde posible. Es recomendable definir todas las variables que podamos antes de realizar la conexión.

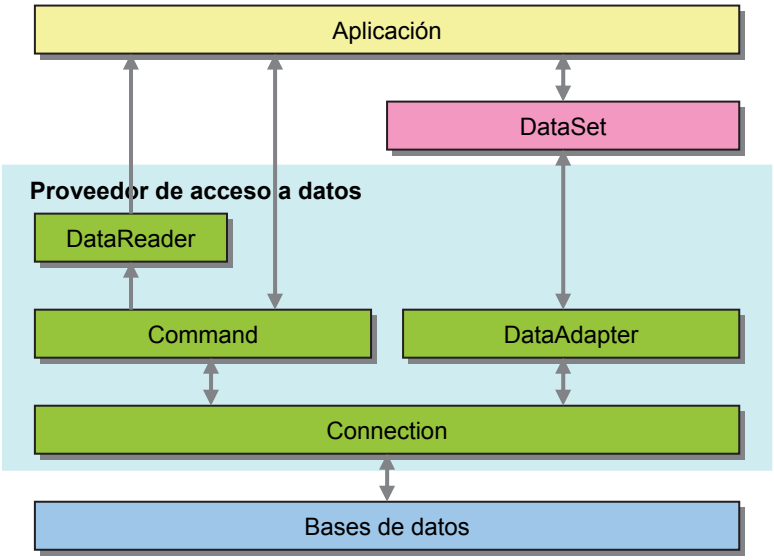
- La conexión debe cerrarse lo antes posible, siempre y cuando no tengamos la necesidad de utilizarla posteriormente.

Evidentemente se nos presentarán casos en los que será difícil asumir qué acción tomar. Esto es, parece que deberíamos mantener una conexión abierta sólo en el instante de acceder a la base de datos. Pero, ¿y si estamos trabajando continuamente con esa base de datos? ¿La penalización sería menor si la mantuviéramos abierta? No está del todo claro, ya que no existe en sí una regla clara que especifique qué acción tomar en cada caso. Por eso, dependiendo de lo que vayamos a realizar, nos inclinaremos por una acción u otra.

Lo que sí está claro es que el modelo de datos de ADO.NET que hemos visto quedaría resumido, en cuanto a la conectividad se refiere, de la forma indicada en la figura siguiente, en la que se pueden observar dos formas de trabajar con los datos de una base de datos: conectado a la base de datos y desconectado.

El objeto **DataSet** nos ofrece la posibilidad de almacenar datos de una determinada base de datos (tablas de una base de datos). Esto hace posible que una aplicación pueda trabajar con los datos procedentes de una base de datos estando desconectada de dicha base. Esta forma de trabajo no significa que dentro del **DataSet** podamos abrir una tabla con una cantidad de registros enorme, y trabajemos sobre ella creyendo que esto nos beneficiará. Todo lo contrario. Esta práctica penalizaría seriamente el rendimiento de nuestra aplicación.

Obsérvese que el adaptador de datos actúa como puente entre el **DataSet** y el origen de datos para la recuperación y el almacenamiento de datos. Para poder actuar como un puente, **DataAdapter** proporciona el método **Fill**, que modifica los datos de **DataSet** de forma que coincidan con los del origen de datos, y el método **Update**, que modifica los datos del origen de datos para hacerlos coincidir con los de **DataSet**.



En otras ocasiones será necesario trabajar estando la aplicación conectada a la base de datos. Esto es precisamente lo que nos ofrece el objeto **DataReader**: la posibilidad de trabajar con bases de datos conectadas. No obstante, este objeto tiene algunas particularidades que conviene conocer y que especificamos a continuación:

- El objeto **DataReader** recupera un conjunto de valores llenando un pequeño búfer de datos.

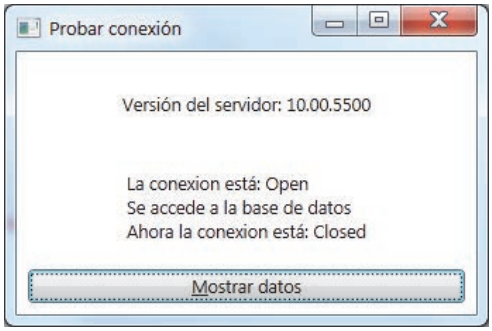
- Si los registros que hay en el búfer se acaban, el objeto **DataReader** regresará a la base de datos para recuperar más registros. Por lo tanto, si el servicio de SQL Server estuviera detenido en ese momento o, en su caso, el origen de datos estuviera detenido, la aplicación generaría un error a la hora de leer el siguiente registro.

Resumiendo, **DataReader** es un objeto conectado, pero trabaja en segundo plano con un conjunto de datos, por lo que a veces nos podría resultar chocante su comportamiento.

Probando una conexión

Una vez generada la base de datos (al principio de este capítulo creamos una base de datos Sql Server denominada *bd_telefonos*) y definida la cadena de conexión, simplemente tenemos que crear el objeto conexión y utilizar sus métodos **Open** y **Close** para abrir y cerrar dicha conexión. Podemos ver esto con un ejemplo que

muestre una ventana con un botón que nos permita abrir y cerrar la conexión, reflejando el estado de la misma en la propia ventana, además de mostrar la versión de Sql Server. Para ello, haciendo uso de los conocimientos adquiridos en los capítulos anteriores, cree un proyecto WPF, por ejemplo *ProbarConexion*, y diseñe esta ventana:



Una vez diseñada la ventana, añadimos el código que se debe ejecutar para abrir y cerrar la conexión. Esto es, en primer lugar, cuando la ventana se muestre, crearemos el objeto conexión. Añada por lo tanto el controlador del evento **Loaded** de la ventana y edítelo como se indica a continuación:

```
Private Sub Window_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Crear la conexión
  Dim strConexion As String = "Data Source=.\sqlexpress;" &
    "Initial Catalog=bd_telefonos; Integrated Security=True"
  con = New SqlConnection(strConexion)
End Sub
```

La referencia *con* al objeto de conexión la declaramos como un atributo de la clase *MainWindow* para que esté accesible para toda la ventana:

```
Class MainWindow
  Private con As SqlConnection = Nothing
  ' ...
End Class
```

Cuando el usuario haga clic en el botón "Mostrar datos", supuestamente para acceder a la base de datos y obtener un conjunto de datos, se abrirá la conexión, se obtendrán los datos y se cerrará la conexión. Añada por lo tanto el controlador del evento **Click** del botón y edítelo como se muestra a continuación:

```
Private Sub btMostrarDatos_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  Try
```

```
    ' Probar a abrir la conexión
    con.Open()
    tbVersion.Text = "Versión del servidor: " &
      Convert.ToString(con.ServerVersion)
    tbEstadoConexion.Text = "La conexion está: "
    tbEstadoConexion.Text += con.State.ToString()
    tbEstadoConexion.Text += vbLf & "Se accede a la base de datos"
  Catch ex As Exception
    ' Manipular la excepción
    tbEstadoConexion.Text = "Error al acceder a la base de datos. " +
                            ex.Message
  Finally
    ' Asegurarse de que la conexión queda cerrada.
    ' Aunque la conexión estuviera cerrada,
    ' llamar a Close() no produce un error.
    con.Close()
    tbEstadoConexion.Text += vbLf & "Ahora la conexion está: " &
                             con.State.ToString()
  End Try
End Sub
```

Obsérvese que pase lo que pase, el bloque **Finally** nos asegura que la conexión con la base de datos quedará cerrada.

La versión del SGBD utilizado nos la proporciona la propiedad **ServerVersion** del objeto conexión y el estado de la conexión, la propiedad **State**.

Servicio de conexiones

Abrir una conexión consume una pequeña cantidad de tiempo. Pero piense ahora, por ejemplo, en una aplicación web y en las peticiones que le llegarán a la base de datos desde múltiples usuarios de la red. En un caso como éste, las conexiones se abrirán y cerrarán sin parar, una por cada solicitud que se procesa. Esta sobrecarga necesaria para establecer una conexión puede llegar a ser significativa y, evidentemente, limita la escalabilidad del sistema.

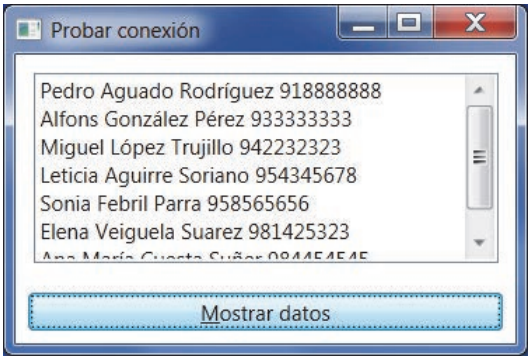
Una solución es utilizar un *servicio de conexiones* o *pool de conexiones*. Este servicio mantiene permanentemente un número de conexiones abiertas con la base de datos las cuales son compartidas por las peticiones que llegan a la misma. Esto evita la necesidad de crear y destruir las conexiones todo el tiempo. El servicio de conexiones en ADO.NET es completamente transparente para el programador. Esto es, cuando un cliente solicita al servidor de bases de datos una conexión llamando a **Open**, se sirve directamente desde el servicio de conexiones, en lugar de crear una nueva conexión y cuando un cliente libera una conexión llamando a **Close** o **Dispose**, ésta no se cierra, sino que queda disponible de nuevo para una nueva solicitud a través del servicio de conexiones.

ADO.NET no incluye un servicio de conexiones, pero la mayoría de los proveedores ADO.NET, como ocurre con Sql Server y Oracle, sí implementan alguna forma del servicio de conexiones.

Para casos especiales, parámetros tales como **Max Pool Size** o **Min Pool Size**, entre otros, pueden ser especificados en la cadena de conexión.

ACCESO CONECTADO A BASE DE DATOS

El siguiente ejemplo establece una conexión con el origen de datos *bd_telefonos*, base de datos *Sql Server*, y utilizando un **DataReader** muestra en una lista los datos *nombre* y *telefono* de cada uno de los registros de la tabla *telefonos*.



Para implementar el ejemplo anterior, cree un proyecto WPF, por ejemplo *DataReader*, y diseñe una ventana que contenga una lista, objeto **ListBox**, y un botón, objeto **Button**. Lo que queremos es mostrar en la lista las filas *nombre-telefono* obtenidas tras una consulta a la base de datos *bd_telefonos* que se realizará en el instante en el que el usuario haga clic en el botón. Una vez diseñada la ventana añadimos el código que se debe ejecutar para mostrar la lista.

En primer lugar, cuando la ventana se visualice, crearemos el objeto conexión. Añada por lo tanto el controlador del evento **Loaded** de la ventana y edítelo como se indica a continuación:

```
Private Sub Window_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Crear la conexión
  Dim strConexion As String = "Data Source=.\sqlexpress;" &
    "Initial Catalog=bd_telefonos; Integrated Security=True"
  ConexionConBD = New SqlConnection(strConexion)
End Sub
```

La referencia *ConexionConBD* al objeto **SqlConnection** la declaramos como un atributo de la clase *MainWindow* para que esté accesible para toda la ventana:

```
Class MainWindow
  Private ConexionConBD As SqlConnection = Nothing
  Private OrdenSql As SqlCommand
  Private Lector As SqlDataReader
  ' ...
End Class
```

También hemos declarado los atributos *OrdenSql*, de tipo **SqlCommand**, para construir la orden SQL que tenemos que ejecutar para obtener las filas *nombre-telefono* y *Lector*, de tipo **SqlDataReader**, para ejecutar la orden anterior contra la base de datos. Estas operaciones serán ejecutadas desde el controlador del evento **Click** del botón "Mostrar datos" una vez abierta la conexión con la base de datos, para después añadir a la lista las filas del conjunto de datos obtenido. Según esto, añada este controlador y edítelo como se indica a continuación:

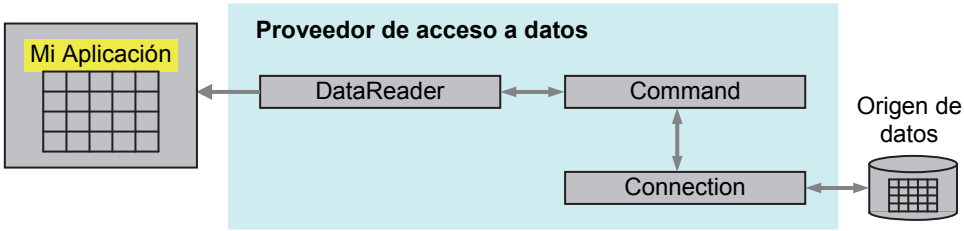
```
Private Sub btMostrarDatos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Using ConexionConBD
    ' Crear una consulta
    Dim Consulta As String = "SELECT nombre, telefono FROM telefonos"
    OrdenSql = New SqlCommand(Consulta, ConexionConBD)

    ' Abrir la conexión con la base de datos
    ConexionConBD.Open()
    ' ExecuteReader hace la consulta y devuelve un SqlDataReader
    Lector = OrdenSql.ExecuteReader()
    ' Llamar siempre a Read antes de acceder a los datos
    While Lector.Read() ' siguiente registro
      lsTfnos.Items.Add(Lector("nombre") + " " +
                        Lector("telefono"))
    End While
    ' Llamar siempre a Close una vez finalizada la lectura
    Lector.Close()
  End Using
  btMostrarDatos.IsEnabled = False
End Sub
```

En este caso, para cerrar la base de datos utilizamos, como alternativa a lo ya expuesto anteriormente, una sentencia **Using**. La sentencia **Using** define un ámbito fuera del cual se invoca automáticamente al método **Dispose** del objeto que interviene en la misma; en nuestro caso, del objeto *ConexionConBD*. Este objeto tiene que implementar la interfaz **IDisposable**, cosa que sucede con **SqlConnection**, que es la que obliga a implementar **Dispose**, método que, en este caso, ha sido diseñado para cerrar la conexión encapsulada por **SqlConnection**. Esto

ocurrirá cuando se salga del ámbito definido por **Using**, bien porque se alcanza el final de dicha sentencia o bien porque se lanza una excepción. Esto es, ocurra lo que ocurra, la conexión quedará siempre cerrada.

El código anterior, desde un punto de vista gráfico, se corresponde con la arquitectura siguiente:



A continuación se presenta otra versión de este mismo ejemplo, pero utilizando Microsoft Access. La base de datos se denomina también *bd_telefonos* y la forma de crearla fue expuesta anteriormente en este mismo capítulo.

```
Imports System.Data.OleDb

Class MainWindow
  Private ConexionConBD As OleDbConnection = Nothing
  Private OrdenSql As OleDbCommand
  Private Lector As OleDbDataReader

  Private Sub Window_Loaded(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ' Crear la conexión
    Dim strConexion As String =
      "Provider=Microsoft.ACE.OLEDB.12.0;" &
      "Data Source=C:./../../bd_telefonos.accdb;"
    ConexionConBD = New OleDbConnection(strConexion)
  End Sub

  Private Sub btMostrarDatos_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    Using ConexionConBD
      ' Crear una consulta
      Dim Consulta As String = "SELECT nombre, telefono FROM telefonos"
      OrdenSql = New OleDbCommand(Consulta, ConexionConBD)

      ' Abrir la conexión con la base de datos
      ConexionConBD.Open()
      ' ExecuteReader hace la consulta y devuelve un SqlDataReader
      Lector = OrdenSql.ExecuteReader()
      ' Llamar siempre a Read antes de acceder a los datos
      While Lector.Read() ' siguiente registro
        lsTfnos.Items.Add(Lector("nombre") + " " +
```

```
                        Lector("telefono"))
        End While
        ' Llamar siempre a Close una vez finalizada la lectura
        Lector.Close()
    End Using
    btMostrarDatos.IsEnabled = False
End Sub
```

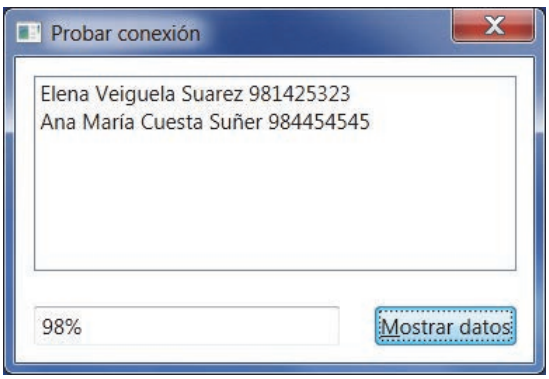
Cuando utilizamos OLE DB en un sistema X64 con el proveedor *Microsoft.ACE.OLEDB.12.0* de 64 bits, habrá que seleccionar *X64* como *CPU de destino* en la pestaña *Compilar*, *Opciones de compilación avanzadas*, de las propiedades del proyecto.

ATAQUES DE INYECCIÓN DE CÓDIGO SQL

Hasta ahora, todos los ejemplos que hemos visto han utilizado cadenas de caracteres para crear órdenes SQL. Eso hace que los ejemplos sean simples, directos y relativamente seguros, pero no son realistas y no demuestran uno de los riesgos más graves para las aplicaciones (en especial, para las aplicaciones webs) que interactúan con una base de datos: ataques de inyección de código SQL.

La inyección de SQL es el proceso de transmisión de código SQL a una aplicación, sin que esto haya sido previsto por el desarrollador de la misma, lo que se traduce en un mal diseño de la aplicación, y que afecta sólo a las aplicaciones que utilizan cadenas de caracteres para crear órdenes SQL a partir de los valores suministrados por el usuario. Veamos un ejemplo a partir de la aplicación anterior.

Consideremos el ejemplo que se muestra en la figura siguiente. En este ejemplo, el usuario introduce un prefijo de teléfono con la intención de mostrar en la lista todas las filas con ese prefijo. Vamos a diseñar esta nueva versión de la aplicación modificando la anterior. Para ello, añada a la ventana de la versión anterior un objeto **TextBox** que denominaremos *ctSql*.



Ahora, en esta nueva versión de la aplicación, la orden SQL tiene que ser creada dinámicamente añadiendo a la misma la información de la caja de texto *ctSql* en el lugar correspondiente:

```
Dim Consulta As String = "SELECT nombre, telefono FROM telefonos " +
                         "WHERE telefono LIKE '"+ctSql.Text+"'"
```

El resto del código no cambia mucho. Los pocos cambios que vamos a hacer son debidos a que ahora la operación "Mostrar datos" podrá repetirse tantas veces como queramos, para lo cual deberemos tener en cuenta que el método **Dispose** que cierra la conexión cuando finaliza la sentencia **Using**, también vacía la propiedad **ConnectionString** del objeto *ConexionConBD*, lo que nos obliga a iniciarlo cada vez que se ejecuta el controlador del evento **Click** del botón.

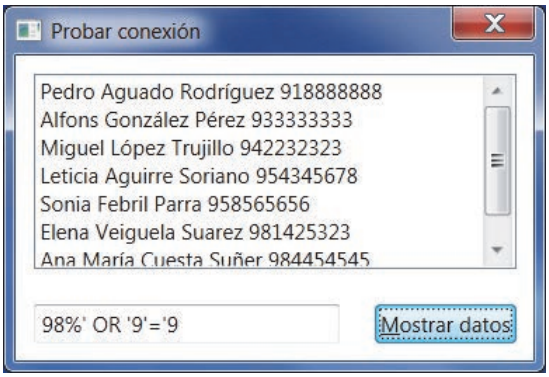
```
Class MainWindow
  Private ConexionConBD As SqlConnection = Nothing
  Private OrdenSql As SqlCommand
  Private Lector As SqlDataReader
  Private strConexion As String = Nothing

  Private Sub Window_Loaded(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    ' Crear la conexión
    strConexion = "Data Source=.\sqlexpress;" &
      "Initial Catalog=bd_telefonos; Integrated Security=True"
  End Sub

  Private Sub btMostrarDatos_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    Using ConexionConBD = New SqlConnection(strConexion)
      ' Crear una consulta
      Dim Consulta As String =
        "SELECT nombre, telefono FROM telefonos " &
        "WHERE telefono LIKE '" & ctSql.Text & "'"
      OrdenSql = New SqlCommand(Consulta, ConexionConBD)
      ' Abrir la conexión con la base de datos
      ConexionConBD.Open()
      ' ExecuteReader hace la consulta y devuelve un SqlDataReader
      Lector = OrdenSql.ExecuteReader()
      ' Limpiar la lista
      lsTfnos.Items.Clear()
      ' Llamar siempre a Read antes de acceder a los datos
      While Lector.Read() ' siguiente registro
        lsTfnos.Items.Add(Lector("nombre") + " " +
                          Lector("telefono"))
      End While
      ' Llamar siempre a Close una vez finalizada la lectura
      Lector.Close()
    End Using
```

```
  End Sub
End Class
```

El problema es que ahora el usuario puede inyectar código SQL en la caja de texto *ctSql* según muestra la figura siguiente (piense por un momento que, en lugar de un prefijo, se le solicitara una contraseña):



En este ejemplo, el usuario hace una entrada, *98%' OR '9'='9*, con la intención de alterar la sentencia SQL y lo que consigue son todas las filas de la tabla *telefonos* de la base de datos (muchas veces, el primer objetivo de un ataque es un mensaje de error y si el error no se maneja correctamente y la información de bajo nivel se expone al atacante, esa información puede ser utilizada para lanzar un ataque más sofisticado). Esa entrada ha dado lugar a que se cree la siguiente sentencia SELECT:

```
SELECT nombre, telefono FROM telefonos
                        WHERE telefono LIKE '98%' OR '9'='9'
```

La condición *9=9*, evidentemente, se cumple siempre, razón por la que toda la información es expuesta al atacante. Ahora podríamos lanzar un nuevo ataque malicioso introduciendo la siguiente cadena de texto:

```
98%'; DELETE FROM telefonos--
```

Los dos guiones finales comentan el resto del código de la sentencia SQL, en este caso el ' (en MySQL se utiliza la # y en Oracle, el ;). El resultado es que se muestran las filas que empiezan por 98 pero después se borran todas las filas de la tabla *telefonos*.

¿Cómo podemos defendernos contra estos ataques? Hay varias formas de hacerlo. Las más sencillas pueden ser:

- Sustituir cada ’ de la entrada del usuario por dos ’ para impedir que un usuario malintencionado cierre una cadena antes de tiempo.

```
ctSql.Text.Replace("’", "’’")
```

- Utilizar la propiedad **TextBox.MaxLength** para evitar entradas excesivamente largas si no son necesarias.

- Validar la entrada.

- Utilizar órdenes parametrizadas.

- Utilizar procedimientos almacenados.

Los objetos **Command** pueden utilizar parámetros para pasar valores a instrucciones SQL o a procedimientos almacenados que permiten realizar operaciones de comprobación de tipos y validación. A diferencia de una sentencia SQL como cadena de texto, un parámetro se trata como un valor literal y no como código ejecutable. De esta forma, quedamos protegidos contra ataques por inyección de código SQL.

Órdenes parametrizadas

Una orden parametrizada utiliza parámetros de sustitución en el texto SQL. Los parámetros indican los valores que serán suministrados de forma dinámica, y que luego serán enviados a través de la colección **Parameters** del objeto **Command**. Por ejemplo, tomemos la sentencia SQL del ejemplo anterior:

```
SELECT nombre, telefono FROM telefonos WHERE telefono LIKE ‘98%’
```

Se convertiría en algo como esto:

```
SELECT nombre, telefono FROM telefonos WHERE telefono LIKE @prefijo
```

Los parámetros de sustitución se añaden por separado a la colección **Command.Parameters**. La sintaxis utilizada difiere de unos proveedores a otros. Con el proveedor SQL Server, las órdenes parametrizadas utilizan parámetros de sustitución con nombres únicos. Con el proveedor OLE DB, cada valor en el código se reemplaza por un signo de interrogación. En cualquier caso, hay que proporcionar un objeto **Parameter** para cada parámetro y añadirlo a la colección **Parameters**. Con el proveedor OLE DB, debe asegurarse de agregar los parámetros en el mismo orden en que aparecen en la cadena SQL. Éste no es un requisito con el proveedor de SQL Server, ya que los parámetros son nombrados.

El ejemplo siguiente reescribe la consulta que implementamos en la aplicación anterior para eliminar la posibilidad de un ataque de inyección SQL:

```
Dim Consulta As String = "SELECT nombre, telefono FROM telefonos " &
                         "WHERE telefono LIKE @prefijo"
OrdenSql = New SqlCommand(Consulta, ConexionConBD)
OrdenSql.Parameters.AddWithValue("@prefijo", ctSql.Text)
```

El parámetro de sustitución puede también ser añadido así:

```
OrdenSql.Parameters.Add("@prefijo", SqlDbType.VarChar)
OrdenSql.Parameters("@prefijo").Value = ctSql.Text
```

Esta forma de proceder ha quedado obsoleta por la posible ambigüedad con la sobrecarga de **Add** que toma un objeto **String** y un valor de enumeración **SqlDbType**, ya que el paso de un entero con la cadena podía interpretarse como el paso del valor del parámetro o del valor del **SqlDbType** correspondiente.

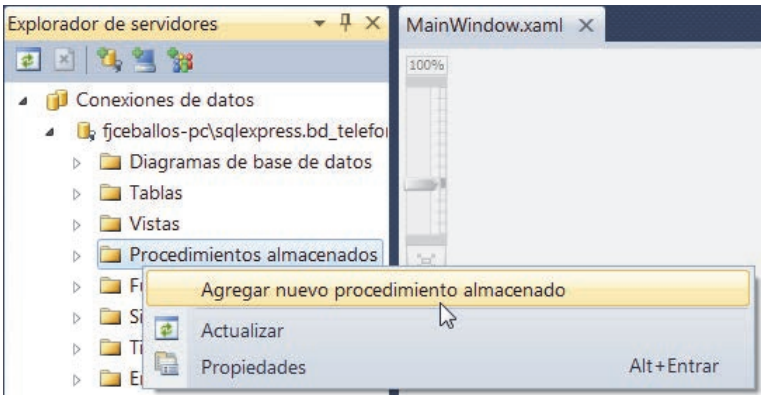
Si ahora intentamos realizar el ataque de inyección de código SQL, observaremos que no se devuelve ninguna fila. Eso es porque no hay filas cuyo teléfono empiece por el prefijo "98%' OR '9'='9". Toda la cadena es ahora el prefijo, que es lo que realmente queríamos.

Procedimientos almacenados

Un procedimiento almacenado define un bloque de sentencias SQL y se almacena en la propia base de datos. Los procedimientos almacenados son similares a las funciones, pueden aceptar datos (a través de parámetros de entrada) y devolver datos (a través de conjuntos de resultados y parámetros de salida), son fáciles de mantener, mejoran la seguridad de la aplicación, pueden llamar a otros procedimientos y mejoran el rendimiento de la aplicación.

A continuación mostramos un procedimiento almacenado creado para seleccionar los teléfonos de la tabla *telefonos* que empiecen por un determinado prefijo pasado como parámetro. Para añadirlo a la base de datos Sql Server *telefonos*, proceda según se explica a continuación:

1. Desde el EDI Visual Studio, añada una conexión a la base de datos en el explorador de servidores (también denominado explorador de base de datos). También puede añadir el procedimiento almacenado desde la herramienta de administración *SQL Server Management Studio*.



2. Expanda la base de datos y haga clic con el botón secundario del ratón en el nodo *Procedimientos almacenados*. Después seleccione la opción *Agregar nuevo procedimiento almacenado*.

3. Se añade el esqueleto del procedimiento almacenado. En el editor, haga clic con el botón secundario del ratón y seleccione *Insertar SQL* para implementar la sentencia SQL mostrada a continuación:

```
ALTER PROCEDURE stproObtenerTfnos
  @prefijo varchar(12)
AS
  SELECT nombre, telefono
  FROM telefonos
  WHERE telefono LIKE @prefijo
```

Este procedimiento almacenado toma un parámetro: un prefijo de teléfono con la intención de obtener todas las filas con ese prefijo. A continuación, modificamos la aplicación anterior para añadir el código fuente que utiliza este procedimiento almacenado:

```
OrdenSql = New SqlCommand("stproObtenerTfnos", ConexionConBD)
OrdenSql.CommandType = CommandType.StoredProcedure
OrdenSql.Parameters.AddWithValue("@prefijo", ctSql.Text)
```

La primera línea del código anterior crea un objeto **SqlCommand** que encapsula el procedimiento almacenado; obsérvese que su primer argumento es el nombre del procedimiento almacenado, la segunda línea especifica que el tipo de la sentencia es un procedimiento almacenado y la tercera añade el parámetro a la colección de parámetros de la sentencia SQL.

Finalmente, ejecute la aplicación y pruebe los resultados.

TRANSACCIONES

Una transacción es una secuencia de operaciones realizadas como una sola unidad; esto es, como una operación atómica. A este tipo de operaciones se les exige cuatro propiedades: atomicidad, coherencia, aislamiento y durabilidad (ACID), para ser calificadas como transacciones.

Una transacción debe ser una *unidad atómica* de trabajo, tanto si, cuando finaliza, se realizan todas sus modificaciones en los datos como si no se realiza ninguna de ellas; debe dejar todos los datos en un *estado coherente* (esto es, debe mantener la integridad de todos los datos); las modificaciones realizadas por transacciones simultáneas se deben *aislar* de las modificaciones llevadas a cabo por otras transacciones simultáneas (esto es, una transacción reconoce los datos en el estado en el que estaban antes de que otra transacción simultánea los modificara o después de que la segunda transacción haya concluido, pero no reconoce un estado intermedio); y una vez concluida una transacción, las modificaciones persisten (*durabilidad*) aun en el caso de producirse un error del sistema.

Por ejemplo, en una transacción bancaria consistente en transferir dinero desde una cuenta A a una cuenta B intervienen dos operaciones que se tienen que ejecutar de forma atómica: la retirada de fondos de A y el depósito en B. Si una de las dos operaciones no puede llevarse a cabo, la transacción debe cancelarse, lo que implica anular todos los cambios realizados y recuperar el estado de la base de datos previo a la transacción.

Las aplicaciones controlan las transacciones especificando cuándo se inicia y cuándo se finaliza una transacción, bien mediante sentencias SQL o bien mediante funciones de la API de bases de datos. Estas transacciones pueden ser iniciadas de forma explícita o de forma implícita.

Transacción implícita TransactionScope

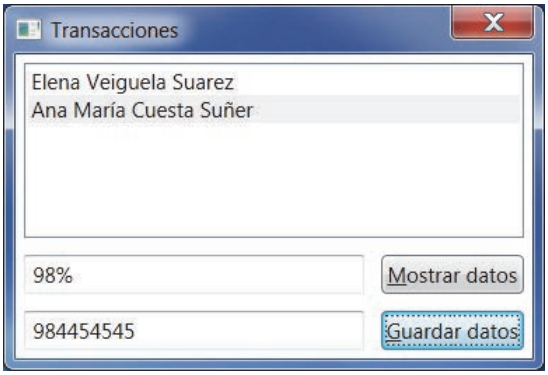
La clase **TransactionScope** proporciona una forma sencilla de marcar un bloque de código para que participe en una transacción.

```
Using tr As New TransactionScope()
  Using ConexionConBD = New SqlConnection(strConexion)
    ' Operaciones contra la base de datos
  End Using
  tr.Complete()
End Using
```

La transacción comienza una vez que se ha creado un nuevo objeto **TransactionScope**. Tal y como se ilustra en el código anterior se recomienda delimitar el

alcance de la transacción con una sentencia **Using**. Cuando la aplicación termina todo el trabajo que tiene que llevar a cabo en una transacción, se debe llamar al método **Complete**, normalmente al final del bloque **Using**, sólo una vez, para notificar al administrador de transacciones que la transacción se puede confirmar. Si no se puede llamar a este método, la transacción se anula, dado que el administrador de transacciones interpreta esto como un error del sistema.

Veamos un ejemplo. Modifiquemos la aplicación anterior para que muestre ahora la ventana siguiente. Obsérvese que hemos añadido una caja de texto que muestra el teléfono correspondiente al elemento seleccionado en la lista y un botón "Guardar datos" que permitirá modificar ese teléfono en la base de datos.



Una vez finalizado el diseño de la ventana, para facilitar la implementación de esta nueva aplicación, vamos a añadir a la misma una clase *CTelefono* con dos propiedades: *Nombre* y *Tfno*. Esto permitirá construir una colección de objetos con las filas obtenidas de la consulta a la base de datos que vincularemos a la lista.

```
Public Class CTelefono
  Public Property Nombre() As String
  Public Property Tfno() As String

  Public Sub New()
  End Sub

  Public Sub New(nom As String, tel As String)
    Nombre = nom
    Tfno = tel
  End Sub
End Class
```

El siguiente paso es construir la colección de objetos *CTelefono* con las filas del conjunto de datos obtenido tras la consulta a la base de datos. Después, asignaremos dicha colección al contexto de datos de la lista *lsTfnos*. Todo esto lo ha-

remos en el instante de realizar la consulta a la base de datos desde el controlador del evento **Click** del botón "Mostrar datos":

```
Private Sub btMostrarDatos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Try
    Using ConexionConBD = New SqlConnection(strConexion)
      ' Procedimiento almacenado
      OrdenSql = New SqlCommand("stproObtenerTfnos", ConexionConBD)
      OrdenSql.CommandType = CommandType.StoredProcedure
      OrdenSql.Parameters.AddWithValue("@prefijo", ctSql.Text)

      ' Abrir la base de datos
      ConexionConBD.Open()
      ' ExecuteReader hace la consulta y devuelve un SqlDataReader
      Lector = OrdenSql.ExecuteReader()
      ' Colección de datos obtenida de la consulta
      Dim lista As New ObservableCollection(Of CTelefono)()
      Dim nombre As String, tfno As String
      ' Llamar siempre a Read antes de acceder a los datos
      While Lector.Read() ' siguiente registro
        nombre = TryCast(Lector("nombre"), String)
        tfno = TryCast(Lector("telefono"), String)
        lista.Add(New CTelefono(nombre, tfno))
      End While
      ' Llamar siempre a Close una vez finalizada la lectura
      Lector.Close()
      ' Contextos de datos
      lsTfnos.DataContext = lista
      ctTfno.DataContext = lsTfnos
    End Using
  Catch ex As System.Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

Obsérvese que *lista* es una colección, de tipo **ObservableCollection**, de objetos *CTelefono*. Cada objeto *CTelefono* se construye a partir de cada una de las filas obtenidas tras ejecutar el método **ExecuteReader**. Una vez construida la colección, la asignamos al contexto de datos de la lista *lsTfnos* con el fin de vincularla con la propiedad **ItemsSource** de la misma para que presente los datos.

```
<ListBox Grid.ColumnSpan="2" Name="lsTfnos" Margin="5"
         ItemsSource="{Binding}" />
```

Ahora bien, según estudiamos en el capítulo anterior, un control **ListBox** cuyos elementos sean objetos de una colección, sin instrucciones concretas acerca de cómo se deben presentar esos elementos, muestra la representación de cadena

de los objetos (en nuestro caso, *Transacciones.CTelefono*). ¿Cuál es la solución? Pues utilizar un objeto **DataTemplate** para definir cómo deben mostrarse los objetos de datos o utilizar la propiedad **DisplayMemberPath** de la lista para especificar la propiedad (*Nombre* en nuestro caso) del objeto (*CTelefono*) cuyo valor debe mostrar cada elemento de la lista y la propiedad **SelectedValuePath** para guardar el valor de la propiedad del objeto (*Tfno* en nuestro caso) cuyo valor deseamos obtener del elemento seleccionado de la misma por medio de su propiedad **SelectedValue**.

```
<ListBox Grid.ColumnSpan="2" Name="lsTfnos" Margin="5"
         ItemsSource="{Binding}"
         DisplayMemberPath="Nombre" SelectedValuePath="Tfno"" />
```

El dato teléfono asociado con el elemento seleccionado de la lista lo vamos a mostrar en la nueva caja de texto, *ctTfno*, que hemos añadido, según muestra la figura anterior. De esta forma, podremos cambiar su valor y guardar el cambio en la base de datos. Para ello, hemos asignado al contexto de datos de esta caja el objeto *lsTfnos* (el control **ListBox**) para a continuación vincular su propiedad **Text** con el elemento seleccionado de *lsTfnos* (propiedad **SelectedValue**). El flujo de datos de este enlace se ha establecido para que vaya desde el origen al destino (*OneWay*) para que cuando el usuario modifique la caja de texto y ésta pierda el foco (recuerde lo estudiado en el capítulo *Introducción a WPF*: el valor predeterminado de **UpdateSourceTrigger** para la propiedad de dependencia **Text** es *LostFocus*) no pierda su contenido por no haber una correspondencia con un elemento de la lista.

```
<TextBox Grid.Row="2" Name="ctTfno" Margin="5"
         VerticalContentAlignment="Center"
         Text="{Binding SelectedValue, Mode=OneWay}" />
```

Ahora, cuando el usuario ejecute la aplicación y muestre un conjunto de nombres en la lista, seleccionará uno para conocer su teléfono o para modificarlo. La modificación de un teléfono en la base de datos implica ejecutar el siguiente procedimiento almacenado:

```
ALTER PROCEDURE stproModificarTfno
  @tfnoAntiguo varchar(12),
  @tfnoNuevo varchar(12)
AS
  UPDATE telefonos
  SET telefono=@tfnoNuevo
  WHERE telefono=@tfnoAntiguo
```

Este procedimiento será ejecutado cuando el usuario haya modificado el teléfono con la intención de cambiarlo y haga clic en el botón “Guardar datos”. Añada, por lo tanto, este procedimiento almacenado a la base de datos y después,

añada el controlador del evento **Click** del botón "Guardar datos" y edítelo como se indica a continuación:

```
Private Sub btGuardarDatos_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Try
    Dim strTelefonoAntiguo As String =
      TryCast(lsTfnos.SelectedValue, String)
    Dim strTelefonoNuevo As String = ctTfno.Text
    If strTelefonoAntiguo = strTelefonoNuevo Then Return
    Using tr As New TransactionScope()
      Using ConexionConBD = New SqlConnection(strConexion)
        ' Procedimiento almacenado
        OrdenSql = New SqlCommand("stproModificarTfno",
                                  ConexionConBD)
        OrdenSql.CommandType = CommandType.StoredProcedure
        OrdenSql.Parameters.AddWithValue("@tfnoAntiguo",
                                         strTelefonoAntiguo)
        OrdenSql.Parameters.AddWithValue("@tfnoNuevo",
                                         strTelefonoNuevo)

        ' Abrir la base de datos
        ConexionConBD.Open()
        ' Ejecutar la sentencia SQL
        OrdenSql.ExecuteNonQuery()
      End Using
      tr.Complete()
    End Using
    ' Actualizar los datos de la lista
    btMostrarDatos_Click(sender, e)
  Catch ex As System.Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

Este método, para realizar una modificación en la base de datos, inicia una transacción y cuando el método termina todo el trabajo que tiene que llevar a cabo en dicha transacción, llama al método **Complete** para notificar al administrador de transacciones que la transacción se puede confirmar. Si no se puede llamar a este método, se anulará la transacción automáticamente, dado que el administrador de transacciones interpretará esto como un error.

Transacciones explícitas

Una transacción explícita es aquélla que define explícitamente el inicio y el final de la transacción. Puede especificarse utilizando las interfaces de programación de aplicaciones (API) ADO.NET, OLE DB y ODBC o utilizando sentencias SQL.

Por ejemplo, en el proveedor **SqlCliente** de ADO.NET utilizaremos el método **BeginTransaction** de **SqlConnection** para iniciar una transacción y para finalizarla, llamaremos a los métodos **Commit** (finalizar una transacción correctacorrectamente) o **Rollback** (eliminar una transacción en la que se encontraron errores). Según esto, el método *btGuardarDatos_Click* implementado anteriormente también podría escribirse así:

```
Private Sub btGuardarDatos_Click(sender As Object,
e As RoutedEventArgs)
  Try
    Dim strTelefonoAntiguo As String =
      TryCast(lsTfnos.SelectedValue, String)
    Dim strTelefonoNuevo As String = ctTfno.Text
    If strTelefonoAntiguo = strTelefonoNuevo Then Return
    Dim transaccion As SqlTransaction = Nothing
    Try
      Using ConexionConBD = New SqlConnection(strConexion)
        ' Procedimiento almacenado
        OrdenSql = New SqlCommand("stproModificarTfno",
                                  ConexionConBD)
        OrdenSql.CommandType = CommandType.StoredProcedure
        OrdenSql.Parameters.AddWithValue("@tfnoAntiguo",
                                         strTelefonoAntiguo)
        OrdenSql.Parameters.AddWithValue("@tfnoNuevo",
                                         strTelefonoNuevo)
        ' Abrir la base de datos
        ConexionConBD.Open()
        ' Iniciar una transacción
        transaccion = ConexionConBD.BeginTransaction()
        OrdenSql.Transaction = transaccion
        ' Ejecutar la sentencia SQL
        OrdenSql.ExecuteNonQuery()
        ' Finalizar la transacción
        transaccion.Commit()
      End Using
    Catch
      ' En caso de error, volver al estado inicial
      transaccion.Rollback()
    End Try
    ' Actualizar los datos de la lista
    btMostrarDatos_Click(sender, e)
  Catch ex As System.Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

La propiedad **Transaction** de **SqlCommand**, en este caso, establece la transacción **SqlTransaction** en la que se ejecuta **SqlCommand**.

Las transacciones pueden también establecerse en los procedimientos almacenados utilizando instrucciones SQL:

- **BEGIN TRANSACTION**. Marca el punto de inicio de una transacción explícita para una conexión.

- **COMMIT TRANSACTION** o **COMMIT WORK**. Se utiliza para finalizar una transacción correctamente si no hubo errores, llevando a efecto las modificaciones correspondientes en la base de datos.

- **ROLLBACK TRANSACTION** o **ROLLBACK WORK**. Se utiliza para eliminar una transacción en la que se encontraron errores, volviendo al estado inicial, esto es, al que había antes de iniciar la transacción.

Por ejemplo, en lugar de establecer la transacción en el método *btGuardarDatos_Click* implementado anteriormente, podríamos establecerla explícitamente en el propio procedimiento almacenado así:

```
ALTER PROCEDURE stproModificarTfno
  @tfnoAntiguo varchar(12),
  @tfnoNuevo varchar(12)
AS
  BEGIN TRY
    BEGIN TRANSACTION
      UPDATE telefonos
      SET telefono=@tfnoNuevo
      WHERE telefono=@tfnoAntiguo
    COMMIT
  END TRY
  BEGIN CATCH
    IF (@@TRANCOUNT > 0)
      ROLLBACK
    -- Lanzar una excepción indicando el error ocurrido
    DECLARE @ErrMsg nvarchar(4000), @ErrSeverity int
    SELECT @ErrMsg = ERROR_MESSAGE(),
           @ErrSeverity = ERROR_SEVERITY()
    RAISERROR(@ErrMsg, @ErrSeverity, 1)
  END CATCH
```

Este ejemplo verifica @@TRANCOUNT para determinar si la transacción está en proceso (es el contador de transacciones para la conexión actual; BEGIN TRANSACTION incrementa este contador y ROLLBACK o COMMIT lo decrementa). Y RAISERROR permite enviar una notificación del error ocurrido para informar al usuario, que se traducirá en una excepción de tipo **SqlException**.

También puede utilizar transacciones explícitas en OLE DB. En este caso, llamaremos al método **ITransactionLocal::StartTransaction** para iniciar una transacción y a **ITransaction::Commit** o **ITransaction::Abort** con **fRetaining** establecido en FALSE para finalizar la transacción sin iniciar otra automáticamente.

CONSTRUIR COMPONENTES DE ACCESO A DATOS

En aplicaciones profesionales, el código de acceso a la base de datos está encapsulado en clases dedicadas a este tipo de operaciones. De esta forma, la aplicación, cliente de la base de datos, cuando tiene que ejecutar una operación contra la misma, crea un objeto de la clase adecuada y llama al método apropiado.

A la hora de crear estos componentes de acceso a datos es aconsejable seguir unas reglas básicas:

- Abrir y cerrar las conexiones rápidamente para favorecer la escalabilidad.
- Implementar código para manipular posibles errores.
- Seguir prácticas de diseño sin estado; esto es, toda la información necesaria para un método debe ser pasada a través de sus parámetros y el valor obtenido debe también ser retornado por el mismo.
- Cada consulta realizada a la base de datos debe recuperar sólo las columnas necesarias.
- Un buen diseño sugiere crear una clase para cada tabla de la base de datos o para un grupo lógicamente relacionado de tablas.
- Implementar un método por cada operación de inserción, borrado, modificación o selección.
- Utilizar procedimientos almacenados para cada operación sobre la base de datos.

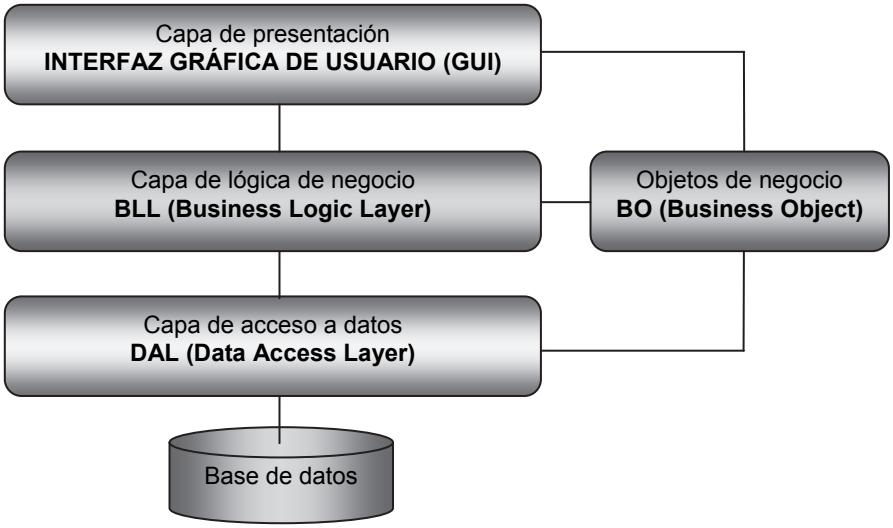
La figura mostrada un poco más adelante describe el modelo de datos que estamos proponiendo.

La base de datos incluirá los procedimientos almacenados necesarios para todas las operaciones que se desee realizar.

La capa de acceso a datos es la capa que interactúa con la base de datos. Como hemos sugerido anteriormente, su diseño está basado en la creación de una clase por cada tabla en la base de datos. Cada una de estas clases deberá definir un método por cada procedimiento almacenado relacionado con la tabla en cuestión.

Para intercambiar información entre la capa de acceso a datos y la capa de presentación se utilizarán objetos de negocio. Esta técnica está fundamentada en

el patrón DTO (*Data Transfer Object*), también conocido como *Value Object* (VO). Un objeto de negocio se construye a partir de una clase que, generalmente, sólo contiene propiedades que se corresponden, en la mayoría de los casos, con las columnas que define cada tabla. Su misión principal es servir de contenedor destinado exclusivamente a la transferencia de información.



Opcionalmente, podemos añadir otra capa con más lógica de negocio. En esta capa, cuando sea preciso, se pueden comprobar las reglas de negocio y los datos devueltos por la capa de acceso a datos, manipulándolos si es necesario, antes de enviarlos a la capa de presentación, para lo cual podrá utilizar los objetos de negocio. Cada clase de esta capa, normalmente, contendrá la misma interfaz pública que su clase análoga en la capa de acceso a datos. Estos métodos realizarán funciones de validación y de filtrado de datos antes de llamar a sus homólogos en la capa de acceso a datos.

La capa de presentación implementa la interfaz gráfica que será presentada por la aplicación al usuario.

El siguiente ejemplo demuestra cómo implementar este modelo de datos que acabamos de describir.

Capa de presentación

Deseamos realizar una aplicación que muestre una ventana como la de la figura siguiente:



Obsérvese que, básicamente, la ventana tiene un control **DataGrid** que permitirá: presentar los datos de la tabla *telefonos* de la base de datos *bd_telefonos*, modificar esos datos, borrar filas y añadir nuevas filas. El usuario realizará todas estas operaciones sobre el control **DataGrid** y, evidentemente, serán reflejadas sobre la base de datos. El **DataGrid** presentará dos columnas, nombre y teléfono, y cada fila, al hacer clic sobre ella, mostrará una sección de detalles con los datos dirección y observaciones correspondientes al nombre y teléfono seleccionados. Según lo expuesto, a falta de establecer los enlaces a datos, el diseño podría ser así:

```
<Window ...>
  <Grid>
    <StackPanel>
      <TextBlock Name="tbTitulo" FontSize="20" Margin="4"
                 Text="Lista de teléfonos:"/>
      <DataGrid Name="dgTelefonos" Margin="5" MaxHeight="230"
              AutoGenerateColumns="False"
              ItemsSource=...>
        <DataGrid.Columns>
          <DataGridTextColumn Header="Nombre" MinWidth="240"/>
          <DataGridTextColumn Header="Teléfono" MinWidth="120"/>
        </DataGrid.Columns>
        <!-- Sección de detalles de la fila -->
        <DataGrid.RowDetailsTemplate>
          <DataTemplate>
            <Border BorderThickness="0"
                    Background="BlanchedAlmond" Padding="5">
              <StackPanel>
                <TextBox Name="ctDir" Padding="2" />
```

```
                <TextBox Name="ctObs" Padding="2" />
              </StackPanel>
            </Border>
          </DataTemplate>
        </DataGrid.RowDetailsTemplate>
      </DataGrid>
    </StackPanel>
  </Grid>
</Window>
```

Operaciones contra la base de datos

El siguiente paso es analizar qué operaciones deseamos realizar contra la base de datos y escribir los procedimientos almacenados correspondientes. Según hemos indicado anteriormente, la aplicación deberá mostrar las filas de la tabla *telefonos* y permitir añadir, borrar o modificar cualquier fila de dicha tabla, por lo tanto, las operaciones que vamos a implementar son:

1. Obtener la fila de la tabla correspondiente a un número de teléfono determinado, lo cual requiere ejecutar el siguiente procedimiento almacenado:

```
ALTER PROCEDURE dbo.stproObtenerFilaTfnos
    @telefono varchar(12)
AS
    SELECT nombre, direccion, telefono, observaciones
    FROM telefonos
    WHERE telefono=@telefono
```

2. Mostrar todas las filas de la tabla *telefonos*, lo cual exige recuperarlas de la misma, operación que realiza el siguiente procedimiento almacenado:

```
ALTER PROCEDURE dbo.stproObtenerFilasTfnos
AS
    SELECT nombre, direccion, telefono, observaciones
    FROM teléfonos
```

3. Insertar una fila en la tabla, operación que realiza el siguiente procedimiento almacenado:

```
ALTER PROCEDURE dbo.stproInsertarFilaTfnos
    @nombre varchar(30),
    @direccion varchar(30),
    @telefono varchar(12),
    @observaciones varchar(240)
AS
    INSERT INTO telefonos
               (nombre, direccion, telefono, observaciones)
```

```
    VALUES (@nombre, @direccion, @telefono, @observaciones)
```

4. Borrar una fila de la tabla, operación que realiza el siguiente procedimiento almacenado:

```
ALTER PROCEDURE dbo.stproBorrarFilaTfnos
    @telefono varchar(12)
AS
    DELETE FROM telefonos
    WHERE (telefono = @telefono)
```

5. Modificar una fila de la tabla. Esta operación requiere ejecutar uno de los procedimientos almacenados siguientes, dependiendo de que se modifique o no la clave *telefono*:

```
ALTER PROCEDURE dbo.stproActualizarNomDirObs
    @nombre varchar(30),
    @direccion varchar(30),
    @telefono varchar(12),
    @observaciones varchar(240)
AS
    UPDATE telefonos
    SET nombre = @nombre, direccion = @direccion,
        observaciones = @observaciones
    WHERE (telefono = @telefono)

ALTER PROCEDURE dbo.stproActualizarNomDirTfnObs
    @nombre varchar(30),
    @direccion varchar(30),
    @tfnoAntiguo varchar(12),
    @tfnoNuevo varchar(12),
    @observaciones varchar(240)
AS
    UPDATE telefonos
    SET nombre = @nombre, direccion = @direccion,
        telefono = @tfnoNuevo, observaciones = @observaciones
    WHERE (telefono = @tfnoAntiguo)
```

Objetos de negocio

Para facilitar el intercambio de datos entre la capa de presentación y la capa de acceso a datos vamos a crear una clase *CTelefonosBO* que proporcione todas las columnas de la tabla *telefonos* como propiedades públicas, a las que añadiremos una más para seguir la pista a las modificaciones realizadas sobre el conjunto de datos recuperado. Esta clase puede escribirse así:

```
Imports System.ComponentModel
```

```
Class CTelefonoBO
  Implements INotifyPropertyChanged
  Private _nombre As String
  Public Property Nombre() As String
    Get
      Return _nombre
    End Get
    Set(value As String)
      _nombre = value
      OnPropertyChanged(New PropertyChangedEventArgs("Nombre"))
    End Set
  End Property

  Private _direccion As String
  Public Property Direccion() As String
    Get
      Return _direccion
    End Get
    Set(value As String)
      _direccion = value
      OnPropertyChanged(New PropertyChangedEventArgs("Direccion"))
    End Set
  End Property

  Private _telefono As String
  Public Property Telefono() As String
    Get
      Return _telefono
    End Get
    Set(value As String)
      _telefono = value
      OnPropertyChanged(New PropertyChangedEventArgs("Telefono"))
    End Set
  End Property

  Private _observaciones As String
  Public Property Observaciones() As String
    Get
      Return _observaciones
    End Get
    Set(value As String)
      _observaciones = value
      OnPropertyChanged(New PropertyChangedEventArgs("Observaciones"))
    End Set
  End Property

  Public Sub New()
  End Sub
  Public Sub New(nom As String, dir As String, tel As String,
  Optional obs As String = Nothing)
```

```
    Nombre = nom
    Direccion = dir
    Telefono = tel
    Observaciones = obs
  End Sub

  Private _modificado As Boolean
  Public Property Modificado() As Boolean
    Get
      Return _modificado
    End Get
    Set(value As Boolean)
      _modificado = value
    End Set
  End Property

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged
  Public Sub OnPropertyChanged(e As PropertyChangedEventArgs)
    _modificado = True
    RaiseEvent PropertyChanged(Me, e) ' generar evento
  End Sub
End Class
```

La clase *CTelefonoBO* implementa la interfaz **INotifyPropertyChanged** para notificar a los clientes enlazados de los cambios que se produzcan en sus propiedades. La interfaz de esta clase la componen cinco propiedades: *Nombre*, *Direccion*, *Telefono*, *Observaciones* y *Modificado*. Las cuatro primeras se corresponden con las columnas de la tabla *telefonos* y la quinta permitirá seguir la pista a las modificaciones realizadas sobre el conjunto de datos inicialmente recuperado de la base de datos; esto es, permitirá conocer qué filas se han añadido o modificado, información necesaria para actualizar la base de datos en un instante determinado.

Para trabajar con colecciones de objetos *CTelefonoBO* añadiremos también una clase *ColCTelefonos* derivada de **ObservableCollection(Of *T*)** que implementa las interfaces **INotifyCollectionChanged** e **INotifyPropertyChanged** que permiten utilizar enlaces dinámicos que actualizan la IU de forma automática.

```
Imports System.Collections.ObjectModel

Class ColCTelefonos
  Inherits ObservableCollection(Of CTelefonoBO)

End Class
```

Capa de acceso a datos

La capa de acceso a datos implementará las clases que proporcionen la interfaz necesaria para interactuar con la base de datos. En nuestro caso, vamos a crear una clase denominada *CTelefonoDAL* que defina un método por cada procedimiento almacenado que implementamos anteriormente para satisfacer las operaciones relacionadas con la tabla *telefonos*. Esta clase puede escribirse así:

```
Imports System.Data.SqlClient
Imports System.Data
Class CTelefonoDAL
  Private strConexion As String

  Public Sub New()
    ' Obtener la cadena de conexión
    strConexion = "Data Source=.\sqlexpress;" &
      "Initial Catalog=bd_telefonos; Integrated Security=True"
  End Sub

  Public Sub New(strCon As String)
    ' Establecer la cadena de conexión especificada
    strConexion = strCon
  End Sub

  Public Function ObtenerFilaTfnos(tfno As String) As CTelefonoBO

  End Function

  Public Function ObtenerFilasTfnos() As ColCTelefonos

  End Function

  Public Function InsertarFilaTfnos(bo As CTelefonoBO) As String

  End Function

  Public Sub BorrarFilaTfnos(tfno As String)

  End Sub

  Public Sub ActualizarNomDirTfnObs(bo As CTelefonoBO,
  tfnoAntiguo As String)

  End Sub

  Public Sub ActualizarNomDirObs(bo As CTelefonoBO)

  End Sub
End Class
```

Observamos que la clase *CTelefonoDAL* implementa dos constructores, que permiten establecer la cadena de conexión con la base de datos, y los métodos *ObtenerFilaTfnos*, *ObtenerFilasTfnos*, *InsertarFilaTfnos*, *BorrarFilaTfnos*, *ActualizarNomDirTfnObs* y *ActualizarNomDirObs*.

El método *ObtenerFilaTfnos* recupera la fila de la tabla *telefonos* de la base de datos que se corresponda con el teléfono pasado como argumento y devuelve el objeto *CTelefonoBO* correspondiente:

```
Public Function ObtenerFilaTfnos(tfno As String) As CTelefonoBO
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproObtenerFilaTfnos",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
      ' Parámetros
      OrdenSql.Parameters.AddWithValue("@telefono", tfno)
      ' Abrir la base de datos
      Conexion.Open()
      Dim lector As SqlDataReader = OrdenSql.ExecuteReader()
      If lector.Read() Then
        Dim fila As New CTelefonoBO( _
          DirectCast(lector("nombre"), String), _
          DirectCast(lector("direccion"), String), _
          DirectCast(lector("telefono"), String), _
          DirectCast(lector("observaciones"), String))
        Return fila
      End If
      Return Nothing
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error SELECT telefonos por ID")
  End Try
End Function
```

ObtenerFilasTfnos devuelve una colección de objetos *CTelefonoBO* correspondientes a todas las filas de la tabla *telefonos*:

```
Public Function ObtenerFilasTfnos() As ColCTelefonos
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproObtenerFilasTfnos",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
      ' Crear una colección para todos los teléfonos
      Dim colTelefonos As New ColCTelefonos()
      ' Abrir la base de datos
      Conexion.Open()
      Dim lector As SqlDataReader = OrdenSql.ExecuteReader()
```

```
      While lector.Read()
        Dim fila As New CTelefonoBO( _
          DirectCast(lector("nombre"), String), _
          DirectCast(lector("direccion"), String), _
          DirectCast(lector("telefono"), String), _
          DirectCast(lector("observaciones"), String))
        colTelefonos.Add(fila)
      End While
      Return colTelefonos
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error SELECT telefonos")
  End Try
End Function
```

InsertarFilaTfnos añade una nueva fila a la tabla *telefonos*, la correspondiente al objeto *CTelefonoBO* pasado como argumento:

```
Public Function InsertarFilaTfnos(bo As CTelefonoBO) As String
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproInsertarFilaTfnos",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
      ' Parámetros
      OrdenSql.Parameters.AddWithValue("@nombre", bo.Nombre)
      OrdenSql.Parameters.AddWithValue("@direccion", bo.Direccion)
      OrdenSql.Parameters.AddWithValue("@telefono", bo.Telefono)
      OrdenSql.Parameters.AddWithValue("@observaciones",
        bo.Observaciones)
      ' Abrir la base de datos
      Conexion.Open()
      OrdenSql.ExecuteNonQuery()
      Return DirectCast(OrdenSql.Parameters("@telefono").Value, String)
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error INSERT telefonos")
  End Try
End Function
```

BorrarFilaTfnos borra la fila de la tabla *telefonos* que se corresponde con el teléfono pasado como argumento:

```
Public Sub BorrarFilaTfnos(tfno As String)
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproBorrarFilaTfnos",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
```

```
      ' Parámetros
      OrdenSql.Parameters.AddWithValue("@telefono", tfno)
      ' Abrir la base de datos
      Conexion.Open()
      OrdenSql.ExecuteNonQuery()
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error DELETE telefonos")
  End Try
End Sub
```

ActualizarNomDirTfnObs actualiza la fila de la tabla *telefonos* que se corresponde con el teléfono pasado como argumento, con los datos del objeto *CTelefonoBO* pasado también como argumento:

```
Public Sub ActualizarNomDirTfnObs(bo As CTelefonoBO, tfnoAntiguo As
String)
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproActualizarNomDirTfnObs",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
      ' Parámetros
      OrdenSql.Parameters.AddWithValue("@nombre", bo.Nombre)
      OrdenSql.Parameters.AddWithValue("@direccion", bo.Direccion)
      OrdenSql.Parameters.AddWithValue("@tfnoAntiguo", tfnoAntiguo)
      OrdenSql.Parameters.AddWithValue("@tfnoNuevo", bo.Telefono)
      OrdenSql.Parameters.AddWithValue("@observaciones",
        bo.Observaciones)
      ' Abrir la base de datos
      Conexion.Open()
      OrdenSql.ExecuteNonQuery()
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error INSERT telefonos")
  End Try
End Sub
```

Y *ActualizarNomDirObs* es análoga a *ActualizarNomDirTfnObs* excepto en que no modifica la columna *telefono*:

```
Public Sub ActualizarNomDirObs(bo As CTelefonoBO)
  Try
    Using Conexion As New SqlConnection(strConexion)
      Dim OrdenSql As New SqlCommand("stproActualizarNomDirObs",
                                     Conexion)
      OrdenSql.CommandType = CommandType.StoredProcedure
      ' Parámetros
      OrdenSql.Parameters.AddWithValue("@nombre", bo.Nombre)
```

```
      OrdenSql.Parameters.AddWithValue("@direccion", bo.Direccion)
      OrdenSql.Parameters.AddWithValue("@telefono", bo.Telefono)
      OrdenSql.Parameters.AddWithValue("@observaciones",
        bo.Observaciones)
      ' Abrir la base de datos
      Conexion.Open()
      OrdenSql.ExecuteNonQuery()
    End Using
  Catch err As SqlException
    Throw New ApplicationException("Error INSERT telefonos")
  End Try
End Sub
```

Capa de lógica de negocio

Opcionalmente, podemos añadir otra capa de lógica de negocio que interactúe con la capa de presentación y la capa de acceso a datos. Cada clase de esta capa, normalmente, contendrá la misma interfaz pública que su clase análoga en la capa de acceso a datos y sus métodos realizarán, si es preciso, funciones de validación y de filtrado de datos antes de llamar a sus homólogos en la capa de acceso a datos. A modo de ejemplo, vamos a añadir una clase *CTelefonoBLL* cuyos métodos, en principio, simplemente llamarán a sus homólogos de la capa de acceso a datos:

```
Class CTelefonoBLL
  Private bd As New CTelefonoDAL()

  Public Function ObtenerFilaTfnos(tfno As String) As CTelefonoBO
    Return bd.ObtenerFilaTfnos(tfno)
  End Function

  Public Function ObtenerFilasTfnos() As ColCTelefonos
    Dim coTfnos As ColCTelefonos = bd.ObtenerFilasTfnos()
    Return coTfnos
  End Function

  Public Function InsertarFilaTfnos(bo As CTelefonoBO) As String
    Return bd.InsertarFilaTfnos(bo)
  End Function

  Public Sub BorrarFilaTfnos(tfno As String)
    bd.BorrarFilaTfnos(tfno)
  End Sub

  Public Sub ActualizarNomDirTfnObs(bo As CTelefonoBO,
  tfnoAntiguo As String)
    bd.ActualizarNomDirTfnObs(bo, tfnoAntiguo)
  End Sub
```

```
  Public Sub ActualizarNomDirObs(bo As CTelefonoBO)
    bd.ActualizarNomDirObs(bo)
  End Sub
End Class
```

Lógica de interacción con la capa de presentación

Para que la ventana de nuestra aplicación muestre en el **DataGrid** los datos de la tabla *telefonos* de la base de datos *bd_telefonos*, tendremos que seleccionar dichos datos, almacenarlos en una colección y asignar dicha colección al contexto de datos del **DataGrid**. Esta operación la podemos realizar como respuesta al evento **Loaded** de la ventana principal según se indica a continuación:

```
Class MainWindow
  Private bd As New CTelefonoBLL()
  Private coTfnos As ColCTelefonos

  Private Sub Window_Loaded(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    Try
      ' Contextos de datos
      coTfnos = bd.ObtenerFilasTfnos()
      dgTelefonos.DataContext = coTfnos
    Catch ex As System.Exception
      MessageBox.Show(ex.Message)
    End Try
  End Sub
End Class
```

Obsérvese que la clase *MainWindow* define dos atributos: *bd*, objeto de la clase *CTelefonoBLL* cuya interfaz proporciona los métodos para el acceso a la capa de acceso a datos, y *coTfnos*, colección de objetos *CTelefonoBO* que se corresponderán con las filas de la tabla *telefonos*. Esta colección será obtenida invocando al método *ObtenerFilasTfnos* de *bd* después de que se haya presentado la ventana *MainWindow* y será asignada al contexto de datos del **DataGrid**. De esta forma, el **DataGrid** podrá establecer un enlace con dicha colección, lo que permitirá a sus columnas y a la sección de detalles de cada fila establecer un enlace con las propiedades a mostrar de los objetos *CTelefonoBO* de la colección según se indica a continuación:

```
<DataGrid Name="dgTelefonos" Margin="5" MaxHeight="230"
      AutoGenerateColumns="False"
      ItemsSource="{Binding}">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Nombre"
      Binding="{Binding Path=Nombre}" MinWidth="240"/>
    <DataGridTextColumn Header="Teléfono"
```

```
      Binding="{Binding Path=Telefono}" MinWidth="120"/>
  </DataGrid.Columns>
  <!-- Sección de detalles de la fila -->
  <DataGrid.RowDetailsTemplate>
    <DataTemplate>
      <Border BorderThickness="0"
              Background="BlanchedAlmond" Padding="5">
        <StackPanel>
          <TextBox Name="ctDir" Padding="2"
            Text="{Binding Path=Direccion,
                           UpdateSourceTrigger=PropertyChanged}" />
          <TextBox Name="ctObs" Padding="2"
            Text="{Binding Path=Observaciones,
                           UpdateSourceTrigger=PropertyChanged}" />
        </StackPanel>
      </Border>
    </DataTemplate>
  </DataGrid.RowDetailsTemplate>
</DataGrid>
```

El **DataGrid**, cuya propiedad *ItemsSource="{Binding}"* especifica que se utilizará como origen de datos el contexto actual, presenta dos columnas, "Nombre" y "Teléfono", enlazadas con las propiedades *Nombre* y *Telefono* del objeto *CTelefonoBO* y, a su vez, cada fila, al hacer clic sobre ella, mostrará una sección de detalles con dos cajas de texto enlazadas con las propiedades *Direccion* y *Observaciones* del objeto *CTelefonoBO*; estas dos últimas propiedades serán actualizadas cada vez que cambie la propiedad **Text** (*UpdateSourceTrigger=PropertyChanged*).

Ejecute ahora la aplicación y pruebe a modificar, añadir y borrar filas; observará que todo funciona como esperaba. Después observe la propiedad *Modificado* de los objetos *CTelefonoBO* de la colección *coTfnos* vinculada con el contexto de datos; su valor será **True** para los objetos añadidos y modificados, información necesaria para actualizar la base de datos en un instante determinado, por ejemplo, al cerrar la aplicación.

Pero, ¿qué pasa con los objetos borrados? Cada vez que eliminamos una fila del **DataGrid** se elimina el objeto *CTelefonoBO* correspondiente de la colección *coTfnos*. Pues bien, podemos eliminar simultáneamente la fila correspondiente de la tabla *telefonos*. Para ello, vamos a añadir a la ventana *MainWindow* el controlador del evento **SelectionChanged** del **DataGrid**:

```
Private Sub dgTelefonos_SelectionChanged(sender As System.Object,
e As System.Windows.Controls.SelectionChangedEventArgs)

End Sub
```

El segundo parámetro de este método, de tipo **SelectionChangedEventArgs**, proporciona a través de su propiedad **AddedItems** la lista que contiene los elementos que se seleccionaron (si la selección es simple habrá sólo un elemento) y a través de la propiedad **RemovedItems**, la lista que contiene los elementos cuya selección se anuló. Según esto, si la selección es de una fila cada vez, cuando el usuario seleccione la fila a borrar, la lista **AddedItems** contendrá un elemento y la lista **RemovedItems**, cero elementos y cuando pulse la tecla *Supr* para eliminar la fila seleccionada, la lista **AddedItems** contendrá cero elementos y la lista **RemovedItems**, un elemento, el objeto *CTelefonoBO* eliminado. Según esto, podemos completar el método anterior así:

```
Private Sub dgTelefonos_SelectionChanged(sender As System.Object,
e As System.Windows.Controls.SelectionChangedEventArgs)
  If e.RemovedItems.Count = 0 Then Return
  Dim obj As CTelefonoBO = TryCast(e.RemovedItems(0), CTelefonoBO)
  ' Cuando se añade un nuevo elemento, obj es Nothing
  If obj IsNot Nothing AndAlso e.AddedItems.Count = 0 AndAlso
     e.RemovedItems.Count <> 0 Then
    bd.BorrarFilaTfnos(obj.Telefono)
  End If
End Sub
```

Y, ¿cuándo actualizamos la tabla *telefonos* de la base de datos con el resto de las filas modificadas? Podemos hacerlo cuando se vaya a cerrar la ventana, esto es, como respuesta al evento **Closing** de *MainWindow*. Según lo expuesto, añada el método que responde a este evento y complételo como se indica a continuación:

```
Private Sub Window_Closing(sender As System.Object,
e As System.ComponentModel.CancelEventArgs)
  For i As Integer = 0 To coTfnos.Count - 1
    If coTfnos(i).Modificado Then
      ' Insertar
      If bd.ObtenerFilaTfnos(coTfnos(i).Telefono) Is Nothing Then
        bd.InsertarFilaTfnos(coTfnos(i))
      Else
        ' Actualizar
        bd.ActualizarNomDirObs(coTfnos(i))
      End If
    End If
  Next
End Sub
```

Ahora bien, después de ejecutarse el método anterior, podremos observar que cuando se modifica el número de teléfono de una fila, esa fila aparece en la base de datos con el número antiguo y con el número nuevo. Lógico, cuando modificamos el número de teléfono, la colección de objetos *CTelefonoBO* queda actualizada y esa fila figurará como modificada, pero la base de datos seguirá

conservando la fila con el teléfono antiguo. Este caso lo podemos solucionar borrando de la base de datos la fila con el teléfono antiguo justo a continuación de haberla modificado, lo que nos exigirá conservar el teléfono antiguo en alguna variable, por ejemplo en un atributo *tfnoAntesDeModificar* de *MainWindow*. Según lo expuesto añada al controlador *dgTelefonos_SelectionChanged* el código indicado a continuación:

```
Private Sub dgTelefonos_SelectionChanged(sender As System.Object,
e As System.Windows.Controls.SelectionChangedEventArgs)
  ' ...

  If obj IsNot Nothing AndAlso
     tfnoAntesDeModificar IsNot Nothing AndAlso
     e.AddedItems.Count <> 0 AndAlso e.RemovedItems.Count <> 0 Then
    If obj.Telefono <> tfnoAntesDeModificar Then
      If bd.ObtenerFilaTfnos(tfnoAntesDeModificar) IsNot Nothing Then
        bd.BorrarFilaTfnos(tfnoAntesDeModificar)
      End If
    End If
    tfnoAntesDeModificar = Nothing
  End If
End Sub
```

Añada el atributo privado *tfnoAntesDeModificar* a la clase *MainWindow*. El valor para este atributo lo obtendremos de la propia fila que estamos modificando antes de que finalice la edición, que es cuando la fila adquirirá el nuevo valor. Para ello, simplemente tenemos que controlar el evento **RowEditEnding** que se produce antes de que la edición de la fila se confirme o se cancele. El segundo parámetro de este controlador nos proporciona la fila a través de su propiedad **Row**:

```
Private Sub dgTelefonos_RowEditEnding(sender As System.Object, e As
System.Windows.Controls.DataGridRowEditEndingEventArgs)
  tfnoAntesDeModificar = DirectCast(e.Row.Item,
    CTelefonoBO).Telefono
End Sub
```

Desacoplar la IU del resto de la aplicación

Si nos fijamos en el código correspondiente a la lógica de interacción con la capa de presentación, observamos que ahí se controlan eventos expuestos por el **DataGrid** con el fin de determinar qué operación hay que realizar en función de las operaciones que se realicen sobre la IU (interfaz de usuario). Sin embargo, al hacerlo de esta forma, estamos escribiendo código que es específico de la rejilla; esto es, ¿qué pasaría si decidimos cambiar la interfaz de usuario para que ahora presente una lista y un número de cajas de texto para mostrar los detalles del elemento seleccionado de la lista? Pues que tendríamos que volver a escribir esta ló-

gica y, quizás, los eventos del **DataGrid** no puedan satisfacer todas nuestras necesidades; por ejemplo, porque dispongamos de eventos "EditEnding", pero no "EditEnded", por lo que los datos visibles a los controladores de eventos no estarán en su estado esperado. Un enfoque mejor sería si pudiéramos adaptar nuestra colección de objetos *CTelefonoBO* de tal manera que pudiera ser asociada a cualquier IU WPF adecuada, con las operaciones de editar, añadir o eliminar sincronizadas con la base de datos a través del resto de la lógica de negocio.

Con el fin de ilustrar lo que acabamos de exponer vamos a realizar otra versión de la aplicación anterior que recoja este nuevo enfoque.

Adaptar la colección de objetos

Vamos a añadir a la colección de objetos *ColCTelefonos*, derivada de **ObservableCollection(Of** *CTelefonoBO***)**, la funcionalidad necesaria para que pueda informar a la aplicación de cuándo se inicia la edición de un objeto y cuándo finaliza la misma, para lo cual será necesario que cada objeto, a su vez, informe de estos hechos a su colección.

Lo expuesto se puede hacer de una forma sencilla si los objetos de la colección implementan la interfaz **System.ComponentModel.IEditableObject**:

```
Class CTelefonoBO
  Implements INotifyPropertyChanged, IEditableObject
  ' ...
  Public Sub BeginEdit() Implements IEditableObject.BeginEdit

  End Sub

  Public Sub CancelEdit() Implements IEditableObject.CancelEdit

  End Sub

  Public Sub EndEdit() Implements IEditableObject.EndEdit

  End Sub
End Class
```

La interfaz **IEditableObject** proporciona tres métodos que permiten confirmar o deshacer los cambios realizados en un objeto que se utiliza como origen de datos:

- **BeginEdit**. Este método se ejecuta cuando comienza la edición en un objeto.
- **CancelEdit**. Este otro método permite descartar los cambios (tecla *Esc*) que se han realizado desde la última llamada a **BeginEdit**.

- **EndEdit**. Y este otro método aplica los cambios realizados desde la última llamada a **BeginEdit**.

Según lo expuesto, vamos a hacer que el método **BeginEdit** genere un evento *ItemBeginEdit* cuando comience la edición de un objeto *CTelefonoBO* y que **EndEdit** genere otro evento *ItemEndEdit* cuando finalice la edición. De esta forma, la colección podrá interceptar estos eventos e informarse de lo que ocurrió. Para ello, añada a la clase *CTelefonoBO* el código especificado a continuación:

```
Public Delegate Sub ItemEditEventHandler(sender As IEditableObject)

Class CTelefonoBO
  Implements INotifyPropertyChanged, IEditableObject
  ' ...

  ' Eventos ItemBeginEdit e ItemEndEdit
  Public Event ItemBeginEdit As ItemEditEventHandler
  Public Event ItemEndEdit As ItemEditEventHandler

  ' Miembros de la interfaz IEditableObject
  Private Sub BeginEdit() Implements IEditableObject.BeginEdit
    RaiseEvent ItemBeginEdit(Me) ' generar evento
  End Sub

  Private Sub CancelEdit() Implements IEditableObject.CancelEdit
  End Sub

  Private Sub EndEdit() Implements IEditableObject.EndEdit
    RaiseEvent ItemEndEdit(Me) ' generar evento
  End Sub
End Class
```

Observe que ambos eventos son del tipo definido por el delegado *ItemEditEventHandler*. Los métodos **BeginEdit** y **EndEdit** generan el evento correspondiente adjuntando como información el objeto que está siendo editado.

Para que la colección de objetos *CTelefonoBO* pueda notificar a la aplicación que un determinado objeto está siendo editado, lo que tiene que hacer es capturar los eventos generados por ese objeto y generar eventos análogos por cada uno de ellos. Según esto, complete la clase colección *ColCTelefonos* como se indica a continuación:

```
Class ColCTelefonos
  Inherits ObservableCollection(Of CTelefonoBO)

  ' Eventos ItemBeginEdit e ItemEndEdit de la colección
  Public Event ItemBeginEdit As ItemEditEventHandler
  Public Event ItemEndEdit As ItemEditEventHandler
```

```
  Protected Overrides Sub InsertItem(ind As Integer,
  item As CTelefonoBO)
    ' Insertar un elemento en la posición especificada
    MyBase.InsertItem(ind, item)
    ' Asignar un controlador para los eventos ItemBeginEdit e
    ' ItemEndEdit de cada item CTelefonoBO de la colección
    AddHandler item.ItemBeginEdit, AddressOf ControladorItemBeginEdit
    AddHandler item.ItemEndEdit, AddressOf ControladorItemEndEdit
  End Sub

  Private Sub ControladorItemBeginEdit(sender As IEditableObject)
    ' Generar el evento ItemBeginEdit de la colección por cada
    ' evento ItemBeginEdit generado por un item CTelefonoBO
    RaiseEvent ItemBeginEdit(sender)
  End Sub

  Private Sub ControladorItemEndEdit(sender As IEditableObject)
    ' Generar el evento ItemEndEdit de la colección por cada
    ' evento ItemEndEdit generado por un item CTelefonoBO
    RaiseEvent ItemEndEdit(sender)
  End Sub
End Class
```

Observamos que la clase define dos eventos *ItemBeginEdit* e *ItemEndEdit* que se generarán cada vez que un objeto *CTelefonoBO* de la misma sea editado (el que los eventos de *ColCTelefonos* se llamen igual que los de *CTelefonoBO* es simplemente porque las acciones son las mismas; puede cambiarlos de nombre si lo desea). Para ello, hemos redefinido el método **InsertItem** heredado de **ObservableCollection** con el fin de asignar a los eventos *ItemBeginEdit* e *ItemEndEdit* de cada objeto de la colección los controladores que deben ejecutarse cuando se genere cada uno de ellos. El método **InsertItem** es invocado automáticamente cada vez que se inserta un elemento en el **DataGrid**.

Según lo expuesto, el controlador *ControladorItemBeginEdit* generará un evento *ItemBeginEdit* cuando comience la edición de un objeto *CTelefonoBO* y *ControladorItemEndEdit* generará otro evento *ItemEndEdit* cuando finalice la edición. De esta forma, la aplicación podrá interceptar estos eventos allí donde defina una colección, según veremos a continuación.

Finalmente, para cancelar la edición iniciada en una fila, restaurando los valores iniciales, y para asegurar que una operación de edición se ejecutará una sola vez (no dos veces consecutivas como ocurre con esta versión del **DataGrid**) modifique los métodos como se indica a continuación:

```
' Copia para cuando se cancele la operación de edición
Private copia As CTelefonoBO
Private enEdicion As Boolean = False
```

```
Public Sub BeginEdit() Implements IEditableObject.BeginEdit
  If enEdicion Then Return
  enEdicion = True
  copia = TryCast(Me.MemberwiseClone(), CTelefonoBO)
  RaiseEvent ItemBeginEdit(Me) ' generar evento
End Sub

Public Sub CancelEdit() Implements IEditableObject.CancelEdit
  If Not enEdicion Then Return
  enEdicion = False
  Me.Nombre = copia.Nombre
  Me.Direccion = copia.Direccion
  Me.Telefono = copia.Telefono
  Me.Observaciones = copia.Observaciones
  Me.Modificado = copia.Modificado
End Sub

Public Sub EndEdit() Implements IEditableObject.EndEdit
  If Not enEdicion Then Return
  enEdicion = False
  copia = Nothing
  RaiseEvent ItemEndEdit(Me) ' generar evento
End Sub
```

Capa de lógica de negocio

Esta capa, formada por la clase *CTelefonoBLL*, contiene la misma interfaz pública que su clase análoga en la capa de acceso a datos. Pero, en esta nueva versión de la aplicación, el método *ObtenerFilasTfnos*, además de obtener la colección de objetos *CTelefonoBO*, asignará a los eventos *ItemBeginEdit*, *ItemEndEdit* y **CollectionChanged** de la colección los controladores que deben ejecutarse cuando se genere cada uno de ellos:

```
Imports System.Collections.Specialized
Imports System.ComponentModel

Class CTelefonoBLL
  Private bd As New CTelefonoDAL()

  Public Function ObtenerFilaTfnos(tfno As String) As CTelefonoBO
    Return bd.ObtenerFilaTfnos(tfno)
  End Function

  Public Function ObtenerFilasTfnos() As ColCTelefonos
    Dim coTfnos As ColCTelefonos = bd.ObtenerFilasTfnos()
    AddHandler coTfnos.CollectionChanged,
      AddressOf ControladorCollectionChanged
    ' Asignar un controlador para los eventos ItemBeginEdit e
```

```
    ' ItemEndEdit del objeto colección coTfnos
    AddHandler coTfnos.ItemEndEdit, AddressOf ControladorItemEndEdit
    AddHandler coTfnos.ItemBeginEdit, AddressOf ControladorItemBeginEdit
    Return coTfnos
  End Function

  Private Sub ControladorCollectionChanged(sender As Object,
  e As NotifyCollectionChangedEventArgs)

  End Sub

  Private Sub ControladorItemBeginEdit(sender As IEditableObject)

  End Sub

  Private Sub ControladorItemEndEdit(sender As IEditableObject)

  End Sub

  Public Function InsertarFilaTfnos(bo As CTelefonoBO) As String
    Return bd.InsertarFilaTfnos(bo)
  End Function

  Public Sub BorrarFilaTfnos(tfno As String)
    bd.BorrarFilaTfnos(tfno)
  End Sub

  Public Sub ActualizarNomDirTfnObs(bo As CTelefonoBO, tfnoAntiguo
As String)
    bd.ActualizarNomDirTfnObs(bo, tfnoAntiguo)
  End Sub

  Public Sub ActualizarNomDirObs(bo As CTelefonoBO)
    bd.ActualizarNomDirObs(bo)
  End Sub
End Class
```

El evento **CollectionChanged** de **ObservableCollection** se produce cuando se agrega, quita, cambia, mueve un elemento o se actualiza la lista completa. Observe el método que controla este evento, tiene un segundo parámetro que proporciona las siguientes propiedades:

- **Action**. Acción que provocó el evento:
  - **Add**. Se agregaron uno o varios elementos.
  - **Remove**. Se quitaron uno o varios elementos.
  - **Replace**. Se reemplazaron uno o varios elementos.
  - **Move**. Se movieron uno o varios elementos dentro de la colección.
  - **Reset**. El contenido de la colección ha cambiado significativamente.

- **NewItems**. Lista de los nuevos elementos implicados en el cambio.
- **NewStartingIndex**. Índice donde se produjo el cambio.
- **OldItems**. Lista de elementos a los que afecta la acción Replace, Remove o Move.
- **OldStartingIndex**. Índice donde se produjo la acción Move, Remove o Replace.

Entonces, cuando la acción sea **Remove**, eliminaremos el elemento de la base de datos invocando al método *BorrarFilaTfnos* y cuando la acción sea **Add**, asignaremos a los atributos del objeto *CTelefonoBO* unos datos genéricos predeterminados para evitar que se queden vacíos. Según lo expuesto, implementaremos el controlador del evento **CollectionChanged** como se indica a continuación:

```
Private Sub ControladorCollectionChanged(sender As Object,
e As NotifyCollectionChangedEventArgs)
  If e.Action = NotifyCollectionChangedAction.Remove Then
    For Each item As Object In e.OldItems
      Dim obj As CTelefonoBO = TryCast(item, CTelefonoBO)
      BorrarFilaTfnos(obj.Telefono)
    Next
  ElseIf e.Action = NotifyCollectionChangedAction.Add Then
    For Each item As Object In e.NewItems
      Dim obj As CTelefonoBO = TryCast(item, CTelefonoBO)
      obj.Nombre = "nombre"
      obj.Direccion = "dirección"
      obj.Telefono = "000000000"
      obj.Observaciones = "Ninguna"
    Next
  End If
End Sub
```

Los valores predeterminados también pueden especificarse por medio de la propiedad **TargetNullValue** del objeto **Binding**, pero si lo que queremos es desacoplar la IU de la lógica de negocio, no es una buena opción. Por ejemplo:

```
<DataGridTextColumn Header="Nombre" MinWidth="240"
                    Binding="{Binding Path=Nombre,
                    TargetNullValue=(introducir un nombre)}"/>
```

Una vez implementadas las operaciones de añadir y borrar, nos queda la operación de modificar. Según vimos en la versión anterior, para el caso en el que el usuario modifique el número de teléfono de una fila del **DataGrid**, sabemos que la colección de objetos *CTelefonoBO* queda actualizada, pero la base de datos sigue conservando la fila con el teléfono antiguo, por lo que habrá que borrarla; esta operación no habrá que hacerla cuando las modificaciones no afecten al número

de teléfono. Según lo expuesto, para conocer el teléfono antiguo, añada al controlador *ControladorItemBeginEdit* el código indicado a continuación:

```
Private Sub ControladorItemBeginEdit(sender As IEditableObject)
  tfnoAnterior = TryCast(sender, CTelefonoBO).Telefono
End Sub
```

Añada el atributo privado *tfnoAnterior* a la clase *CTelefonoBLL*. El valor para este atributo lo obtendremos del objeto *CTelefonoBO* cuya edición se inicia, objeto que nos proporciona el evento *ItemBeginEdit* de la colección.

Cuando finalice la edición del objeto *CTelefonoBO*, la colección informará de ello generando el evento *ItemEndEdit* proporcionando el objeto modificado. La respuesta a este evento será borrar de la base de datos la fila que tenga por número de teléfono *tfnoAnterior* e insertar una nueva fila, la correspondiente al objeto *CTelefonoBO* modificado. En el caso de que la modificación realizada sobre el objeto *CTelefonoBO* no afecte al número de teléfono, simplemente actualizaremos la fila correspondiente de la base de datos. Según esto, escriba el controlador del evento *ItemEndEdit* de la colección como se indica a continuación:

```
Private Sub ControladorItemEndEdit(sender As IEditableObject)
  Dim obj As CTelefonoBO = TryCast(sender, CTelefonoBO)
  If obj.Modificado = False Then Return
  If ObtenerFilaTfnos(obj.Telefono) Is Nothing Then
    If ObtenerFilaTfnos(tfnoAnterior) IsNot Nothing Then
      BorrarFilaTfnos(tfnoAnterior)
    End If
    InsertarFilaTfnos(obj)
  Else
    ActualizarNomDirObs(obj)
  End If
End Sub
```

Puede ser que el usuario inicie la edición de una fila y la abandone sin realizar ninguna modificación. En este caso, la propiedad *Modificado* del objeto seguirá valiendo **False**, dato que aprovecharemos para salir de *ControladorItemEndEdit*.

Lógica de interacción con la capa de presentación

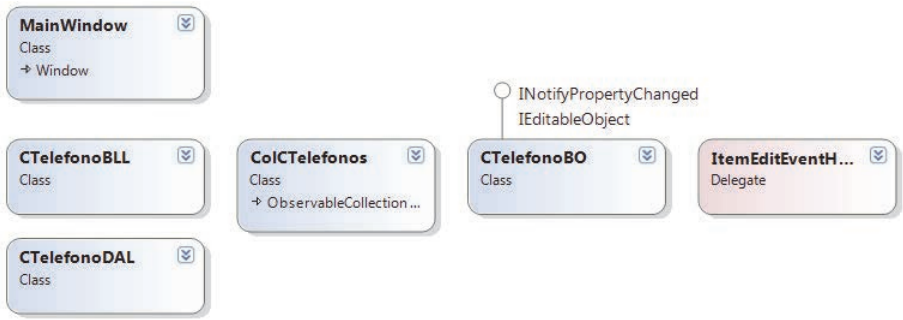
Si nos fijamos en el código correspondiente a la capa de lógica de negocio, observamos que ahí se controlan todos los eventos generados por la colección de objetos *CTelefonoBO*, resultado de las operaciones que el usuario realiza en la interfaz gráfica, con el fin de determinar qué operación hay que ejecutar sobre la base de datos. Este código no es específico de la interfaz gráfica, por lo tanto, si decidimos cambiar la IU esta lógica de negocio no se verá afectada.

Ahora, para que la ventana de nuestra aplicación muestre en el **DataGrid** los datos de la tabla *telefonos* de la base de datos *bd_telefonos*, tendremos simplemente que asignar la colección de objetos *CTelefonoBO* al contexto de datos del **DataGrid**. Esta operación la vamos a realizar en el controlador del evento **Loaded** de la ventana principal según se indica a continuación, y esto es todo:

```
Class MainWindow
  Private bd As New CTelefonoBLL()

  Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
    Try
      ' Contextos de datos
      dgTelefonos.DataContext = bd.ObtenerFilasTfnos()
    Catch ex As System.Exception
      MessageBox.Show(ex.Message)
    End Try
  End Sub
End Class
```

Como resumen, la arquitectura de la aplicación ha quedado así:



Validación

Los datos que introducimos en un control **DataGrid** pueden ser validados individualmente a nivel de celda o bien, a nivel de fila. A nivel de celda, se validan las propiedades individuales del objeto de datos enlazado y a nivel de fila, se valida el objeto de datos completo.

Para validar los valores de las celdas individualmente hay que especificar las reglas de validación en el enlace que se usa con una columna. Según estudiamos en el capítulo *Introducción a WPF*, para establecer implícitamente la regla **ExceptionValidationRule** que verifica las excepciones que se producen durante la actualización de la propiedad origen del enlace, basta con asignar el valor **True** a la propiedad **ValidatesOnExceptions** del enlace. Por ejemplo:

```
<DataGrid.Columns>
  <DataGridTextColumn Header="Nombre" MinWidth="240"
                      Binding="{Binding Path=Nombre,
                      UpdateSourceTrigger=PropertyChanged,
                      ValidatesOnExceptions=True}"/>
  <DataGridTextColumn Header="Teléfono" MinWidth="120"
                      Binding="{Binding Path=Telefono,
                      UpdateSourceTrigger=PropertyChanged,
                      ValidatesOnDataErrors=True}"/>
...
```

Para nuestro ejemplo, esta forma de proceder, sin más, no realizará ninguna notificación de error ya que las propiedades a validar son de tipo **String** lo que implica aceptar cualquier entrada del usuario. Lo que sí podemos hacer es modificar la clase *CTelefonoBO* para que lance una excepción cuando los datos no sean los esperados, excepción que será verificada por la regla **ExceptionValidationRule**. Por ejemplo, para notificar al usuario que las columnas nombre, dirección y teléfono de la fila que está editando no pueden quedarse vacías, podemos validar en este sentido las propiedades correspondientes del objeto *CTelefonoBO* enlazado, según se indica a continuación:

```
Class CTelefonoBO
  Implements INotifyPropertyChanged, IEditableObject

  Private _nombre As String
  Public Property Nombre() As String
    Get
      Return _nombre
    End Get
    Set
      Validar(value)
      _nombre = value
      OnPropertyChanged(New PropertyChangedEventArgs("Nombre"))
    End Set
  End Property

  ' ...

  ' Validación a nivel de celda
  Private Sub Validar(dato As String)
    If dato.Length = 0 Then
      Throw New ArgumentException("Esta columna no se puede dejar vacía")
    End If
  End Sub
End Class
```

Ahora, cuando un usuario deja una celda de las especificadas vacía, aparece un borde rojo alrededor de la celda (en el enlace que se usa con una columna se ha

especificado *UpdateSourceTrigger=PropertyChanged*). Esta información que se da al usuario se puede personalizar. Para el caso de un **DataGrid**, se puede asignar a la propiedad **EditingElementStyle** de la columna un estilo adecuado (dado que el control de edición de la columna se crea en tiempo de ejecución, no se puede usar la propiedad adjunta **Validation.ErrorTemplate** como hicimos en el capítulo *Introducción a WPF* con los controles simples). Para la caja de texto dirección de la sección de detalles, asignar el estilo a la propiedad **Style**. Por ejemplo:

```
<DataGrid.Columns>
  <DataGridTextColumn
    Header="Nombre" MinWidth="240"
    Binding="{Binding Path=Nombre,
    UpdateSourceTrigger=PropertyChanged,
    ValidatesOnExceptions=True}
    EditingElementStyle="{StaticResource EstiloDatosIncorrectos}"/>
...
```

El estilo *EstiloDatosIncorrectos* lo definiremos en la sección de recursos del **DataGrid**; por ejemplo, así:

```
<DataGrid.Resources>
  <Style x:Key="EstiloDatosIncorrectos"
    TargetType="{x:Type TextBox}">
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="True">
        <Setter Property="ToolTip"
          Value="{Binding RelativeSource={RelativeSource Self},
                  Path=(Validation.Errors)[0].ErrorContent}"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</DataGrid.Resources>
```

Ahora, cuando un usuario deja una celda de las especificadas vacía, el estilo agrega información sobre lo ocurrido. Observe que se usa un desencadenador para determinar si hay un error de validación, en cuyo caso se asigna a la propiedad **ToolTip** de la celda el comentario que describe el error ocurrido aportado por la excepción lanzada; en este caso: “Esta columna no se puede dejar vacía”.

Dijimos que los datos que introducimos en un control **DataGrid** pueden ser validados también a nivel de fila. Para ello, vamos a añadir al proyecto una clase derivada de **ValidationRule** que compruebe que la propiedad *Telefono* del objeto de datos enlazado contiene sólo dígitos. En la implementación del método **Validate**, se convierte el valor del parámetro *value* a un objeto **BindingGroup** para después tener acceso al objeto de datos a través de su propiedad **Items**:

```
Public Class ReglaValidacionFilaTfno
  Inherits ValidationRule

  Public Overrides Function Validate(value As Object,
  cultureInfo As Globalization.CultureInfo) As ValidationResult
    Dim fila As CTelefonoBO = TryCast(TryCast(value,
      BindingGroup).Items(0), CTelefonoBO)
    Dim tfno As Long
    If Not Long.TryParse(fila.Telefono, tfno) Then
      Return New ValidationResult(False,
                  "No es un número de teléfono válido")
    End If
    Return ValidationResult.ValidResult
  End Function
End Class
```

Un objeto de esta clase representa una regla de validación. Para aplicarla a las filas del **DataGrid**, tendremos que añadirla a la colección **RowValidationRules** del objeto **DataGrid** (la propiedad **RowValidationRules** proporciona acceso directo a la propiedad **ValidationRules** del objeto **BindingGroup** que agrupa todos los enlaces que usa el control):

```
<DataGrid.RowValidationRules>
  <local:ReglaValidacionFilaTfno ValidationStep="UpdatedValue"/>
</DataGrid.RowValidationRules>
```

En el código anterior, a la propiedad **ValidationStep** de la regla **ValidationRule** se le asigna el valor **UpdatedValue** de la enumeración **ValidationStep** para que la regla se ejecute después de que se actualice el objeto de datos enlazado.

El código anterior implica añadir a la ventana el espacio de nombres al que pertenece la clase *ReglaValidacionFilaTfno*:

```
xmlns:local="clr-namespace:ComponentesAccesoDatos"
```

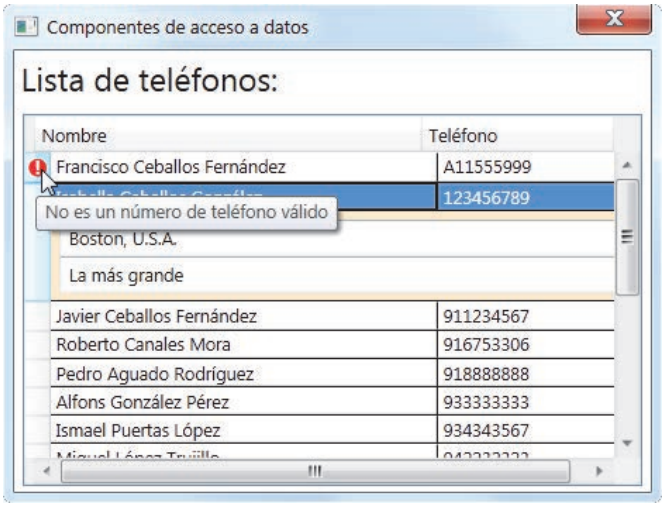
Cuando un usuario especifica un número de teléfono no válido, aparece un signo de admiración rojo (!) en el encabezado de fila. Esta información que se da al usuario se puede personalizar. Por ejemplo, podemos hacer que el símbolo de admiración (!) aparezca en blanco dentro de un círculo rojo. El mensaje de error asociado, proporcionado por la regla de validación, se mostrará al colocar el cursor del ratón sobre dicho círculo. Para ello, hay que establecer la plantilla **RowValidationErrorTemplate** a través de la propiedad del mismo nombre del **DataGrid**. En el siguiente ejemplo puede verse cómo se aplica lo expuesto:

```
<DataGrid.RowValidationErrorTemplate>
  <ControlTemplate>
    <Grid Margin="0,-1,0,-1"
```

```
      ToolTip="{Binding RelativeSource={RelativeSource
                          Mode=FindAncestor,
                          AncestorType={x:Type DataGridRow}},
                        Path=(Validation.Errors)[0].ErrorContent}">
      <Ellipse StrokeThickness="0" Fill="Red"
               Width="{TemplateBinding FontSize}"
               Height="{TemplateBinding FontSize}"/>
      <TextBlock Text="!" FontSize="{TemplateBinding FontSize}"
                 FontWeight="Bold" Foreground="White"
                 HorizontalAlignment="Center"/>
    </Grid>
  </ControlTemplate>
</DataGrid.RowValidationErrorTemplate>
```

RelativeSource describe la ubicación del origen del enlace relativa a la posición del destino del enlace y su propiedad **AncestorType** hace referencia al tipo de antecesor que se va a buscar. Esta propiedad requiere establecer la propiedad **Mode** explícitamente; por ejemplo, cuando el modo es **FindAncestor** hace referencia al antecesor en la cadena principal del elemento enlazado a datos y cuando el modo es **Self** hace referencia al elemento en el que está establecido el enlace.

Ahora, cuando ejecute la aplicación e introduzca un número de teléfono no válido el resultado será el de la figura siguiente:



ACCESO DESCONECTADO A BASE DE DATOS

En este apartado se aborda el acceso desconectado a datos: las características de ADO.NET que giran en torno a la clase **DataSet** y que permiten interactuar con los datos después de haber cerrado la conexión con el origen de datos. La clase

DataSet es una alternativa a los componentes de acceso a datos que hemos creado en el apartado anterior para acceder a los datos de una base de datos, además de proporcionar flexibilidad para navegar, filtrar y clasificar datos.

Un objeto de la clase **DataSet** representa un conjunto completo de datos (que incluye las tablas que contienen, ordenan y restringen los datos, así como las relaciones entre las tablas) residente en memoria que proporciona un modelo de programación relacional coherente independientemente del origen de los datos que contiene. Esto se traduce en que una aplicación, cuando se conecte a una base de datos, llenará el **DataSet** con la información extraída de la base de datos, para después manipular esos datos. Los cambios que a continuación se realicen en el **DataSet** no se verán reflejados en las tablas correspondientes de la base de datos. Para reflejarlos, la aplicación tendrá que conectarse de nuevo a la base de datos y aplicar sobre la misma todos los cambios realizados en el **DataSet**. Por supuesto, esta comodidad no está exenta de inconvenientes, tales como problemas de concurrencia. Dependiendo de cómo esté la aplicación diseñada, un solo error en el momento de aplicar los cambios (por ejemplo, tratando de actualizar un registro que otro usuario actualizó mientras tanto) puede desbaratar el proceso de actualización. Evidentemente, una codificación estudiada puede proteger a la aplicación de estos problemas, pero, lógicamente, requiere un esfuerzo adicional.

Según lo expuesto, la base para implementar un acceso desconectado es la clase **DataSet**. Esta clase ya fue expuesta anteriormente en este mismo capítulo. Si recuerda, un objeto de esta clase está compuesto por una colección de cero o más tablas y por otra colección de cero o más relaciones entre las tablas. Allí también se expuso la clase **DataAdapter**; un objeto de esta clase es necesario para extraer los registros de la base de datos y llenar con ellos el **DataSet**. Cada proveedor de datos proporciona su adaptador de datos; por ejemplo, Sql Server proporciona el adaptador **SqlDataAdapter**. Este adaptador actúa como puente entre la base de datos y el **DataSet** y contiene todas las órdenes SQL necesarias para consultar y modificar la base de datos.

De la clase **DataSet** destacamos los métodos siguientes:

- **Clear**. Borra cualquier dato del **DataSet** quitando todas las filas de todas las tablas.
- **Copy**. Copia la estructura y los datos de un objeto **DataSet**.
- **Clone**. Copia la estructura de **DataSet**, incluidos todos los esquemas, relaciones y restricciones de **DataTable**. No copia ningún dato.
- **Merge(DataSet)**. Combina el objeto **DataSet** especificado y su esquema en el objeto **DataSet** actual.

Y de la clase **DataAdapter** destacamos los métodos siguientes:

- **Fill**. Agrega filas a los objetos **DataTable** del **DataSet** ejecutando la consulta **SelectCommand** correspondiente. Si los objetos **DataTable** todavía no existen, se crean. Si la conexión con la base de datos está cerrada antes de llamar a **Fill**, este método la abre para recuperar datos y, a continuación, la cierra, y si está abierta, permanece abierta.

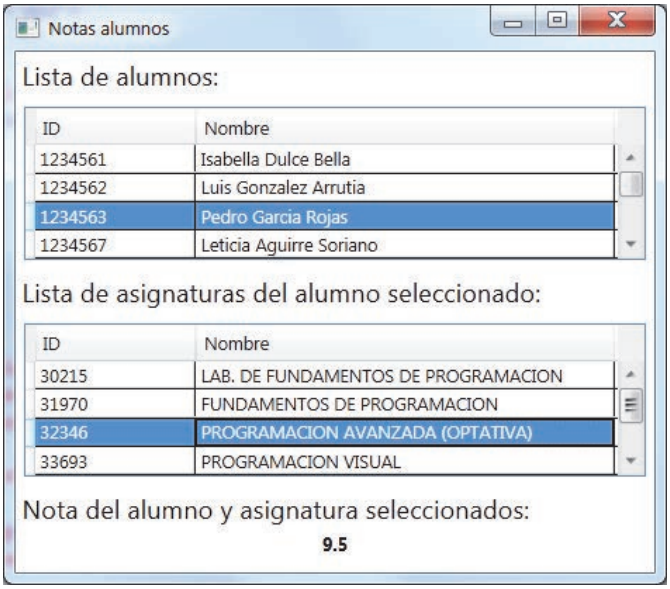
- **FillSchema**. Agrega un **DataTable** al **DataSet** especificado y configura el esquema para hacerlo coincidir con el del origen de datos en función del tipo **SchemaType** especificado. No agrega ningún dato al **DataTable**.

- **Update**. Examina todos los cambios en el **DataTable** y llama a las instrucciones INSERT, UPDATE o DELETE respectivas para cada uno de ellos con el fin de actualizar el origen de datos.

Como ejemplo, vamos a crear una base de datos para almacenar las notas de los alumnos matriculados en un determinado centro de unas determinadas asignaturas. La figura siguiente muestra la arquitectura de esta base de datos:



Después, vamos a construir una aplicación que muestre una interfaz que permita a un alumno acceder a la base de datos para conocer la nota obtenida en cualquiera de las asignaturas en las que está matriculado. La interfaz de la aplicación será similar a la siguiente:



A la vista de la interfaz anterior, el alumno seleccionará su nombre en la primera lista (este control muestra los nombres de la tabla *alumnos*) lo que hará que se muestren en la otra lista las asignaturas de las que está matriculado. Después, seleccionará la asignatura de la cual quiere conocer la nota y ésta le será mostrada en un control de texto. Una posible forma de proceder sería la siguiente:

- Crear la base de datos *bd_notasAlumnos*.

- Crear una solución. Por ejemplo, la solución *AccesoDesconectado* con un proyecto con el mismo nombre. Éste será el proyecto principal.

- Añadir a la solución un proyecto secundario de tipo *Biblioteca de clases* que proporcione un **DataSet** construido a partir de las tablas de la base de datos. Por ejemplo, el proyecto *BDNotasAlumnos*.

- Añadir al proyecto principal un proveedor de datos, *pdAlumnos*, que permita acceder al conjunto de datos obtenido a partir de las filas de la tabla *alumnos*.

- Añadir al proyecto principal un **DataGrid** y vincularlo con *pdAlumnos*. Tiene que mostrar el *id_alumno* y el *nombre* del alumno.

- Añadir al proyecto principal otro proveedor de datos, *pAsigsNotas*, que permita acceder al conjunto de datos obtenido a partir de la consulta siguiente:

```
SELECT id_alumno, id_asignatura, nota,
 (SELECT nombre FROM asignaturas
```

```
  WHERE (alums_asigs.id_asignatura = id_asignatura)) AS nombreAsig
FROM alums_asigs WHERE (id_alumno = @id_alumno)
```

Esta consulta obtiene de la tabla *alums_asigs* las columnas *id_alumno*, *id_asignatura* y *nota* para un alumno determinado (para el seleccionado en la primera lista) y además añade a este conjunto de datos la columna *nombreAsig* correspondiente al nombre de la asignatura *id_asignatura* obtenido de la tabla *asignaturas*.

- Añadir otro **DataGrid** y vincularlo con el proveedor *pAsigsNotas*. Cada vez que se seleccione un alumno en la primera lista, esta segunda lista tiene que mostrar el *id_asignatura* y el *nombreAsig* de las asignaturas de las que está matriculado el alumno seleccionado.

- Añadir un control de texto para mostrar la nota correspondiente a la asignatura seleccionada en la segunda lista.

Crear la base de datos

El primer paso es crear la base de datos Sql Server que denominaremos *bd_notasAlumnos*. Esta base estará formada por tres tablas: *alumnos*, *asignaturas* y *alums_asigs*. Las relaciones entre las tablas especificarán que un alumno puede estar matriculado de muchas asignaturas y que una asignatura puede ser elegida por muchos alumnos. La construcción la realizaremos mediante un *script*, que se proporciona en el CD correspondiente al libro, del cual mostramos a continuación la descripción de las tablas (véase también el apartado *Crear una base de datos* al principio de este capítulo):

```
CREATE TABLE alumnos
(
  id_alumno integer primary key,
  nombre varchar(50) not null,
)

CREATE TABLE asignaturas
(
  id_asignatura integer primary key,
  nombre varchar(60) not null
)

CREATE TABLE alums_asigs
(
  id_alumno integer not null,
  id_asignatura integer not null,
  nota real not null,
  primary key (id_alumno,id_asignatura)
)
```

Una vez ejecutado el *script*, la base de datos *bd_notasAlumnos* habrá quedado añadida al servidor Sql Server.

Crear un proyecto WPF

En el siguiente paso vamos a construir un nuevo proyecto WPF, denominado *AccesoDesconectado*. La ventana principal, *MainWindow*, presentará un **DataGrid** para mostrar la lista de alumnos, otro para mostrar la lista de asignaturas en las que está matriculado el alumno seleccionado y un **TextBlock** para mostrar la nota correspondiente a la asignatura seleccionada.

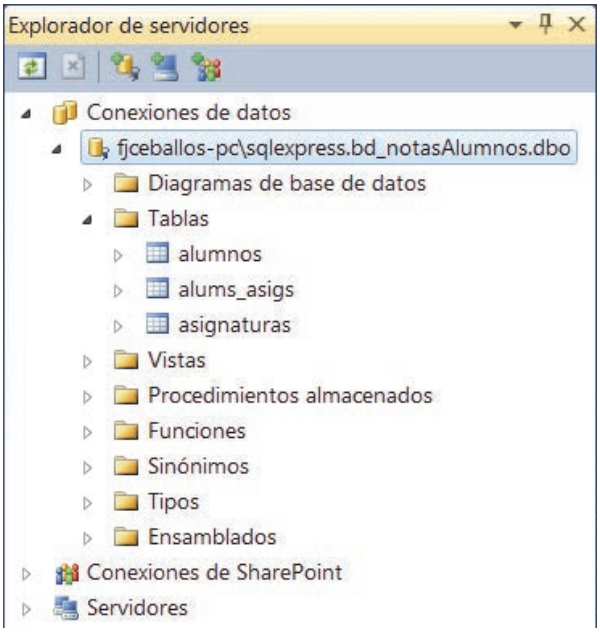
```
<Window x:Class="MainWindow"
        ...
        xmlns:local="clr-namespace:AccesoDesconectado"
        Title="Notas alumnos" Height="380" Width="412">
  <Window.Resources>
  </Window.Resources>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>
    <StackPanel>
      <TextBlock FontSize="16" Margin="4"
                 Text="Lista de alumnos:"/>
      <DataGrid Name="dgAlumnos" Margin="5" MaxHeight="95"
                AutoGenerateColumns="False">
        <DataGrid.Columns>
          <DataGridTextColumn Header="ID" MinWidth="100"/>
          <DataGridTextColumn Header="Nombre" MinWidth="260"/>
        </DataGrid.Columns>
      </DataGrid>
    </StackPanel>
    <StackPanel Name="spAsignaturas" Grid.Row="1">
      <TextBlock FontSize="16" Margin="4"
           Text="Lista de asignaturas del alumno seleccionado:"/>
      <DataGrid Name="dgAsignaturas" Margin="5" MaxHeight="95"
                AutoGenerateColumns="False">
        <DataGrid.Columns>
          <DataGridTextColumn Header="ID" MinWidth="100"/>
          <DataGridTextColumn Header="Nombre" MinWidth="260"/>
        </DataGrid.Columns>
      </DataGrid>
    </StackPanel>
    <StackPanel Grid.Row="2" >
      <TextBlock FontSize="16" Margin="4"
           Text="Nota del alumno y asignatura seleccionados:"/>
```

```
      <TextBlock Name="tbNota"
                 HorizontalAlignment="Center" FontWeight="Bold" />
    </StackPanel>
  </Grid>
</Window>
```

El código anterior será completado en los siguientes apartados con los recursos, con los contextos de datos y con los enlaces necesarios para que los controles muestren lo enunciado anteriormente.

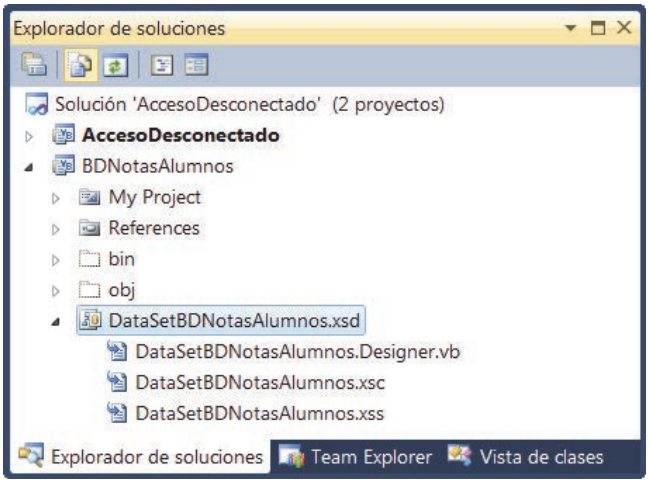
Conectarse a la base de datos Sql Server

Una vez construida la base de datos y creado un nuevo proyecto WPF, es posible establecer una conexión con la base de datos. Para ello, desde el menú *Ver*, abra el *Explorador de servidores* (*Explorador de bases de datos*) y cree una nueva conexión con la base de datos: clic con el botón secundario del ratón sobre el nodo *Conexiones de datos* y seleccione *Agregar conexión*. Esta operación le permitirá especificar la información (origen de datos, nombre del servidor, por ejemplo *localhost\sqlexpress*, y nombre de la base de datos) para establecer la conexión con el origen de datos Sql Server. Finalizada esta operación, la base de datos será añadida como un nodo hijo del nodo *Conexiones de datos* del *Explorador de servidores*. Para inspeccionar las tablas de la base de datos, expanda el nodo de la base de datos y a continuación el nodo *Tablas*. El resultado puede verlo en la figura siguiente:



Crear la capa de acceso a datos

Según lo estudiado hasta ahora, un enfoque adecuado en el desarrollo de aplicaciones es separar la capa de acceso a datos (DAL) de la capa de presentación. La DAL típicamente se implementa como un proyecto *Biblioteca de clases* según se explica en el apartado *Añadir un DataSet al proyecto* del apéndice A; entonces, siguiendo lo allí explicado, añada un nuevo proyecto *Biblioteca de clases* a la solución *AccesoDesconectado*, por ejemplo *BDNotasAlumnos*, y a continuación agregue a este proyecto un **DataSet** denominado *DataSetBDNotasAlumnos*.

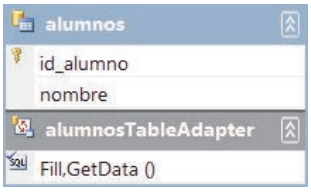


La operación de agregar al proyecto un **DataSet** hará que se muestre el *Diseñador de DataSet* que permite crear y editar visualmente conjuntos de datos con tipo, arrastrando elementos directamente desde la base de datos o desde la caja de herramientas de Visual Studio a la superficie de diseño o bien haciendo clic con el botón secundario del ratón sobre la superficie de diseño y agregando nuevos elementos. La DAL tiene ahora una nueva clase *DataSetBDNotasAlumnos* que permite construir objetos **DataSet**:

```
Dim ds As New DataSetBDNotasAlumnos()
```

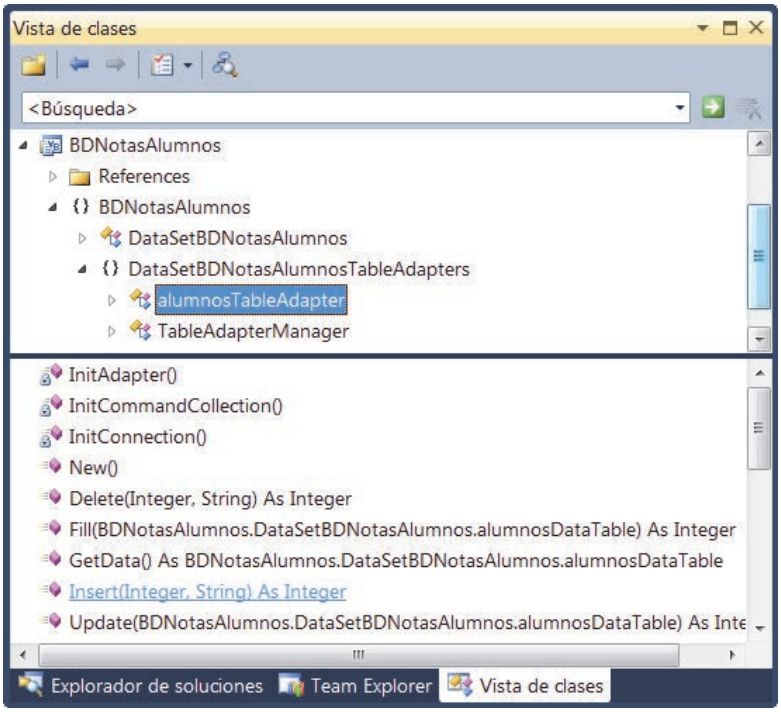
Una vez creada la biblioteca *BDNotasAlumnos.dll*, añada al nodo *References* del proyecto *AccesoDesconectado* una referencia a la misma.

Según lo expuesto anteriormente, para acceder a la base de datos y llenar con datos las tablas que añadamos al **DataSet** necesitamos adaptadores de datos. En primer lugar vamos a añadir un adaptador que nos permita obtener las filas de la tabla *alumnos*. Para ello, arrastre desde el *Explorador de servidores* a la superficie de diseño del **DataSet** la tabla *alumnos*.



La figura anterior indica que se ha añadido al **DataSet** una tabla *alumnos* (objeto **DataTable**) con la misma estructura que su homóloga en la base de datos y el adaptador *alumnosTableAdapter* que, entre otros, proporciona el método *Fill* para llenar la tabla *alumnos* tras realizar la consulta *SELECT id_alumno, nombre FROM dbo.alumnos* a la base de datos. Si hace clic con el botón secundario del ratón sobre la superficie correspondiente al adaptador, verá que puede configurarlo a medida (observará la consulta anterior) o añadir nuevas consultas.

Si ahora muestra la vista de clases del proyecto, podrá observar la capa de acceso a datos (DAL) formada por clases Visual Basic, como *alumnosRow*, *alumnosDataTable* y *alumnosTableAdapter*, que proporcionan la funcionalidad necesaria para realizar operaciones sobre los elementos de la base de datos que seleccionó y añadió al diseñador del **DataSet**.



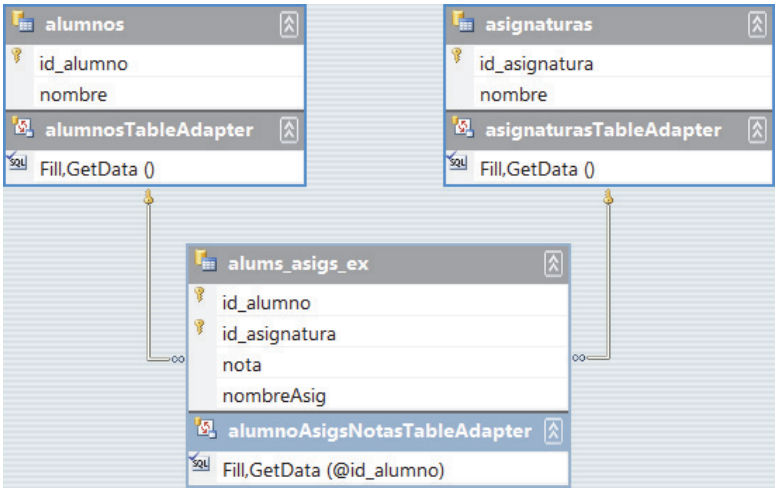
La clase *alumnosRow* se deriva de **DataRow** que representa una fila de datos en un **DataTable**, *alumnosDataTable* se deriva indirectamente de **DataTable** y se

utiliza para la generación de objetos **DataTable** y *alumnosTableAdapter* es el adaptador que permite obtener los datos y registrar las actualizaciones en la tabla *alumnos* de la base de datos. La figura anterior muestra este adaptador que, como vemos, proporciona un constructor para construir un objeto de este tipo y métodos como *Delete*, *Insert* o *Update* que nos permitirán realizar operaciones de borrar, insertar o actualizar una fila de la tabla *alumnos* del **DataSet**. Por ejemplo, la línea siguiente construye un adaptador de datos para la tabla *alumnos*:

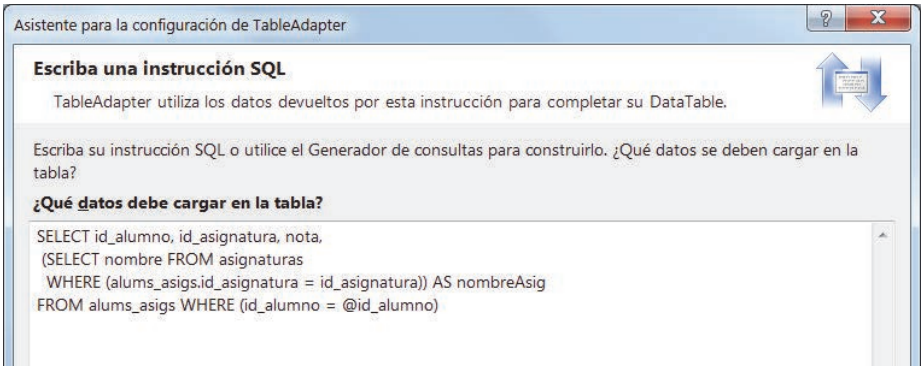
```
Dim ad As New alumnosTableAdapter()
```

La tabla *alumnos* del **DataSet** será quien provea de datos a la primera lista, control *dgAlumos*, de la interfaz gráfica.

Análogamente, para proveer de datos a la segunda lista, control *dgAsignaturas*, y al control de texto *tbNota* de la interfaz gráfica, vamos a añadir otras dos tablas al **DataSet** con sus adaptadores, según muestra la figura siguiente:



Para añadir la tabla *asignaturas* y su adaptador, arrastre desde el *Explorador de servidores* a la superficie de diseño del **DataSet** la tabla *asignaturas* y para añadir la tabla *alums_asigs_ex* y su adaptador *alumnoAsigsNotasTableAdapter*, haga clic con el botón secundario del ratón sobre la superficie de diseño y ejecute *Agregar* > *TableAdapter*, elija la conexión de datos que ya tiene establecida, elija el modo de acceso del adaptador a la base de datos (por ejemplo, usar una sentencia SQL), escriba la sentencia SQL indicada anteriormente, permita que se generen también las sentencias SQL para las operaciones de borrar, insertar y actualizar, y esto es todo. El resultado será el conjunto de asignaturas y las notas correspondientes al alumno *id_alumno*.



El método *Fill* toma como parámetro un **DataTable** o un **DataSet** y ejecuta la sentencia SQL SELECT (o procedimiento almacenado) para llenar la tabla correspondiente con los datos obtenidos. El método *GetData* devuelve un **DataTable** con las filas obtenidas por la sentencia SQL SELECT. Y si eligió crear métodos para enviar actualizaciones directamente a la base de datos, se crearon también los métodos *Delete, Insert* y *Update*.

Los objetos que podemos generar a partir de las clases de la DAL serán utilizados para acceder a las filas de las tablas que seleccionemos. Por ejemplo, para acceder a las filas de la tabla *alumnos* podemos escribir el código siguiente:

```
Dim adAlumnos As New alumnosTableAdapter()
Dim ds As New DataSetBDNotasAlumnos()
adAlumnos.Fill(ds.alumnos)
For Each fila As DataSetBDNotasAlumnos.alumnosRow In ds.alumnos
  System.Console.WriteLine("Alumno: " + fila.nombre)
Next
```

O bien:

```
Dim adAlumnos As New alumnosTableAdapter()
Dim tablaAlumnos As DataSetBDNotasAlumnos.alumnosDataTable
tablaAlumnos = adAlumnos.GetData()
For Each fila As DataSetBDNotasAlumnos.alumnosRow In tablaAlumnos
   System.Console.WriteLine("Alumno: " + fila.nombre)
Next
```

Capa de lógica de negocio

El siguiente paso es hacer que los controles de la interfaz gráfica muestren los datos correspondientes. Para ello vamos a escribir un poco de lógica de negocio.

El control **DataGrid** *dgAlumno* enlazará con los datos de un origen de datos referenciado o heredado por su contexto de datos. Ese origen de datos bien podría ser un **DataTable** o un **DataView**. Nos inclinamos por este último por las facilidades que da para realizar operaciones de ordenación, filtrado, búsqueda, edición y navegación sobre la vista vinculada con el **DataTable** obtenido. Esta vista, para el caso de la tabla *alumnos* del **DataSet**, será devuelta por el método *ObtenerAlumnos* de la clase *CProveedorDatosAlumno* que escribimos a continuación (añada esta clase, por ejemplo, a una carpeta *BLL* de su proyecto):

```
Imports BDNotasAlumnos
Imports BDNotasAlumnos.DataSetBDNotasAlumnosTableAdapters
Imports System.Data

Class CProveedorDatosAlumno
  Private ds As DataSetBDNotasAlumnos
  Private ad As alumnosTableAdapter

  Public Sub New()
    ds = New DataSetBDNotasAlumnos()
    ad = New alumnosTableAdapter()
    ad.Fill(ds.alumnos)
  End Sub

  Public Function ObtenerAlumnos() As DataView
    Return ds.alumnos.DefaultView
  End Function
End Class
```

Análogamente, el control **DataGrid** *dgAsignaturas* enlazará con los datos de un origen de datos proporcionado por el método *ObtenerAsignaturasNotas* de la clase *CProveedorAsignaturasNotas* que escribimos a continuación (añada esta clase a la carpeta *BLL* de su proyecto):

```
Imports BDNotasAlumnos
Imports BDNotasAlumnos.DataSetBDNotasAlumnosTableAdapters
Imports System.Data

Class CProveedorAsignaturasNotas
  Private ds As DataSetBDNotasAlumnos
  Private ad As alumnoAsigsNotasTableAdapter

  Public Sub New()
    ds = New DataSetBDNotasAlumnos()
    ad = New alumnoAsigsNotasTableAdapter()
  End Sub

  Public Function ObtenerAsignaturasNotas(IdAlumno As Integer) As DataView
    ad.Fill(ds.alums_asigs_ex, IdAlumno)
```

```
    Return ds.alums_asigs_ex.DefaultView
  End Function
End Class
```

Lógica de interacción con la capa de presentación

El origen de datos para que el **DataGrid** *dgAlumnos* muestre la lista de alumnos tiene que ser un objeto de la clase *CProveedorDatosAlumno*. Este objeto podemos crearlo en el código subyacente (fichero *.vb*) o en el código XAML de la ventana (fichero *.xaml*). La ventaja de crearlo en el código XAML es que durante el diseño dispondremos del soporte que proporciona el control **DataGrid** a través de la orden *DataGrid* del menú contextual que se muestra al hacer clic con el botón secundario del ratón sobre dicho control, soporte que no tenemos en el otro caso.

El objeto *CProveedorDatosAlumno* lo podemos proporcionar por medio de un proveedor de datos, según explicamos en el capítulo *Enlaces de datos en WPF*, así:

```
<Window.Resources>
  <ObjectDataProvider x:Key="pdAlumnos"
    ObjectType="{x:Type local:CProveedorDatosAlumno}"
    MethodName="ObtenerAlumnos">
  </ObjectDataProvider>
</Window.Resources>
```

Una vez construido el recurso proveedor de datos, lo asignamos al contexto de datos del contenedor del **DataGrid**, enlazamos la propiedad **ItemSource** del **DataGrid** con ese recurso y enlazamos sus columnas con las columnas correspondientes del origen de datos:

```
<StackPanel DataContext="{StaticResource pdAlumnos}">
  <TextBlock FontSize="16" Margin="4" Text="Lista de alumnos:"/>
  <DataGrid Name="dgAlumnos" Margin="5" MaxHeight="95"
            AutoGenerateColumns="False"
            ItemsSource="{Binding}"
            SelectionChanged="dgAlumnos_SelectionChanged">
    <DataGrid.Columns>
      <DataGridTextColumn Header="ID"
        Binding="{Binding Path=id_alumno}" MinWidth="100"/>
      <DataGridTextColumn Header="Nombre"
        Binding="{Binding Path=nombre}" MinWidth="260"/>
    </DataGrid.Columns>
  </DataGrid>
</StackPanel>
```

Si prueba ahora la aplicación, observará cómo este **DataGrid** muestra la lista de alumnos.

El siguiente paso es hacer un proceso análogo con el **DataGrid** *dgAsignaturas*. En este caso, el conjunto de datos generado se corresponde con la lista de asignaturas, además de otras columnas, del alumno seleccionado en el **DataGrid** *dgAlumnos*. Recuerde que la consulta para obtener este conjunto de datos necesitaba del parámetro *@id_alumno*: identificador del alumno seleccionado. Para obtener este identificador interceptaremos el evento **SelectionChanged** de *dgAlumnos*. El método que responda a este evento, una vez obtenida la fila seleccionada en *dgAlumnos*, realizará la consulta invocando al método *ObtenerAsignaturasNotas* de la clase *CProveedorAsignaturasNotas* y asignará el origen de datos obtenido al contexto de datos del contenedor del **DataGrid** *dgAsignaturas*:

```
Private Sub dgAlumnos_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  If e.AddedItems.Count = 0 Then Return
  Dim drv As DataRowView = TryCast(e.AddedItems(0), DataRowView)
  If drv Is Nothing Then Return
  Dim fila As DataSetBDNotasAlumnos.alumnosRow =
    TryCast(drv.Row, DataSetBDNotasAlumnos.alumnosRow)
  If fila.RowState = DataRowState.Detached Then Return
  vistaAsignaturas =
    pAsigsNotas.ObtenerAsignaturasNotas(fila.id_alumno)
  spAsignaturas.DataContext = vistaAsignaturas
End Sub
```

La propiedad **RowState** define el estado actual de la fila con respecto a su colección **DataRowCollection**. Este estado es un valor, entre los siguientes, de la enumeración **DataRowState**:

- **Detached**. La fila ha sido creada, pero aún no ha sido agregada a la colección, o bien se ha quitado de la colección.
- **Unchanged**. La fila no ha cambiado.
- **Added**. La fila se ha agregado a la colección **DataRowCollection**.
- **Deleted**. La fila se ha eliminado mediante el método **Delete** del **DataRow**.
- **Modified**. La fila se ha modificado.

El código anterior requiere definir como atributos de la clase *MainWindow* las referencias *pAsigsNotas* y *vistaAsignaturas* a los objetos de sus clases respectivas:

```
Imports System.Data
Imports BDNotasAlumnos

Class MainWindow
  Private pAsigsNotas As New CProveedorAsignaturasNotas()
```

```
  Private vistaAsignaturas As DataView

  , ...
End Class
```

Una vez construido el origen de datos y asignado al contexto de datos del contenedor del **DataGrid**, enlazamos la propiedad **ItemSource** de éste con ese origen y enlazamos sus columnas con las columnas correspondientes del mismo:

```
<StackPanel Name="spAsignaturas" Grid.Row="1">
  <TextBlock FontSize="16" Margin="4"
             Text="Lista de asignaturas del alumno seleccionado:"/>
  <DataGrid Name="dgAsignaturas" Margin="5" MaxHeight="95"
            AutoGenerateColumns="False"
            ItemsSource="{Binding}">
    <DataGrid.Columns>
      <DataGridTextColumn Header="ID"
        Binding="{Binding Path=id_asignatura}" MinWidth="100"/>
      <DataGridTextColumn Header="Nombre"
        Binding="{Binding Path=nombreAsig}" MinWidth="260"/>
    </DataGrid.Columns>
  </DataGrid>
</StackPanel>
```

Finalmente, cuando el usuario seleccione una asignatura del **DataGrid** *dgAsignaturas*, un control de texto mostrará la nota correspondiente. Recuerde que cada fila de este control tiene cuatro columnas, entre ellas el nombre de la asignatura y la nota obtenida. Según esto, el origen de datos para el control de texto será la fila seleccionada en el **DataGrid** *dgAsignaturas*:

```
<StackPanel Grid.Row="2"
  DataContext="{Binding SelectedItem, ElementName=dgAsignaturas}">
  <TextBlock FontSize="16" Margin="4"
    Text="Nota del alumno y asignatura seleccionados:"/>
  <TextBlock Name="tbNota" Text="{Binding Path=nota}"
             HorizontalAlignment="Center" FontWeight="Bold" />
</StackPanel>
```

Ejecute ahora la aplicación y pruebe los resultados.

Actualizaciones

Cuando un usuario edita una tabla mostrada por un **DataGrid**, el objeto **DataTable** origen de los datos correspondiente es también actualizado. Sin embargo, debido a que estamos trabajando en modo desconectado, estas actualizaciones no se registran de forma automática en la base de datos, sino que es responsabilidad del desarrollador decidir cuándo se registrarán estos cambios dependiendo de los re-

querimientos de la aplicación. Por ejemplo, los cambios podrían ser registrados al cerrar la aplicación, al hacer clic en un botón "Guardar cambios" o al finalizar la edición de la fila que se está modificando. De cualquiera de las formas, la actualización de la base de datos pasa por invocar al método **Update** del adaptador correspondiente.

Cuando se finaliza una operación de añadir, borrar o modificar una fila, se generan, dependiendo de la operación, los eventos **RowChanged** o **RowDeleted** del **DataTable**. Por lo tanto, podemos manipular estos eventos para registrar los cambios en la base de datos cuando el usuario finalice los cambios en una fila. Aplicando esta idea a nuestra solución, podemos modificar el constructor *CProveedorDatosAlumno*, como se indica a continuación, para que el manejador de los eventos descritos para el **DataTable** *alumnos* ejecute el método **Update** del adaptador y actualice la base de datos con los cambios de dicho objeto cada vez que el usuario finalice la edición de una fila o borre una fila:

```
Public Sub New()
  ds = New DataSetBDNotasAlumnos()
  ad = New alumnosTableAdapter()
  ad.Fill(ds.alumnos)

  AddHandler ds.alumnos.alumnosRowChanged,
    AddressOf alumnos_FilaModificada
  AddHandler ds.alumnos.alumnosRowDeleted,
    AddressOf alumnos_FilaModificada
End Sub

Private Sub alumnos_FilaModificada(sender As Object,
e As DataSetBDNotasAlumnos.alumnosRowChangeEvent)
  Try
    ad.Update(ds.alumnos)
  Catch ex As System.Data.SqlClient.SqlException
    System.Windows.MessageBox.Show(ex.Message)
  End Try
End Sub
```

Como ejercicio, puede añadir una nueva funcionalidad a la solución para que permita matricular nuevos alumnos o dar de baja alumnos, teniendo en cuenta las relaciones entre las tablas. Por ejemplo, el siguiente código indica cómo se pueden añadir las notas de las asignaturas en las que se ha matriculado el alumno añadido a la tabla *alumnos*:

```
Private Sub alumnos_FilaModificada(sender As Object,
e As DataSetBDNotasAlumnos.alumnosRowChangeEvent)
  Try
    ad.Update(ds.alumnos)
    If e.Action = DataRowAction.Add Then
```

```
      AñadirNotas(e)
    End If
  Catch ex As System.Data.SqlClient.SqlException
    System.Windows.MessageBox.Show(ex.Message)
  End Try
End Sub

Private Sub AñadirNotas(e As DataSetBDNotasAlumnos.alumnosRowChangeEvent)
  Dim filaAsigs As DataSetBDNotasAlumnos.asignaturasRow =
    TryCast(ds.asignaturas.NewRow(),
            DataSetBDNotasAlumnos.asignaturasRow)
  filaAsigs.id_asignatura = 33693
  Dim nota As Single = 5.5F
  ds.alums_asigs_ex.Addalums_asigs_exRow(e.Row, filaAsigs, nota, "")
  Dim ad2 As New alumnoAsigsNotasTableAdapter()
  ad2.Update(ds.alums_asigs_ex)
End Sub
```

El método *AñadirNotas*, utilizando la funcionalidad proporcionada por la DAL, añade una fila a la tabla *alums_asigs_ex* del conjunto de datos *ds*. Dicha fila se corresponde con la nota de la asignatura 33693 del alumno cuya fila se supone que el usuario acaba de añadir al **DataGrid** *dgAlumnos*. Evidentemente el identificador de la asignatura o asignaturas y sus notas los obtendríamos a través de una nueva ventana con una lista de asignaturas, para seleccionar la asignatura, y con un control de edición para introducir la nota.

Clase DataView

Un objeto **DataView** define una vista personalizada de un **DataTable**. En otras palabras, se trata de una representación de los datos del objeto **DataTable** que permite ordenación, filtrado, búsqueda, edición y navegación. Para configurar estas características el objeto **DataView** proporciona propiedades como **Sort** y **RowFilter**. Estas propiedades permiten clasificar y filtrar los datos que mostrará la vista, pero sin afectar al contenido del **DataTable**.

Cada **DataTable** tiene asociado por defecto un **DataView**, aunque es posible crear múltiples **DataView** para mostrar diferentes vistas de la misma tabla. Según hemos podido ver anteriormente, esta vista predeterminada se obtiene a través de la propiedad **DefaultView** del **DataTable**.

Por ejemplo, para mostrar las filas del **DataGrid** *dgAsignaturas* ordenadas por el nombre de la asignatura simplemente tendríamos que modificar el método *dgAlumnos_SelectionChanged* de nuestra solución como se indica a continuación:

```
vistaAsignaturas =
  pAsigsNotas.ObtenerAsignaturasNotas(fila.id_alumno)
```

```
vistaAsignaturas.Sort = "nombreAsig"
spAsignaturas.DataContext = vistaAsignaturas
```

Mediante la propiedad **Sort** puede especificar criterios simples o múltiples de ordenación de columnas, separados por comas, e incluir parámetros ASC (ascendente; valor por defecto) y DESC (descendente). Por ejemplo:

```
vista.Sort = "columna1, columna2 DESC"
```

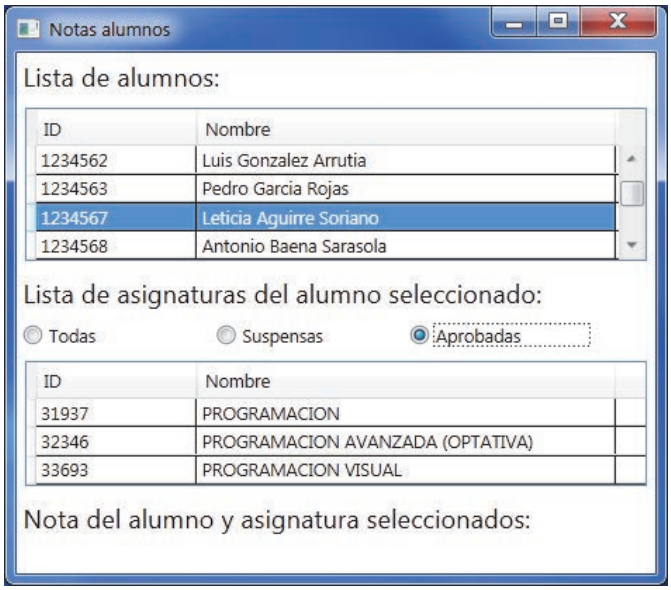
También podemos utilizar el **DataView** para aplicar filtros personalizados para que sólo se muestren ciertas filas. Por ejemplo, el siguiente código, además de ordenar las filas por el nombre, sólo muestra las filas cuya nota es menor que 5:

```
vistaAsignaturas =
  pAsigsNotas.ObtenerAsignaturasNotas(fila.id_alumno)
vistaAsignaturas.Sort = "nombreAsig"
vistaAsignaturas.RowFilter = "nota < '5'"
spAsignaturas.DataContext = vistaAsignaturas
```

Para formar un valor **RowFilter**, especifique el nombre de la columna seguido de un operador y del valor a filtrar entre comillas simples, según muestra el ejemplo anterior. Se pueden utilizar operadores de relación (<, <=, >, >=, <>, =), lógicos (AND, OR, NOT) y otros como LIKE '*patrón*', IS NULL, etc.

Como ejemplo, vamos a añadir tres botones de opción a la ventana de nuestra aplicación para permitir al usuario mostrar: todas las asignaturas del alumno seleccionado, las asignaturas que tiene suspensas o las asignaturas que tiene aprobadas, según muestran el código y la figura siguientes:

```
<StackPanel Name="spAsignaturas" Grid.Row="1">
  <TextBlock FontSize="16" Margin="4"
    Text="Lista de asignaturas del alumno seleccionado:"/>
  <WrapPanel ItemWidth="120">
    <RadioButton Name="boTodas" Content="Todas" Margin="4"
      IsChecked="True" Checked="boTodas_Checked" />
    <RadioButton Name="boSuspensas" Content="Suspensas"  Margin="4"
      Checked="boSuspensas_Checked" />
    <RadioButton Name="boAprobadas" Content="Aprobadas"  Margin="4"
      Checked="boAprobadas_Checked" />
</WrapPanel>
```



Por omisión, se visualizan todas las asignaturas del alumno seleccionado. Para mostrar sólo las suspensas o las aprobadas, el usuario tendrá que seleccionar la opción correspondiente. La respuesta a estas acciones será simplemente aplicar el filtro correspondiente a la vista y asignar ésta al contexto de datos. Por lo tanto, añada los controladores de estos botones de opción y edítelos como se indica a continuación:

```
Private Sub boTodas_Checked(sender As Object, e As RoutedEventArgs)
  If vistaAsignaturas Is Nothing Then Return
  vistaAsignaturas.RowFilter = ""
  spAsignaturas.DataContext = vistaAsignaturas
End Sub

Private Sub boSuspensas_Checked(sender As Object, e As RoutedEventArgs)
  If vistaAsignaturas Is Nothing Then Return
  vistaAsignaturas.RowFilter = "nota < '5'"
  spAsignaturas.DataContext = vistaAsignaturas
End Sub

Private Sub boAprobadas_Checked(sender As Object, e As RoutedEventArgs)
  If vistaAsignaturas Is Nothing Then Return
  vistaAsignaturas.RowFilter = "nota >= '5'"
  spAsignaturas.DataContext = vistaAsignaturas
End Sub
```

RESUMEN

Este capítulo empieza haciendo un breve resumen de SQL, para después continuar indicando los pasos a seguir para crear una base de datos, origen de datos de las aplicaciones desarrolladas en este capítulo.

¿Qué mecanismos puede utilizar una aplicación para interaccionar con una base de datos? Son muchas las aplicaciones, distribuidas o no, que trabajan sobre bases de datos. Por esta razón, continuamos este capítulo exponiendo ADO.NET: la tecnología de acceso a datos de Microsoft. ADO.NET permite dos formas de trabajar con los datos de una base de datos: conectado a la base de datos y desconectado. El objeto **DataSet** nos ofrece la posibilidad de trabajar con los datos procedentes de una base de datos estando desconectada de dicha base y cuando sea necesario trabajar estando la aplicación conectada a la base de datos, ADO.NET nos ofrece el objeto **DataReader**.

Continuamos con el desarrollo de aplicaciones utilizando el modo conectado a bases de datos. Estudiamos el problema que se presenta al utilizar cadenas de caracteres para crear órdenes SQL, ya que favorecen los ataques de inyección de código SQL, y presentamos alternativas que impidan esos ataques. Después, para mantener la integridad de todos los datos de la base, introdujimos el concepto de transacción y expusimos cómo controlar cuándo se inicia y cuándo se finaliza una transacción, bien mediante sentencias SQL o bien mediante funciones de la API de bases de datos.

A continuación, pensando en el desarrollo de aplicaciones profesionales, se expuso un modelo estándar de desarrollo basado en capas, lo cual exige encapsular el código de acceso a la base de datos en clases que permitan la construcción de componentes de acceso a datos que posibiliten el desacoplo de la aplicación de la base de datos. También se estudió cómo desacoplar la interfaz gráfica del resto de la aplicación y la validación de los datos introducidos por el usuario.

Finalmente abordamos el acceso desconectado a la base de datos mediante el uso del objeto **DataSet** y el estudio de la presentación de los datos empleando vistas, objetos de la clase **DataView**.

CAPÍTULO 7

© F.J.Ceballos/RA-MA

LINQ

LINQ (***L**anguage **IN**tegrated **Q**uery*) fue inicialmente soportado por .NET Framework 3.0 con la finalidad de ofrecer la posibilidad de expresar las operaciones de consulta en el propio lenguaje (Visual Basic, por ejemplo) y no como literales de cadena pertenecientes a otro lenguaje incrustados en el código de aplicación o como procedimientos almacenados. Finalmente, es con la versión 3.5 de .NET Framework, espacios de nombres *System.Linq* y *System.Data.Linq*, cuando LINQ queda totalmente integrado en este marco de trabajo, junto con otras bibliotecas como WPF, WCF, WF o ASP.NET AJAX, haciendo realidad la implementación de aplicaciones que contengan única y exclusivamente código .NET.

RECURSOS DEL LENGUAJE COMPATIBLES CON LINQ

LINQ es una combinación de extensiones al lenguaje y bibliotecas de código administrado que permite expresar de manera uniforme consultas sobre colecciones de datos de diversa procedencia utilizando recursos del propio lenguaje de programación; por ejemplo, sobre objetos en memoria, sobre bases de datos relacionales o sobre documentos XML, entre otros. Los elementos básicos sobre los que se construyeron estas nuevas extensiones son los siguientes:

1. Declaración implícita de variables locales.
2. Matrices de tipos definidos de forma implícita.
3. Tipos anónimos.
4. Propiedades auto-implementadas.
5. Iniciadores de objetos y colecciones.
6. Métodos extensores.
7. Expresiones *lambda*.
8. Operadores de consulta.
9. Árboles de expresiones lambda.

Estas características son la base para formar las *expresiones de consulta*, las cuales constituyen el principal reflejo en el lenguaje de la tecnología LINQ.

Declaración implícita de variables locales

Utilizando el identificador **var** es posible declarar una variable iniciada con un valor sin especificar su tipo. En este caso, el de la variable será tomado del tipo del valor con el que se inicia la misma. Por ejemplo:

```
Dim e = 0 ' equivale a Dim e As Integer = 0
Console.WriteLine(e.GetType().FullName) ' imprime: System.Int32
```

Tal práctica de esta característica va en contra de la claridad del código fuente, pero es verdaderamente potente cuando se utiliza en combinación con otras, como, por ejemplo, con los tipos anónimos.

Matrices de tipos definidos de forma implícita

Apoyándonos en lo expuesto en el apartado anterior, también es posible iniciar una matriz sin especificar explícitamente el tipo de los elementos, el cual será obtenido por Visual Basic a partir del tipo de los valores proporcionados. Por ejemplo:

```
Dim m() = {1, 2, 3} ' equivale a Dim m() As Object = {1, 2, 3}
For Each a In m
  Console.WriteLine(a) ' imprime: 1 2 3
 Next
Console.WriteLine(m.GetType().FullName) ' imprime: System.Object[]
```

Tipos anónimos

Es posible generar un tipo nuevo de datos a partir de expresiones de iniciación. Por ejemplo:

```
Dim persona = New With {.Nombre = "Isabella", .Edad = 1}
Console.WriteLine(persona.Nombre)
Console.WriteLine(persona.GetType().FullName)
```

La primera sentencia del ejemplo anterior genera un tipo anónimo con los atributos especificados en su iniciación. Un tipo anónimo es una manera cómoda de agrupar temporalmente un conjunto de propiedades en un resultado de una consulta sin tener que definir un tipo con nombre independiente.

Propiedades auto-implementadas

Una propiedad auto-implementada hace que la declaración de la misma sea más breve cuando no se requiere ninguna lógica adicional en los descriptores de acceso de la propiedad. Al declarar una propiedad como se muestra en el ejemplo siguiente, el compilador crea un atributo de respaldo privado y anónimo que no está accesible directamente. Por ejemplo:

```
Public Property Nombre() As String ' a partir de VB 10
```

La declaración de la propiedad *Nombre* anterior es equivalente a esta otra:

```
Private _nombre As String
Public Property Nombre() As String
  Get
    Return _nombre
  End Get
  Set(value As String)
    _nombre = value
  End Set
End Property
```

El nivel de acceso en los descriptores puede ser diferente:

```
Private Set(value As String)
```

Iniciadores de objetos y colecciones

Los iniciadores de objeto y de colección permiten iniciar los objetos sin llamar explícitamente al constructor correspondiente del objeto. Los iniciadores suelen utilizarse en expresiones de consulta. Por ejemplo, supongamos que hemos definido la clase *CPersona* así:

```
Public NotInheritable Class CPersona
  Public Property Nombre() As String
  Public Property FechaNac () As DateTime
End Class
```

Partiendo de esta declaración es posible iniciar un objeto empleando la siguiente construcción:

```
Dim persona = New CPersona With {.Nombre = "Isabella", _
                   .FechaNac = New DateTime(2011, 11, 7)}
```

También podemos iniciar una colección, iniciando cada uno de sus elementos de la forma expuesta anteriormente:

```
Dim listPersona = New List(Of CPersona) From { _
  New CPersona With { _
    .Nombre = "Isabella", .FechaNac = New DateTime(2011, 11, 7)}, _
  New CPersona With { _
    .Nombre = "Manuel", .FechaNac = New DateTime(1991, 7, 26)} _
} ' a partir de VB 2010
```

Métodos extensores

Los métodos extensores permiten extender o ampliar la funcionalidad de una clase sin recurrir a la herencia (técnica que en ocasiones es imposible, por ejemplo, si la clase ha sido calificada **NotInheritable**) ni a la delegación (que requiere una implementación más laboriosa). Los métodos de extensión se pueden declarar únicamente dentro de los módulos. Si el módulo en el que se define el método de extensión no es el mismo que el módulo en el que se llama, habrá que importar el módulo que lo contiene. Como ejercicio, supongamos que deseamos ampliar la clase *CPersona* implementada en el párrafo anterior, por ejemplo, para conocer la edad que tiene una determinada persona. La solución pasaría por escribir un procedimiento (**Function** o **Sub**), por ejemplo *Edad*, marcado con el atributo **<Extension()>** del espacio de nombres **System.Runtime.CompilerServices** con un primer parámetro del tipo de la clase a extender (en nuestro caso de tipo *CPersona*):

```
<Extension()> _
Function Edad(persona As CPersona) As Integer
  Dim _edad As Integer = DateTime.Today.Year - _
                         persona.FechaNac.Year
  If (DateTime.Today.Month > persona.FechaNac.Month Or _
      DateTime.Today.Month = persona.FechaNac.Month And _
      DateTime.Today.Day >= persona.FechaNac.Day) Then
    Return _edad
  Else
    Return _edad - 1
  End If
End Function
```

Cuando se ejecuta el método, el primer parámetro se enlaza al objeto que invoca al método.

Si ahora modificamos el ejemplo que hicimos en el apartado anterior para que un elemento de la lista llame al método extensor, podremos observar que su comportamiento es como si fuera un método nativo de *CPersona*.

```
Dim listPersona = New List(Of CPersona)()
listPersona.AddRange(New CPersona() { _
  New CPersona With { _
    .Nombre = "Isabella", .FechaNac = New DateTime(2011, 11, 7)}, _
  New CPersona With { _
    .Nombre = "Manuel", .FechaNac = New DateTime(1991, 7, 26)} _
})

For Each persona In listPersona
  Console.WriteLine("{0}, {1}", persona.Nombre, persona.Edad())
Next
```

Los métodos extensores en la resolución de llamadas por parte del compilador tienen menor resolución que los métodos nativos; esto es, si la clase *CPersona* tuviera un método nativo *Edad*, se utilizaría éste.

Expresiones lambda

Las expresiones *lambda*, un recurso tradicional en los lenguajes de programación funcional, pueden transformarse en tipos de delegados para la generación y posterior ejecución de código. Esto es, pueden reemplazar de una manera más concisa a los métodos anónimos: bloques de código que pueden colocarse *inline* en aquellos sitios donde el compilador espera encontrarse un delegado. La sintaxis de estas expresiones es la siguiente:

```
Function(lista de parámetros) expresión|bloque de sentencias
```

La lista de parámetros, separados por comas, irá entre paréntesis precedidos por la palabra reservada **Function**. A continuación se escribe la expresión o bloque de sentencias a ejecutar, tomando como argumentos esos parámetros. Ejemplos:

```
Dim expLambda1 = Function(a) a * 2 ' expresión; tipo implícito
Dim expLambda2 = Function(a As Integer) a * 2 ' tipo explícito
Dim expLambda3 = Function(a, b) a + b ' varios parámetros
Dim expLambda4 = Function() Math.Sqrt(2) ' sin parámetros
Dim expLambda5 = Function(s As String, x As Integer) s.Length > x
    ' varios parámetros con tipo

Console.WriteLine(expLambda1(7))
Console.WriteLine(expLambda2(7))
Console.WriteLine(expLambda3(5, 8))
Console.WriteLine(expLambda4())
Console.WriteLine(expLambda5("Hola", 4))
```

Cuando el compilador no pueda deducir los tipos de entrada, habrá que especificarlos explícitamente, como ocurre en el último ejemplo.

A diferencia de una función estándar, una función lambda:

- No tiene nombre.
- No utilizan una cláusula **As** para designar el tipo de valor retornado.
- El cuerpo de la función debe ser una expresión o un conjunto de sentencias y puede estar formado por una llamada a un procedimiento **Function** o **Sub**.

Los ejemplos mostrados a continuación aclaran lo expuesto:

```
Dim exprLambda1 = Function(a)
                    Return a * 2
                  End Function
Dim exprLambda2 = Function(a As Integer)
                    If (a > 2) Then a = 10
                    Return a + 2
                  End Function
Dim exprLambda3 = Sub() Console.WriteLine("hola")

Console.WriteLine(exprLambda1(1))
Console.WriteLine(exprLambda2(2))
exprLambda3()
```

Como ejemplo, vamos a añadir a la clase *CPersona* un método *NacidoAño* que muestre las personas de una lista que hayan nacido en un año determinado. El primer argumento de este método será una referencia a la lista de personas y el segundo, un delegado que ejecutará la expresión condicional necesaria.

```
' Delegado
Public Delegate Function ExprCondicional(pers As CPersona) As Boolean

Public NotInheritable Class CPersona
  ' ...
  Public Shared Sub NacidosAño(lista As List(Of CPersona), _
                               condición As ExprCondicional)
    For Each p In lista
      If (condición(p)) Then
        Console.WriteLine(p.Nombre)
      End If
    Next
  End Sub
End Class

Module Test
  Public Function Comparar(pers As CPersona) As Boolean
    Return pers.FechaNac.Year = 1991
```

```
  End Function

  Sub Main()
    Dim listPersona = New List(Of CPersona)()
    listPersona.AddRange(New CPersona() { _
      New CPersona With { _
       .Nombre = "Isabella", .FechaNac = New DateTime(2011, 11, 7)}, _
      New CPersona With { _
       .Nombre = "Manuel", .FechaNac = New DateTime(1991, 7, 26)} _
     })
    Dim delegado As ExprCondicional = AddressOf Comparar
    CPersona.NacidosAño(listPersona, delegado)
  End Sub
End Module
```

La llamada a *NacidosAño* desde *Test* incluye en su segundo argumento un delegado. Reemplacemos este delegado por un método anónimo. En Visual Basic un método anónimo se construye a partir de una expresión *lambda*:

```
Dim métodoAnónimo As ExprCondicional = _
  Function(p As CPersona) p.FechaNac.Year = 1991
CPersona.NacidosAño(listPersona, métodoAnónimo)
```

La llamada a *NacidosAño* desde *Test* incluye en su segundo argumento un método anónimo. Reemplacemos este método por la expresión *lambda*:

```
CPersona.NacidosAño(listPersona, _
  Function(p As CPersona) p.FechaNac.Year = 1991)
```

o bien,

```
CPersona.NacidosAño(listPersona, _
  Function(p) p.FechaNac.Year = 1991)
```

El tipo de *p* se deduce de la definición del delegado.

El delegado Func(Of *T*, *TResu*)

El delegado **Func** utiliza parámetros de tipo:

```
Public Delegate Function Func(Of T,..., TResult)(arg As T,...) As TResult
```

El valor devuelto siempre se corresponde con el parámetro especificado en el último lugar; el resto son parámetros de entrada. Por ejemplo, el siguiente código define un delegado **Func** que, cuando se invoca, devuelve verdadero o falso para indicar si el parámetro de entrada es un número par:

```
Dim esPar As Func(Of Integer, Boolean) = Function(n) n Mod 2 = 0
Dim resultado As Boolean = esPar(11)
```

Según lo expuesto, en el ejemplo del apartado anterior podemos prescindir del delegado *ExprCondicional* y modificar el método *NacidosAño* para que utilice un delegado **Func**:

```
Public Shared Sub NacidosAño(lista As List(Of CPersona), _
  condición As Func(Of CPersona, Boolean))
  For Each p As CPersona In lista
    If condición(p) Then
      Console.WriteLine(p.Nombre)
    End If
  Next
End Sub
```

La llamada a este método sería idéntica a la del ejemplo anterior:

```
CPersona.NacidosAño(listPersona, _
  Function(p) p.FechaNac.Year = 1991)
```

También, podríamos prescindir del método *NacidosAño* y utilizar un delegado **Func** con dos parámetros de entrada (el tercero es el tipo del valor devuelto):

```
Dim esNacidoEnAño As Func(Of CPersona, Integer, Boolean) = _
  Function(p, año) p.FechaNac.Year = año
For Each p As CPersona In listPersona
  If esNacidoEnAño(p, 1991) Then
    Console.WriteLine(p.Nombre)
  End If
Next
```

Operadores de consulta

Los operadores de consulta son los métodos que forman el modelo de LINQ. Incluyen operaciones de filtrado, proyección, agregación, ordenación y otras. Algunos de ellos son (se indica, entre paréntesis, la palabra clave Visual Basic si existe, utilizada en las expresiones de consulta que veremos más adelante):

- **Select**. Proyecta valores basados en una función de transformación.
- **SelectMany** (utilizar varias cláusulas **From**). Proyecta secuencias de valores basados en una función de transformación y, a continuación, los condensa en una sola secuencia (un enumerable).
- **Where**. Selecciona valores basados en una función de predicado.

- **OrderBy** (**Order By**). Ordena los valores de forma ascendente.

- **Join** (**Join … [As …]In … On …**). Combina dos secuencias según las funciones del selector de claves y extrae pares de valores.

- **GroupBy** (**Group … By … Into …**). Agrupa los elementos que comparten un atributo común. Cada grupo se representa mediante un objeto **IGrouping(Of** *TKey*, *T***)**.

- **Count**. Cuenta los elementos de una colección y, opcionalmente, sólo aquéllos que satisfacen una función de predicado.

- **Max**. Determina el valor máximo de una colección.

- **Min**. Determina el valor mínimo de una colección.

- **Sum**. Calcula la suma de los valores de una colección.

- **TakeWhile** (**Take While**). Devuelve los elementos de una colección mientras que el valor de la condición especificada sea **True**.

Puede ver un listado completo de todos los operadores de consulta en un listado proporcionado en la ayuda de Visual Studio.

La mayoría de estos métodos funcionan sobre objetos cuyo tipo implementa la interfaz **IEnumerable(Of** *T***)** (interfaz que proporciona iteración simple en una colección de un tipo especificado) o la interfaz **IQueryable(Of** *T***)** (interfaz para evaluar consultas con respecto a un origen de datos; hereda de **IEnumerable(Of** *T***)**). Se definen como métodos extensores del tipo sobre el que operan. Esto significa que pueden ser llamados utilizando la sintaxis del método estático (o de clase) o la sintaxis del método de un objeto (o instancia).

Muchos operadores de consulta tienen un parámetro de entrada de tipo **Func**. Según esto, veamos un ejemplo de cómo se utiliza uno de estos operadores de consulta, por ejemplo, **Count**:

```
Dim números As Integer() = {6, 7, 11, 9, 8, 5, 4, 1, 3, 2}
Dim númerosPares As Integer = números.Count(Function(n) n Mod 2 = 0)
  ' resultado: 4
```

Compare el argumento pasado con el segundo parámetro del método *NacidosAño* del ejemplo anterior: ambos son de tipo **Func**. La expresión lambda utilizada cuenta aquellos enteros (*n*) que divididos por 2 dan como resto 0. Observe tam-

bién que el compilador puede deducir el tipo del parámetro de entrada; también se puede especificar explícitamente como en el ejemplo siguiente.

El método siguiente generará una secuencia que contiene todos los elementos de la matriz de números que aparecen antes del 11 menores que 9 (6 y 7), ya que 11 es el primer número de la secuencia que no cumple la condición:

```
Dim números As Integer() = {6, 7, 11, 9, 8, 5, 4, 1, 3, 2}
Dim primerosNumsMenoresQue9 =
  números.TakeWhile(Function(n As Integer) n < 9)
```

Las llamadas a los métodos de consulta se pueden encadenar en una sola consulta, lo que permite hacer consultas bastante complejas. Además, según vimos anteriormente, algunos de los operadores de consulta más frecuentemente utilizados poseen una sintaxis de palabras clave específicas del lenguaje Visual Basic que les permite ser llamados como parte de una expresión de consulta. Por ejemplo:

```
Dim numsImparesOrdenados = _
    From n In números _
    Where (n Mod 2 <> 0) _
    Order By n _
    Select n ' resultado: 1, 3, 5, 7, 9, 11
```

Una expresión de consulta constituye una forma diferente de expresar una consulta, más legible que su equivalente basada en métodos. Las cláusulas de las expresiones de consulta se traducen en llamadas a los métodos de consulta en tiempo de compilación. Por ejemplo, la consulta anterior utilizando llamadas a métodos podría realizarse así:

```
Dim numsImparesOrdenados = números.Where( _
  Function(n) n Mod 2 <> 0).OrderBy( _
  Function(n) n).Select(Function(n) n)
```

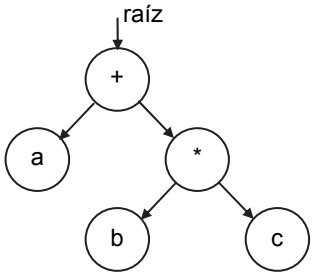
O, de una forma más cercana a la expresión de consulta anterior, así:

```
Dim numsImparesOrdenados = números. _
    Where(Function(n) n Mod 2 <> 0). _
    OrderBy(Function(n) n). _
    Select(Function(n) n)
```

En esta otra versión observamos claramente tipos anónimos, expresiones *lambda*, etc., como base de la expresión de consulta, cuestión que ya indicamos anteriormente.

Árboles de expresiones lambda

Los árboles de expresiones lambda permiten representar expresiones lambda como estructuras de datos en lugar de como código ejecutable. Esto es, las expresiones lambda pueden transformarse en árboles de expresiones para su posterior mamanipulación, almacenamiento o transmisión. Por ejemplo, la expresión *a + b * c* podemos verla bajo la siguiente estructura en la que los datos se almacenan en forma de árbol (el lector puede reconstruir la expresión recorriendo el árbol de la figura en *in-orden*):



Árbol de expresión

Cuando una expresión lambda está asignada a una variable de tipo **Expression(Of** *TDelegado***)**, por ejemplo **Expression(Of Func)**, el compilador emite un árbol de expresión que representa dicha expresión lambda. Por ejemplo, algunos operadores de consulta que se definen en la clase **Queryable** tienen parámetros de tipo **Expression(Of** *TDelegado***)**. Entonces, cuando se llama a estos métodos, la lambda pasada como argumento se compilará produciendo un árbol de expresión.

En LINQ, los árboles de expresiones lambda se utilizan para representar consultas estructuradas para orígenes de datos que implementan **IQueryable(Of** *T***)**. Por ejemplo, el proveedor *LINQ to SQL* implementa esta interfaz para realizar consultas en almacenes de datos relacionales. Por lo tanto, según hemos explicado en el párrafo anterior, el compilador Visual Basic compilará las consultas destinadas a esos orígenes de datos generando un árbol de expresión. Entonces, el proveedor de la consulta podrá recorrer el árbol de expresión y traducirlo en un lenguaje de consulta apropiado para el origen de datos.

¿Cómo se ejecuta un árbol de expresión? Sólo se pueden ejecutar los árboles de expresiones que representen expresiones lambda, los cuales son de tipo **Expression(Of** *TDelegado***)** (por ejemplo, **Expression(Of Func(Of** *CPersona***, Boolean))**) o **LambdaExpression**. Para ejecutar un árbol de expresión primero hay que compilarlo invocando a su método **Compile** con el fin de crear un delegado ejecutable y después hay que invocar al delegado para que se ejecute.

El ejemplo siguiente asigna una expresión lambda a la variable *exprNacidoEnAño* de tipo **Expression(Of Func(Of** *CPersona*, **Boolean))** a partir de la cual el compilador generará un árbol de expresión lambda con la finalidad de obtener de una lista de personas las que han nacido en un año determinado.

```
Dim personas1991 As String = ""
Dim exprNacidoEnAño As Expression(Of Func(Of CPersona, Boolean)) = _
  Function(p) p.FechaNac.Year = 1991
For Each p As CPersona In listPersona
  Dim esNacidoEn As Boolean = exprNacidoEnAño.Compile().Invoke(p)
  If esNacidoEn Then
    personas1991 += p.Nombre + Environment.NewLine
  End If
Next
```

La figura siguiente muestra el árbol de expresión lambda generado a partir de la variable *exprNacidoEnAño* a la que hemos asignado el valor:

```
Function(p) p.FechaNac.Year = 1991 ' (p) => p.FechaNac.Year == 1991
```



Los árboles de expresiones son útiles para crear consultas dinámicas de LINQ necesarias cuando no se conocen los detalles de la consulta durante la compilación; por ejemplo, porque sea necesario especificar durante la ejecución uno o más predicados para filtrar los datos que se desean obtener, lo que implica crear la consulta durante la ejecución. Pues bien, para crear los árboles de expresiones tendremos que utilizar la funcionalidad proporcionada por las clases del espacio de nombres **System.Linq.Expressions**. Por ejemplo, haciendo uso de esta funcionalidad, ¿cómo crearíamos el árbol de expresión de la figura anterior? El código siguiente es la respuesta a esta pregunta (el código siguiente reconstruye la expresión recorriendo el árbol en *postorden*):

```
' Function(p) p.FechaNac.Year = 1991
Dim parametro_p As ParameterExpression = _
  Expression.Parameter(GetType(CPersona), "p")
Dim m1 As MemberExpression = _
  Expression.Property(parametro_p, "FechaNac")
Dim left As MemberExpression = Expression.Property(m1, "Year")
Dim right As ConstantExpression = _
  Expression.Constant(1991, GetType(Integer))
Dim exprb As BinaryExpression = Expression.Equal(left, right)
Dim exprNacidoEnAño As Expression(Of Func(Of CPersona, Boolean)) = _
      Expression.Lambda(Of Func(Of CPersona, Boolean)) ( _
        exprb, _
        New ParameterExpression() {parametro_p})
```

Y si en lugar de utilizar en la expresión lambda una constante (1991) utilizamos una variable *año*, la solución sería esta otra:

```
Dim añoNac As Integer = 1991

' Function(p, año) p.FechaNac.Year = año
Dim parametro_p As ParameterExpression = _
  Expression.Parameter(GetType(CPersona), "p")
Dim m1 As MemberExpression = _
  Expression.Property(parametro_p, "FechaNac")
Dim left As MemberExpression = Expression.Property(m1, "Year")
Dim right As ParameterExpression = _
  Expression.Parameter(GetType(Integer), "año")
    ' right = parámetro año
Dim exprb As BinaryExpression = Expression.Equal(left, right)
Dim exprNacidoEnAño As Expression(Of _
  Func(Of CPersona, Integer, Boolean)) = _
    Expression.Lambda(Of Func(Of CPersona, Integer, Boolean))( _
      exprb, _
      New ParameterExpression() {parametro_p, right})

For Each p In listPersona
  If exprNacidoEnAño.Compile().Invoke(p, añoNac) Then
```

```
    personas1991 += p.Nombre + Environment.NewLine
  End If
Next
```

La clase **Expression** proporciona la funcionalidad necesaria para crear nodos del árbol de expresión de tipos específicos; por ejemplo, un objeto **ParameterExpression**, que representa una expresión de parámetro con nombre, un objeto **MemberExpression**, que representa un acceso a un campo o propiedad, un objeto **ConstantExpression**, que representa una expresión que tiene un valor constante, un objeto **MethodCallExpression**, que representa una llamada a un método, un objeto **ConditionalExpression**, que representa una expresión que tiene un operador condicional, un objeto **BinaryExpression**, que representa una expresión que tiene un operador binario, o un objeto **LambdaExpression**, que describe una expresión lambda; todos definidos en el espacio de nombres **System.Linq.Expressions**. Estas clases se derivan de la clase abstracta **Expression**.

EXPRESIONES DE CONSULTA

Al principio de este tema dijimos que LINQ es una combinación de extensiones al lenguaje y bibliotecas de código administrado que permite expresar de manera uniforme "consultas" sobre colecciones de datos de diversa procedencia (objetos en memoria, bases de datos relacionales o documentos XML) utilizando recursos del propio lenguaje de programación. Estas consultas son llevadas a cabo mediante lo que se denomina *expresiones de consulta*.

Por lo tanto, las expresiones de consulta responden a una nueva sintaxis que se ha añadido al lenguaje Visual Basic (y a C#) y pueden actuar sobre objetos que implementen la interfaz **IEnumerable(Of** *T***)** o **IQueryable(Of** *T***)**, entre los que se incluyen las matrices, transformándolos mediante un conjunto de operaciones en otras colecciones que implementen la misma interfaz.

Para explicar con más claridad las expresiones de consulta, supongamos que hemos definido las clases *CPais* y *CPersona* así:

```
Public NotInheritable Class CPais
  Public Property Codigo() As String
  Public Property Nombre() As String
End Class

Public NotInheritable Class CPersona
  Public Property Nombre() As String
  Public Property FechaNac() As DateTime
  Public Property PaisNac() As String
End Class
```

Desde el punto de vista de LINQ una consulta no es más que una expresión que recupera datos de un origen de datos. Todas las operaciones de consulta LINQ se componen de tres acciones distintas:

1. Obtención del origen de datos. Por ejemplo:

```
Dim listPersona = New List(Of CPersona)()
listPersona.AddRange(New CPersona() { _
  New CPersona With { _
    .Nombre = "Isabella", _
    .FechaNac = New DateTime(2011, 11, 7), .PaisNac = "US"}, _
  New CPersona With { _
    .Nombre = "Manuel", _
    .FechaNac = New DateTime(1991, 9, 21), .PaisNac = "ES"}, _
  New CPersona With { _
    .Nombre = "Javier", _
    .FechaNac = New DateTime(1990, 7, 2), .PaisNac = "ES"}, _
  New CPersona With { _
    .Nombre = "María", _
    .FechaNac = New DateTime(1991, 9, 1), .PaisNac = "EN"} _
})
```

2. Creación de la consulta. Por ejemplo:

```
Dim personas1990 = From p In listPersona _
    Where p.FechaNac.Year = 1990 _
    Order By p.Nombre _
    Select New With {.Nombre = p.Nombre}
```

3. Ejecución de la consulta. En LINQ, la ejecución de la consulta es distinta de la propia consulta; dicho de otra forma, no se recuperan los datos con la simple creación de la variable de consulta (*personas1990* en nuestro caso), sino que hay que ejecutarla, por ejemplo, en una instrucción **For Each**:

```
For Each persona In personas1990
  Console.WriteLine(persona.Nombre)
```

En el ejemplo siguiente se muestra cómo se expresan las tres partes de una operación de consulta en una aplicación concreta. En este ejemplo se utiliza por comodidad una matriz de objetos como origen de datos, pero los mismos conceptos se aplican a otros orígenes de datos. En dicho ejemplo, se realiza una consulta sobre la colección **List** que hemos venido utilizando en los ejemplos anteriores, con el fin de obtener la colección de personas nacidas en un año determinado:

```
Imports System
Imports System.Collections.Generic
Imports System.Linq

Public NotInheritable Class CPais
```

```
  Public Property Codigo() As String
  Public Property Nombre() As String
End Class

Public NotInheritable Class CPersona
  Public Property Nombre() As String
  Public Property FechaNac() As DateTime
  Public Property PaisNac() As String
End Class

Module Test
  Sub Main()
    Dim listPais = New List(Of CPais)()
    listPais.AddRange(New CPais() { _
      New CPais With { _
        .Codigo = "ES", .Nombre = "España"}, _
      New CPais With { _
        .Codigo = "EN", .Nombre = "Inglaterra"}, _
      New CPais With { _
        .Codigo = "FR", .Nombre = "Francia"}, _
      New CPais With { _
        .Codigo = "US", .Nombre = "Estados Unidos"} _
     })

    Dim listPersona = New List(Of CPersona)()
    listPersona.AddRange(New CPersona() { _
      New CPersona With { _
        .Nombre = "Elena", _
        .FechaNac = New DateTime(2011, 11, 7), .PaisNac = "US"}, _
      New CPersona With { _
        .Nombre = "Manuel", _
        .FechaNac = New DateTime(1991, 9, 21), .PaisNac = "ES"}, _
      New CPersona With { _
        .Nombre = "Javier", _
        .FechaNac = New DateTime(1990, 7, 2), .PaisNac = "ES"}, _
      New CPersona With { _
        .Nombre = "María", _
        .FechaNac = New DateTime(1991, 9, 1), .PaisNac = "EN"} _
     })

    Dim personas1990 = From p In listPersona _
        Where p.FechaNac.Year = 1990 _
        Order By p.Nombre _
        Select New With {.Nombre = p.Nombre}

    For Each persona In personas1990
      Console.WriteLine(persona.Nombre)
    Next
  End Sub
End Module
```

Observemos la expresión de consulta (código sombreado). A cualquiera que esté familiarizado con la sentencia SELECT de SQL, le habrá resultado fácil entender dicha expresión. Quizás nos sorprenda que la cláusula **Select** no esté al principio. La razón es que si estuviera al principio sería imposible ofrecer la ayuda inteligente a la hora de escribir la expresión de consulta, porque aún no se habría especificado la colección de objetos sobre los que se ejecutará la consulta. Esto es, si *listPersona* es de tipo **List(Of** *CPersona***)** (**List** implementa la interfaz **IEnumerable**), entonces se deduce que *p* es de tipo *CPersona*, con lo que el sistema podrá verificar la sintaxis cuando a continuación escribamos el resto de las cláusulas.

La consulta del ejemplo anterior devuelve todos los nombres de la lista *listPersona* (objeto **List**) de objetos *CPersona* ordenados ascendentemente. Contiene cuatro cláusulas: **From**, **Where**, **Order By** y **Select**. La cláusula **From** especifica el origen de datos y una variable local que representa cada elemento en la secuencia de origen, la cláusula **Where** aplica el filtro (se pueden utilizar los operadores lógicos), la cláusula **Order By** ordena el resultado ateniéndose al dato especificado y la cláusula **Select** especifica el tipo de los elementos devueltos. No olvide que definir la variable de consulta no realiza ninguna acción ni devuelve datos, simplemente almacena la información necesaria para generar los resultados cuando la consulta se ejecute posteriormente.

El ejemplo anterior podría también haberse escrito así:

```
Dim personas1990 = From p In listPersona _
    Where p.FechaNac.Year = 1990 _
    Order By p.Nombre _
    Select p.Nombre

For Each persona In personas1990
  Console.WriteLine(persona)
```

¿Cuál es la diferencia? En el ejemplo anterior *personas1990* era una colección de objetos de tipo anónimo con una propiedad *Nombre* de tipo **String** y en éste, *personas1990* es una colección de objetos **String**.

Compilación de una expresión de consulta

Cuando el compilador encuentra una expresión de consulta la transforma en una consulta compuesta por llamadas a métodos. Por ejemplo:

```
Dim personas1990 = listPersona. _
    Where(Function(p) p.FechaNac.Year = 1990). _
    OrderBy(Function(p) p.Nombre). _
    Select(Function(p) New With {.Nombre = p.Nombre})
```

En el código anterior observamos llamadas a los métodos **Where**, **OrderBy** y **Select**. Observemos la firma de uno de ellos:

```
<ExtensionAttribute> _
Public Shared Function Where(Of TOrigen) ( _
    origen As IEnumerable(Of TOrigen), _
    predicado As Func(Of TOrigen, Boolean) _
) As IEnumerable(Of TOrigen)
```

Se trata de un método extensor, como ya habíamos estudiado anteriormente, que, por definición, se puede invocar como un método estático (se ha declarado **Shared**) o como un método de un objeto (es una extensión a los métodos del tipo **IEnumerable**). El primer parámetro hace referencia al tipo extendido, en este caso al tipo **IEnumerable** ya que hemos partido de que los objetos sobre los que operarán las expresiones de consulta tienen que implementar esta interfaz, y el segundo, el predicado (el filtro), es un delegado que hace referencia a un método que recibirá como argumento un objeto de tipo *TOrigen* y devolverá un valor de tipo **Boolean**. La función lambda es la que se transformará en ese delegado anónimo.

¿Por qué estos métodos se han implementado como métodos extensores? Pues para que LINQ sea una arquitectura abierta y extensible. Según esto, añadir nuestro propio método **Where** es tan sencillo, según explicamos anteriormente en el apartado *Métodos extensores*, como añadir a nuestra aplicación un procedimiento (**Function** o **Sub**) marcado con el atributo **<Extension()>** del espacio de nombres **System.Runtime.CompilerServices** con un primer parámetro del tipo de la clase a extender. Por ejemplo, añadamos a la aplicación anterior la siguiente clase para extender el tipo **IEnumerable** con el siguiente método **Where**:

```
Imports System
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

<Extension()> _
Public Function Where(Of T)( _
       origen As IEnumerable(Of T), _
       predicado As Func(Of T, Boolean)) _
       As IEnumerable(Of T)
  Dim lista As List(Of T) = New List(Of T)()
  For Each p In origen
    If predicado(p) Then
      lista.Add(p)
    End If
  Next
  Return lista
End Function
```

Si compila y ejecuta ahora la aplicación, observará que al ejecutarse la consulta, el método **Where** invocado es éste que hemos añadido. Precisamente en esto consiste la arquitectura abierta de LINQ: cualquiera puede añadir otros operadores de consulta siempre que cumplan con las firmas que exige el compilador. Precisamente ésta ha sido la vía a través de la cual se han integrado en el lenguaje las extensiones LINQ.

Sintaxis de las expresiones de consulta

Básicamente, una expresión de consulta siempre comienza con la cláusula **From ... In**, en la que se especifica una variable local que representa a cada elemento en la secuencia de origen así como el origen de datos, y a continuación se pueden escribir una o más cláusulas **From ... In**, **Let**, **Where**, **Join ... In ... On ... Equals**, **Join ... In ... On ... Equals ... Into**, **Order By ...** [**Ascending** | **Descending**], **Select**, **Group ... By**, o **Group ... By ... Into**. Opcionalmente, al final de la expresión puede escribirse una cláusula de continuación que comienza con **Into** y continúa con el cuerpo de otra consulta.

Cláusula Group

La cláusula **Group** permite agrupar los resultados según la clave que se especifique. Por ejemplo, la siguiente expresión de consulta genera una secuencia *personasPorAño* con las personas de la lista *listPersona* agrupadas por años:

```
Dim personasPorAño = From p In listPersona _
    Group p By p.FechaNac.Year Into gr = Group

For Each grupoPersonas In personasPorAño
  Console.WriteLine(grupoPersonas.Year)
  For Each persona In grupoPersonas.gr
    Console.WriteLine(" {0}", persona.Nombre)
  Next
Next
```

El resultado es una secuencia de elementos de tipo **IGrouping(Of** *TKey*, *T***)** (hereda de **IEnumerable(Of** *T***)**) que define una propiedad, en nuestro caso *Year*, como clave de agrupación (de tipo **Integer**).

Este otro ejemplo, haciendo uso de la lista de países y de la de personas, agrupa las personas según su país de nacimiento:

```
Dim personasPais = From pais In listPais _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Group New With {.Nombre = pers.Nombre} By pais.Nombre _
    Into gr = Group
```

```
For Each grupoPerPais In personasPais
  Console.WriteLine(grupoPerPais.Nombre) ' nombre país
  For Each persona In grupoPerPais.gr
    Console.WriteLine(" {0}", persona.Nombre)
  Next
Next
```

Productos cartesianos

En bases de datos, el producto cartesiano de dos tablas no es más que otra tabla resultante de combinar cada fila de la primera con cada fila de la segunda. En LINQ podemos aplicar esta definición utilizando dos cláusulas **From** (un producto cartesiano se implementa mediante el método **SelectMany**). Por ejemplo:

```
Dim productoCartesiano = _
    From pais In listPais _
    From pers In listPersona _
    Select New With {.NomPais = pais.Nombre, _
                     .NomPers = pers.Nombre}
For Each elem In productoCartesiano
  Console.WriteLine("{0} {1}", elem.NomPers, elem.NomPais)
Next
```

Es recomendable evitar los productos cartesianos por la explosión combinatoria que generan, o crearlos con restricciones. Por ejemplo, la siguiente expresión de consulta muestra el nombre de las personas junto con el país donde nacieron:

```
Dim productoCartesiano = _
    From pais In listPais _
    From pers In listPersona _
    Where pais.Codigo = pers.PaisNac _
    Select New With {.NomPers = pers.Nombre, _
                     .NomPais = pais.Nombre}
```

Cláusula Join

La cláusula **Join** se utiliza para realizar una operación de combinación (**Join** funciona siempre con colecciones de objetos, en lugar de con tablas de base de datos, aunque en LINQ no es necesario utilizar esta cláusula tan a menudo como en SQL, porque las claves externas en LINQ se representan en el modelo de objetos como propiedades que contienen una colección de elementos). Con **Join** se trata de limitar las combinaciones que produciría un producto cartesiano, manteniendo únicamente los elementos de las secuencias que casan de acuerdo con el criterio establecido. Por ejemplo, la combinación que realizamos en el apartado anterior (producto cartesiano) podríamos escribirla mejor así, ya que el rendimiento es muy superior:

```
Dim combinacion = _
    From pais In listPais _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Select New With {.NomPers = pers.Nombre, _
                     .NomPais = pais.Nombre}

For Each elem In combinacion
  Console.WriteLine("{0} {1}", elem.NomPers, elem.NomPais)
```

Cláusula Into

La cláusula **Into** puede utilizarse para crear un identificador temporal que almacene los resultados de una cláusula **Group**, **Join** o **Select** en un nuevo identificador. Por ejemplo, la siguiente expresión de consulta genera una secuencia *persNacidasPorAño* con los años de nacimiento de las personas de la lista *listPersona* agrupadas por año de nacimiento más el número de personas de cada grupo, pero sólo los grupos con un número mínimo de personas:

```
Dim persNacidasPorAño = _
    From p In listPersona _
    Group p By p.FechaNac.Year Into grupoPersAño = Group _
    Where grupoPersAño.Count() >= 2 _
    Select New With {.Año = Year, .PorAño = grupoPersAño.Count()}

For Each persona In persNacidasPorAño
  Console.WriteLine("En {0} nacieron {1} o más personas.", _
                    persona.Año, persona.PorAño)
```

¿Recuerda el resultado de la expresión de consulta siguiente?

```
Dim personasPais = _
    From pais In listPais _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Group New With {.Nombre = pers.Nombre} By pais.Nombre _
    Into gr = Group
```

Era éste:

```
España
    Manuel
    Javier
Inglaterra
    María
Estados Unidos
    Isabella
```

¿Cómo modificamos la expresión de consulta para que los países aparezcan en orden ascendente? Pues introduciendo ese resultado en una secuencia y ordenando ésta. Para ello utilizaremos **Into** con **Group**. Esto es:

```
Dim persPais = _
    From pais In listPais _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Group New With {.Nombre = pers.Nombre} By pais.Nombre _
    Into persPaisOrd = Group _
    Order By Nombre _
    Select persPaisOrd
```

Otro ejemplo. ¿Cómo obtenemos una lista de países ordenada ascendentemente con el número de personas por país? Combinamos cada país con las personas de ese país y las contamos. Para ello utilizaremos **Into** con **Group**. Esto es:

```
Dim PaisNumPers = _
    From pais In listPais Order By pais.Nombre _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Group By pais.Nombre Into grupoPais = Group _
    Select New With {.NomPais = Nombre, _
                     .NumPers = grupoPais.Count()}

    For Each pais In PaisNumPers
      Console.WriteLine("En {0} hay {1} personas.", _
      pais.NomPais, pais.NumPers)
    Next
```

Cláusula Let

La cláusula **Let** se utiliza para almacenar el resultado de una subexpresión con el fin de utilizarlo en cláusulas posteriores. Por ejemplo, la siguiente expresión de consulta, haciendo uso de la cláusula **Let**, genera una secuencia *persPorAñoMes* con las personas que han nacido en un mes y un año determinados:

```
Dim unMes As Integer = 9, unAño As Integer = 1991
Dim persPaisAñoMes = _
    From pais In listPais Order By pais.Nombre _
    Join pers In listPersona On pais.Codigo Equals pers.PaisNac _
    Let mesNac = pers.FechaNac.Month _
    Let añoNac = pers.FechaNac.Year _
    Where añoNac = unAño AndAlso mesNac = unMes _
    Group New With {.Nombre = pers.Nombre} By pais.Nombre _
    Into perPais = Group

Console.WriteLine("Nacidos en el mes {0} del año {1}:", unMes, unAño)
For Each grupoPerPais In persPaisAñoMes
  Console.WriteLine(grupoPerPais.Nombre)
```

```
  For Each persona In grupoPerPais.perPais
    Console.WriteLine(" {0}", persona.Nombre)
  Next
Next
```

PROVEEDORES DE LINQ

En LINQ distinguimos proveedores locales y proveedores remotos. Los proveedores locales son aquéllos que operan sobre objetos en memoria (es lo que hemos estado haciendo en los ejemplos anteriores). Estos objetos tienen que implementar la interfaz **IEnumerable(Of *T*)**. A esta categoría pertenecen *LINQ to Objects* (el proveedor que hemos venido utilizando hasta ahora), *LINQ to DataSet* y *LINQ to XML*. Y los proveedores remotos son aquéllos que operan sobre bases de datos relacionales. En este caso, el mecanismo basado en el recorrido de colecciones de objetos en memoria (más bien secuencias), que tan bien nos ha funcionado en los proveedores locales, no resulta adecuado en los proveedores remotos, simplemente porque cualquier implementación de un operador de consulta basada en la navegación de cursores (un cursor puede verse como un iterador sobre la colección de filas de un conjunto de datos obtenido después de una consulta SQL) sería un fracaso desde el punto de vista de rendimiento. Debido a esto, se definió la interfaz **IQueryable(Of *T*)**, que hereda de **IEnumerable(Of *T*)**, que es la que implementan los proveedores *LINQ to SQL* y *LINQ to Entities*. Quiere esto decir que los objetos que implementen la interfaz **IQueryable(Of *T*)** pueden ser también orígenes de consultas integradas, pero lo que verdaderamente aporta esta interfaz es un conjunto de métodos extensores con una implementación más adecuada para interaccionar con bases de datos relacionales que la proporcionada por los métodos de *LINQ to Objects*.

Centrándonos ya en los proveedores *LINQ to Objects* y *LINQ to SQL/Entities*, podemos decir que ambos proporcionan prácticamente los mismos operadores de consulta. La diferencia está en que los métodos extensores de *LINQ to SQL/Entities*, en lugar de recibir delegados como parámetros, reciben árboles de expresiones. No obstante, esto no es un problema ya que, según vimos anteriormente en este mismo capítulo, una expresión lambda puede transformarse en un delegado y también en un árbol de expresión.

Todos los métodos incorporan una referencia a un árbol de expresión que define el algoritmo de obtención de la secuencia. Como ejemplo, observe la firma del método **Where** aportado por la interfaz **IQueryable(Of *T*)** y compárelo con la que vimos anteriormente:

```
<ExtensionAttribute> _
Public Shared Function Where(Of TOrigen) ( _
    origen As IQueryable(Of TOrigen), _
```

```
    predicado As Expression(Of Func(Of TOrigen, Boolean)) _
) As IQueryable(Of TOrigen)
```

Observamos que este método tiene al menos un parámetro de tipo **Expression(Of** *TDelegado***)** cuyo argumento de tipo es un delegado **Func**. Por lo tanto, es posible pasar una expresión lambda y compilarla en un objeto **Expression(Of** *TDelegado***)**. Finalmente, la consulta que se obtiene como resultado de ejecutar un árbol de expresión que representa al método **Where** que realiza la llamada devolverá los elementos de *origen* que satisfagan la condición especificada en *predicado*.

Lo explicado para el método **Where** puede aplicarse a otros métodos.

ENTITY FRAMEWORK

ADO.NET *Entity Framework* permite a los desarrolladores crear aplicaciones que acceden a bases de datos elevando el nivel de abstracción, del nivel lógico relacional al nivel conceptual. En este nivel de abstracción superior, *Entity Framework* admite código que es independiente de cualquier motor de almacenamiento de datos o esquema relacional determinados. Pues bien, utilizando LINQ, concretamente el proveedor *LINQ to Entities*, es posible consultar las entidades que definen el modelo conceptual de *Entity Framework*.

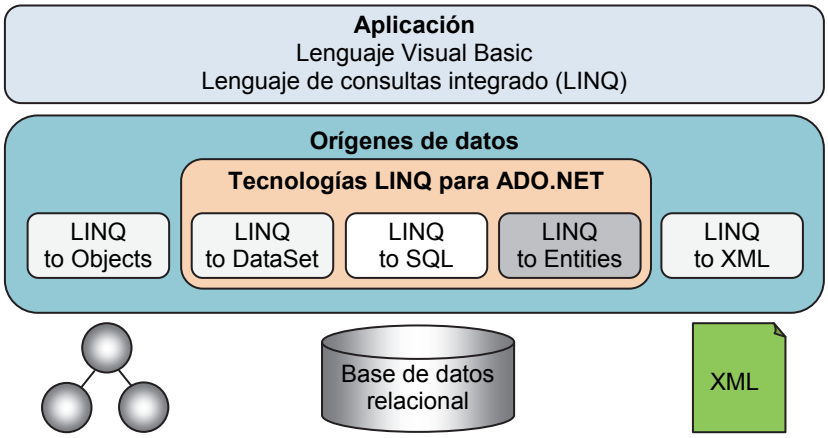
Para ello, durante el diseño de la aplicación, asignaremos el modelo de datos relacional de una base de datos a un modelo de objetos expresado en el lenguaje de programación del programador, para después realizar las consultas sobre el modelo de objetos. Estas consultas serán convertidas por *Entity Framework* a SQL y enviadas a la base de datos para su ejecución. Cuando la base de datos devuelva los resultados, *Entity Framework* los vuelve a convertir en objetos expresados en el propio lenguaje de programación utilizado.

Hay una gran diferencia entre *Entity Framework* y LINQ. LINQ es un lenguaje de consulta integrado que se puede ejecutar sobre varias fuentes por medio de los distintos proveedores de LINQ desarrollados hasta la fecha. Estas fuentes y los proveedores correspondientes son:

- DataSet: *LINQ to DataSet*.
- XML: *LINQ to XML*.
- Objetos de memoria: *LINQ to Objects*.
- Bases de datos relacionales: *LINQ to SQL* y *Entity Framework*.

Todos estos proveedores, excepto *Entity Framework*, se lanzaron con .NET Framework 3.5 y a partir de .NET 4.0, *Entity Framework* (*LINQ to Entities*) será

el proveedor LINQ recomendado por Microsoft para acceso a bases de datos relacionales. ¿Por qué? Porque permite a una aplicación conectarse a fuentes de datos de diferentes proveedores (*LINQ to SQL* está prácticamente destinado a trabajar con SQL Server), permite modelar distintos tipos de herencia (por jerarquía, por subtipo o herencia por tipo concreto), soporta asociaciones de "muchos a muchos" (*LINQ to SQL* sólo permite la correlación o mapeo "uno a uno"), su nivel de abstracción es elevado y su capa de *Servicios de objetos* es mucho más amplia y está orientada a no ser simplemente un ORM (*Object-Relational Mapping*), sino todo un lenguaje conceptual a la hora de desarrollar aplicaciones conectadas a datos y porque la lógica se puede extender y escalar más fácilmente.



MARCO DE ENTIDADES DE ADO.NET

El marco de entidades de ADO.NET (*Entity Framework* o *EF*) es un entorno de desarrollo de la plataforma .NET que permite superponer varias capas de abstracción sobre las bases de datos relacionales (véase la figura que presentamos un poco más adelante) con el fin de hacer posible una programación más conceptual (basada en los conceptos del dominio con el que se trabaja) y reducir al mínimo el desajuste de impedancias causado por las diferencias entre los modelos orientados a objetos y los modelos relacionales.

Para minimizar ese desajuste de impedancias se ha recurrido a un patrón de diseño muy común para el modelado de datos y que consiste en dividir este modelado en tres partes:

- Un *modelo conceptual* para definir las entidades y relaciones del sistema que se está modelando.
- Un *modelo lógico* para describir la base de datos relacional: tablas, relaciones entre las tablas con restricciones de claves externas, etc.

- Y un *modelo físico* para definir detalles específicos (almacenamiento, índices, etc.) en función del motor de bases de datos utilizado.

De todos es conocido que muchos desarrolladores omiten la creación del modelo conceptual y comienzan especificando las tablas, columnas y claves en una base de datos relacional. Pues bien, *EF* nace con la idea de rescatar el modelo conceptual permitiendo a los programadores consultar las entidades y relaciones que lo forman y traduciendo esas consultas en órdenes específicas del origen de datos. Para ello es necesario que exista una correlación entre estos dos mundos diferentes: modelo conceptual y modelo lógico. Pues bien, lo que hace *EF* es expresar esta correlación en una especificación externa denominada *Modelo de entidades* (EDM: *Entity Data Model*).

El *modelo de entidades* aporta múltiples características para eliminar el desajuste de impedancias con el que se encuentran los programadores orientados a objetos con respecto a los modelos relacionales implementados por los profesionales de bases de datos. Está construido sobre las tres secciones siguientes, definidas, como veremos más adelante, en un fichero *.edmx*:

- SSDL (*Storage Schema Definition Languaje*). Define el espacio S (*Storage*). Describe, en formato XML, el modelo relacional de la base de datos subyacente.

- CSDL (*Conceptual Schema Definition Languaje*). Define el espacio C (*Conceptual*). Describe, en formato XML, las entidades que deseamos tener en nuestro modelo conceptual, así como las propiedades de navegación o asociaciones entre las distintas entidades.

- MSL (*Mapping Schema Languaje*). Describe, en formato XML, cómo se asocian o relacionan las entidades del modelo conceptual (CSDL) con las tablas, relaciones, columnas y demás elementos del modelo relacional.

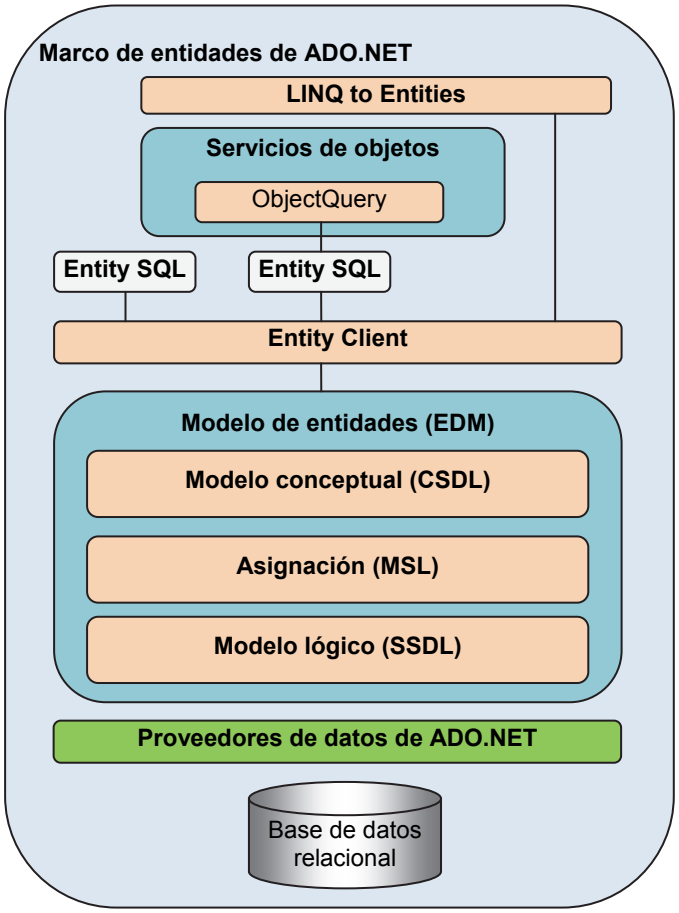
Y, ¿cómo realiza *EF* las consultas sobre el modelo conceptual? Pues utilizando algunos de los mecanismos siguientes:

- *LINQ to Entities*. Es el proveedor de LINQ diseñado para consultar las entidades que definen el modelo conceptual. Como alternativas a LINQ, tenemos la posibilidad de utilizar procedimientos almacenados, o bien expresar directamente las consultas en *eSQL*.

- *Entity SQL* (*eSQL*). Es el lenguaje de consulta de modelos conceptuales; se deriva de SQL. Es precisamente el lenguaje de entrada para el proveedor de entidades (*Entity Client*). Se trata de un lenguaje de consulta de modelos de entidades, independiente de cualquier base de datos específica, que *Entity*

Client se encarga de traducir en sentencias aceptables para la base de datos específica sobre la que se ha construido el modelo conceptual.

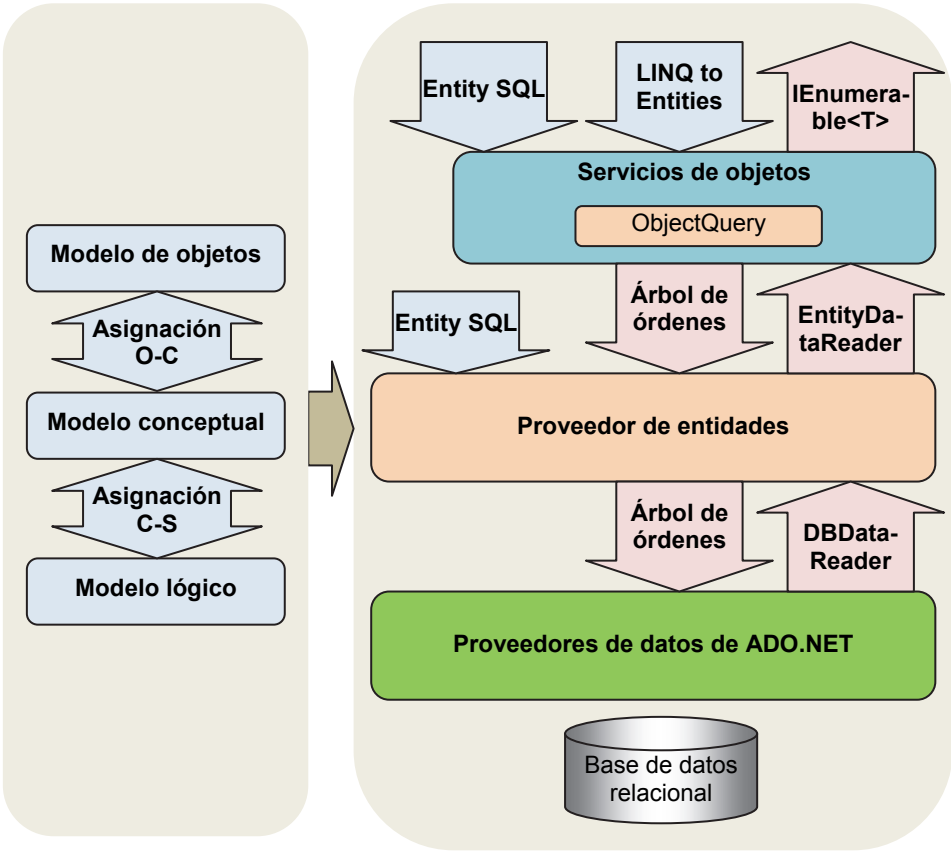
- *Métodos del generador de consultas*. La clase **ObjectQuery** proporciona un conjunto de métodos que se pueden utilizar para construir consultas equivalentes a las obtenidas con *Entity SQL*.



Entity Client es el proveedor de entidades de ADO.NET. Este proveedor, a diferencia de los proveedores de datos de ADO.NET que trabajan sobre el modelo físico de la base de datos, trabajará sobre el modelo de entidades. Su misión es administrar las conexiones, traducir las consultas de entidad en consultas específicas del origen de datos y devolver un lector de datos que los *Servicios de objetos* materializarán en objetos si esta operación es requerida; en otro caso, se puede usar el lector de datos devuelto.

Servicios de objetos. Permite definir cada uno de los objetos de entidad así como el contexto de objetos a los que se refiere el *modelo de objetos*. Este modelo define el espacio O (*Object*).

El diagrama siguiente muestra cómo se accede a los datos:



En la figura anterior, a la izquierda, distinguimos un primer nivel de abstracción, denominado *Entity Data Model* (EDM), que se corresponde con un modelo entidad-relación o modelo conceptual que ofrece una capa de asignación C-S (*Conceptual–Storage*) que nos permite operar sobre el modelo lógico desde las entidades del modelo conceptual, y un segundo nivel de abstracción que se corresponde con un modelo de objetos que ofrece una capa de asignación O-C (*Object–Conceptual*) que nos permite operar sobre el modelo lógico, pasando por el modelo conceptual.

A la derecha de la figura se esquematizan las herramientas *Entity Data Model* (bibliotecas de clases) de las que disponemos para realizar esas operaciones que resumimos a continuación.

Una flecha hacia abajo indica una consulta sobre el origen de datos y una hacia arriba indica los datos devueltos. Los servicios de objetos generan un árbol de órdenes canónico que representa la consulta LINQ o una operación de *Entity SQL* con el modelo conceptual. Después, el proveedor de entidades transforma este árbol, basado en el modelo EDM, en un nuevo árbol que es una operación equivalente en el origen de datos (SELECT, UPDATE, INSERT y DELETE).

Consultar un modelo de objetos

El *modelo de objetos* permite consultar, insertar, actualizar y eliminar datos, expresados como objetos de los tipos de entidad que se definen en el EDM. Las consultas son realizadas a través de los *servicios de objetos* y pueden ser consultas LINQ o consultas *Entity SQL* encapsuladas en un objeto de la clase **ObjectQuery**. Así mismo, los servicios de objetos propagarán los cambios realizados sobre los objetos hacia el origen de datos y materializarán los datos devueltos en objetos. También proporcionan medios para realizar el seguimiento de los cambios, enlazar los objetos a los controles y controlar la simultaneidad. Estos servicios están definidos por las clases de los espacios de nombres **System.Data.Objects** y **System.Data.Objects.DataClasses** y las clases que dan lugar a los objetos de este modelo se derivan de éstas, como por ejemplo de **ObjectContext** o **EntityObject**.

Por ejemplo, supongamos que hemos construido un EDM cuyo modelo de objetos está definido por el contexto de objetos *bd_notasAlumnosEntities*, derivado de **ObjectContext**, y por las entidades correspondientes, como *alumno*, derivadas de **EntityObject** (en el próximo apartado veremos cómo se construye un EDM a partir de una base de datos). Partiendo de este supuesto, el siguiente ejemplo indica cómo sería una consulta LINQ frente a la misma consulta *Entity SQL*:

```
Dim contextoDeObjs As New bd_notasAlumnosEntities()

' Consulta LINQ
Dim consulta = _
    From alums In contextoDeObjs.alumnos _
    Select alums.nombre

For Each nomAlum In consulta
  Console.WriteLine(nomAlum)

' Consulta Entity SQL
Dim sConsulta As String = "SELECT VALUE alums.nombre " & _
  "FROM bd_notasAlumnosEntities.alumnos AS alums"
Dim consulta = _
  New ObjectQuery(Of String)(sConsulta, contextoDeObjs)
```

```
For Each nomAlum In consulta
  Console.WriteLine(nomAlum)
Next
```

La cláusula **FROM** se puede utilizar para especificar uno o varios orígenes, separados por comas, para una instrucción **SELECT**. La forma más sencilla es la del ejemplo anterior: una expresión de una única consulta que identifica una colección y un alias que se usa como origen en una instrucción **SELECT**. La cláusula **SELECT** se evalúa después de la cláusula **FROM**.

El objeto *consulta* de tipo **ObjectQuery(Of String)** implementa la interfaz **IQueryable(Of** *T***)**, por lo que puede ser accedido a través de una sentencia **For Each**.

ObjectQuery también implementa un conjunto de métodos del generador de consultas que se pueden utilizar para construir la consulta equivalente a la expresada mediante *eSQL*. El resultado es una expresión más cercana a LINQ y, por lo tanto, evita tener que conocer el lenguaje *eSQL*. Algunos de estos métodos son: **Select**, **Where**, **GroupBy** u **OrderBy**. Por ejemplo:

```
' Métodos del generador de consultas
Dim consulta = _
  New ObjectQuery(Of alumnos)("alumnos", contextoDeObjs)
consulta = consulta.Where("it.id_alumno=1234567")

For Each alum In consulta
  Console.WriteLine(alum.nombre)
Next
```

Observe la utilización de *it* (“eso”) en el predicado. Es una palabra reservada predefinida para hacer referencia al nombre de la entidad que se consulta.

El *modelo conceptual* contiene entidades (por ejemplo, *alumno*) y asociaciones (por ejemplo, la relación *alumno-alums_asigs*). Las consultas sobre este modelo son realizadas por medio del proveedor de entidades *Entity Client*, que proporciona clases como **EntityConnection**, **EntityCommand** o **EntityDataReader** del espacio de nombres **System.Data.EntityClient**, utilizando el lenguaje *Entity SQL*. Por ejemplo:

```
' Consulta con Entity Client
Dim sConexion As String = ConfigurationManager. _
  ConnectionStrings("bd_notasAlumnosEntities").ConnectionString
Dim conexion = New EntityConnection(sConexion)
Dim sConsulta As String = "SELECT VALUE alums.nombre " & _
  "FROM bd_notasAlumnosEntities.alumnos AS alums"
Dim consulta = New EntityCommand(sConsulta, conexion)
```

```
conexion.Open()
Dim lector As EntityDataReader = _
  consulta.ExecuteReader(CommandBehavior.SequentialAccess)
While lector.Read()
  Console.WriteLine(lector(0))
End While
conexion.Close()
```

El *modelo lógico* contiene tablas, vistas, procedimientos almacenados y funciones definidas por el usuario. Las consultas son realizadas utilizando las clases tradicionales de ADO.NET que vimos en el capítulo *Acceso a una base de datos* (por ejemplo, **SqlConnection**, **SqlCommand** o **SqlDataReader**). Por ejemplo:

```
' Consulta con SqlClient
Dim sConexion As String = _
  "Data Source=.\sqlexpress;Initial Catalog=bd_notasAlumnos;" & _
  "Integrated Security=True"
Dim conexion = New SqlConnection(sConexion)
Dim sConsulta As String = "SELECT alumnos.nombre FROM alumnos"
Dim consulta = New SqlCommand(sConsulta, conexion)
conexion.Open()
Dim lector As SqlDataReader = consulta.ExecuteReader()
While lector.Read()
  Console.WriteLine(lector(0))
End While
conexion.Close()
```

Finalmente, mencionar algunos de los patrones de diseño como *Data Mapper*, *Value Object* (también denominado *Data Transfer Object*) y *Lazy Loading*, muy unidos a *EF*. El patrón de diseño *Data Mapper* tiene como fin separar un modelo de objetos de un modelo relacional y realizar la transferencia de datos entre ambos. *Value Object* es un objeto que permite transportar de una vez una colección de datos entre dos capas diferentes de una aplicación. Y el patrón de diseño *Lazy Loading* permite retardar la carga de un objeto resolviendo problemas de carga desmesurada y dependencias circulares; por ejemplo, piense en una relación uno a muchos, donde un objeto de negocio *Object01* posee una colección de otros objetos *Object02*; cuando el objeto *Object01* es solicitado, se trae a memoria sólo este objeto, con la colección de objetos vacía y los objetos *Object02* serán cargados posteriormente, cuando realmente se necesiten.

ACCESO A UNA BASE DE DATOS

Vamos a realizar una aplicación que trabaje con los datos de una base de datos SQL Server. Para empezar, elegiremos la base de datos *bd_notasAlumnos* que utilizamos en el capítulo *Acceso a una base de datos* (notas de los alumnos matriculados en un determinado centro de unas determinadas asignaturas).



Empecemos por crear una aplicación con Visual Studio denominada *ApEntidades*. Para ello, abra Visual Studio y cree un nuevo proyecto de tipo *Visual Basic > Windows > Aplicación WPF*.

Conectarse a la base de datos

Abra el *Explorador de servidores* (*Explorador de bases de datos*) y añada una conexión a la base de datos *bd_notasAlumnos* (este proceso ya fue explicado en el apartado *Conectarse a la base de datos Sql Server* del capítulo *Acceso a una base de datos*).

Generar el modelo de entidades

El modelo de objetos que ofrece una capa de asignación O-C (*Object–Conceptual*), expresado en el lenguaje de programación con el que se esté escribiendo la aplicación (en nuestro caso Visual Basic), nos va a proporcionar un nivel de abstracción que, pasando por el modelo conceptual, nos permitirá operar sobre el modelo lógico. Gracias al modelo de objetos, que para nosotros representa una base de datos orientada a objetos virtual imagen del modelo relacional, podemos abstraernos no sólo del modelo lógico, sino del lenguaje SQL correspondiente al motor de base de datos que estemos utilizando. Para generar automáticamente este modelo a partir de una base de datos existente hay tres herramientas que se usan conjuntamente y sirven de ayuda para generar, modificar y actualizar un EDM:

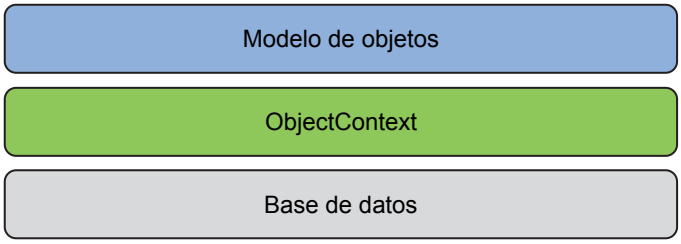
- Asistente para el EDM (*Entity Data Model*). Permite generar un EDM a partir de una base de datos existente, agregar a la aplicación información de conexión a la base de datos y generar las clases (que denominaremos clases de entidad) en el lenguaje de programación utilizado basadas en el modelo conceptual. Cuando este asistente termina de generar un EDM, inicia el diseñador de entidades.

- Diseñador de entidades (*Entity Designer*). Este diseñador proporciona una superficie de diseño que permite crear y modificar visualmente entidades, asociaciones, asignaciones y relaciones de herencia, así como validar o actualizar un EDM.

  También, a partir de VS 2010, se pueden añadir propiedades de navegación o complejas, o usar plantillas *T4* (*Text Template Transformation Toolkit*) para personalizar la generación de código. *EF* utiliza *T4* no sólo para la generación de código, sino para realizar un diseño *El esquema primero* (*Schema First*); esto es, definir primero nuestro modelo conceptual para luego lanzar desde el menú contextual del diseñador un asistente que genere el modelo lógico.

- Asistente para actualizar el modelo desde la base de datos. Este asistente permite actualizar un EDM cuando se realizan cambios en la base de datos subyacente. Este asistente se inicia desde el menú contextual de *Entity Designer*.

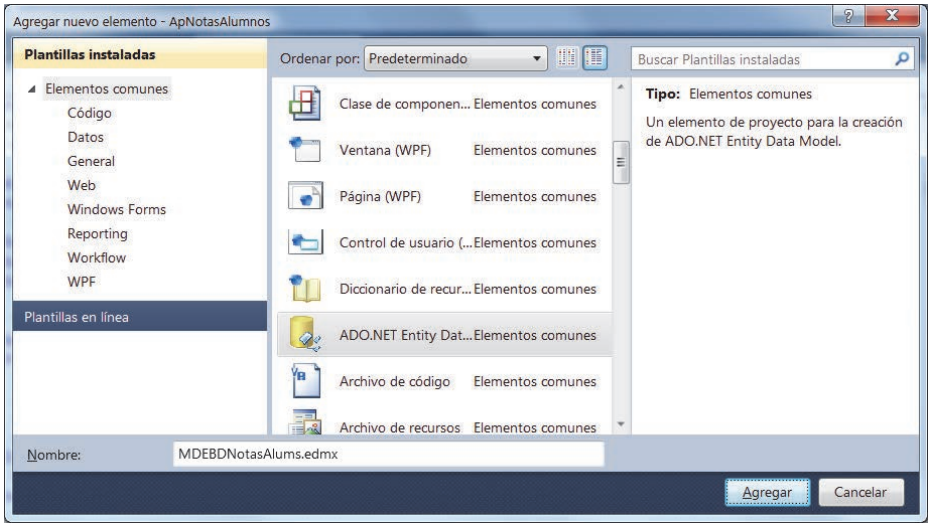
El *asistente para el EDM* se inicia a partir de la plantilla *ADO.NET Entity Data Model* de Visual Studio y almacena el EDM generado en un fichero con extensión *.edmx*. Después, el generador de código de *EF* creará a partir del contenido CSDL (almacenado en el fichero *.edmx*) el código correspondiente a las clases del modelo de objetos representativo de la base de datos virtual con la que vamos a trabajar. También, generará una clase derivada de **ObjectContext** que facilita las operaciones de consultar y trabajar con las entidades como objetos. Todo esto constituye lo que hemos denominado en el capítulo *Acceso a una base de datos* capa de acceso a datos (DAL).



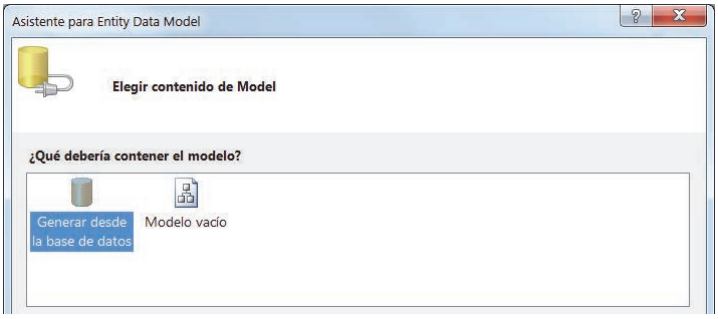
Según lo estudiado hasta ahora, un enfoque adecuado en el desarrollo de aplicaciones es separar la capa de acceso a datos (DAL) de la capa de presentación. Según estudiamos en el capítulo *Acceso a una base de datos*, la DAL típicamente se implementa como un proyecto *Biblioteca de clases*; entonces, como siguiente paso, añada un nuevo proyecto *Biblioteca de clases* a la solución *ApEntidades*, por ejemplo *BDNotasAlumnos*, y a continuación agregue a este proyecto un EDM denominado *MDEBDNotasAlums*.

Según lo expuesto anteriormente, para abrir el *asistente para el EDM* agregue un nuevo elemento de tipo *ADO.NET Entity Data Model* al proyecto *BDNotas-*

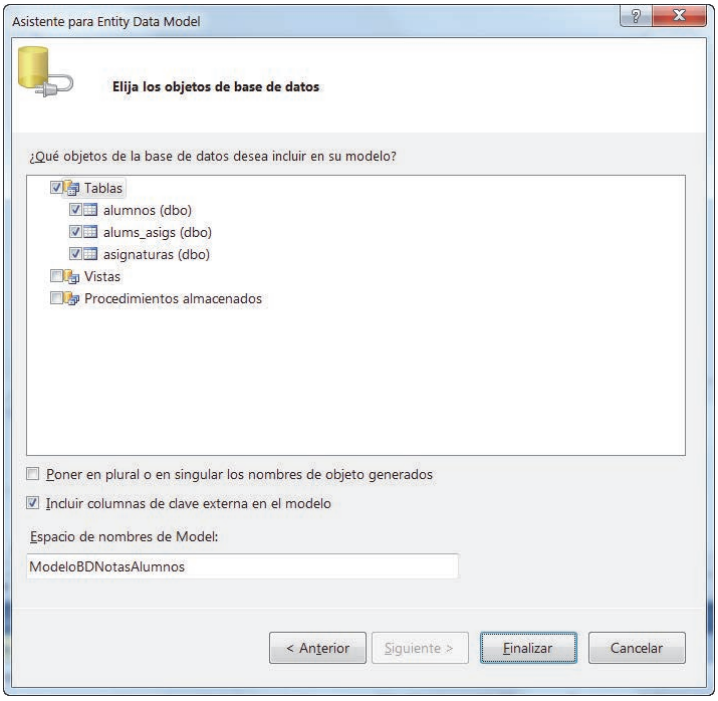
Alumnos. En nuestro caso denominaremos al EDM que se generará *MDEBDNotasAlums.edmx*.



Después de hacer clic en el botón *Agregar*, seremos preguntados por el contenido del modelo y si lo generamos desde una base de datos:

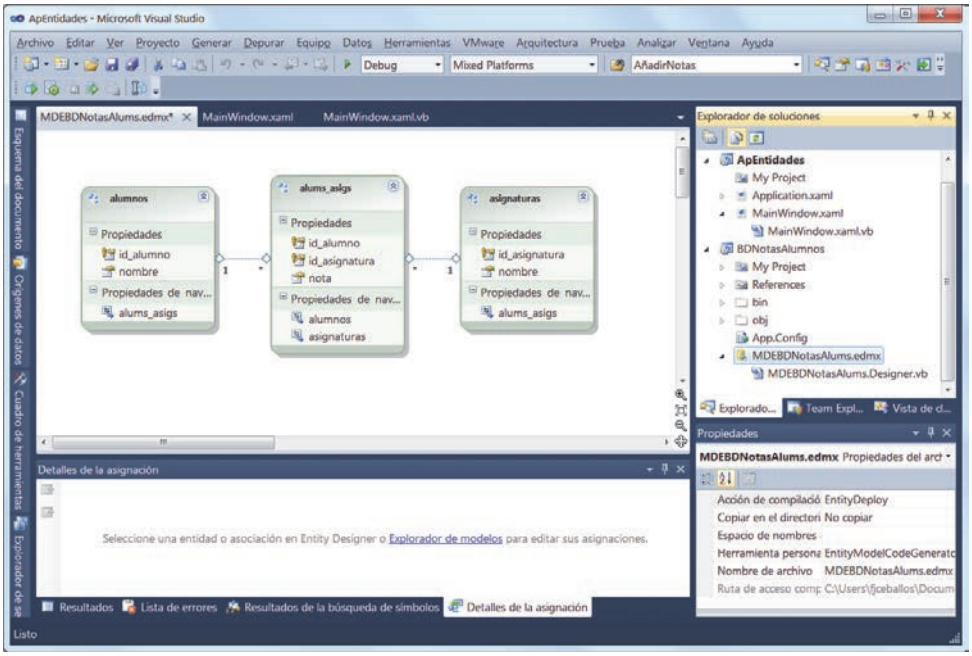


Seleccionamos la opción *Generar desde una base de datos* y hacemos clic en *Siguiente*, elegimos la conexión de datos que debe utilizar la aplicación para conectarse a la base de datos, aceptamos guardar la configuración de la cadena de conexión de *Entity* en el fichero de configuración *App.Config* con el nombre especificado (esta cadena de conexión tiene una sintaxis diferente a la cadena de conexión utilizada para conectarnos desde ADO.NET a una base de datos del servidor SQL Server) y, en el paso siguiente, elegimos los elementos de la base de datos que deberá contener el modelo, en nuestro caso las tablas *alumnos*, *alums_asigs* y *asignaturas*:



Observe que podríamos haber elegido la opción *Poner en plural o en singular los nombres de objeto generados*. Para entender lo que esta opción aporta, por primera vez, haremos esta operación manualmente un poco más adelante.

A continuación, especificamos el espacio de nombres del modelo y hacemos clic en *Finalizar*. La ventana de diseño de Visual Studio mostrará el *diseñador de entidades* que muestra la figura siguiente:



Si ahora hace clic con el botón secundario del ratón sobre el diseñador de entidades y observa el menú contextual, podrá ver todas las operaciones que se pueden realizar sobre el EDM y que anteriormente ya citamos.

Al crear el modelo de entidades (fichero *MDEBDNotasAlums.edmx*) y generar las entidades (fichero *MDEBDNotasAlums.Designer.vb*), se crearon automáticamente las asociaciones entre ellas basándose en las relaciones de clave externa existentes en la base de datos. No obstante, también puede realizar esta asociación haciendo clic sobre la clase de entidad y seleccionando del menú contextual la opción *Agregar Asociación*.

Entity Data Model utiliza tres conceptos clave para describir la estructura de datos: *tipo de entidad*, *tipo de asociación* y *propiedad*. El tipo de entidad es la unidad fundamental para describir la estructura de datos en un modelo conceptual y se construye a partir de las propiedades; las entidades describen la estructura de conceptos de nivel superior, como *alumnos* y *asignaturas*; las propiedades definen sus características; así, por ejemplo, el tipo de entidad *alumnos* contiene propiedades como *id_alumno* o *nombre*. Una instancia de un tipo de entidad representa un objeto específico (como un alumno o asignatura). Cada entidad debe tener una clave de entidad única dentro de un *conjunto de entidades*: colección de instancias de un tipo de entidad concreto. Según lo expuesto la relación entre un tipo de entidad y un conjunto de entidades es análoga a la relación entre una fila y una tabla en una base de datos relacional: al igual que una fila, un tipo de entidad describe la estructura de los datos y, al igual que una tabla, un conjunto de

entidades contiene instancias de un determinado tipo de entidad. Según esto, parece más lógico nombrar las entidades en singular y los conjuntos de entidades en plural, en vez de como las ha nombrado el asistente que generó el EDM.

El diagrama anterior muestra un modelo conceptual con tres tipos de entidad: *alumnos*, *alums_asigs* y *asignaturas*. Para hacer que se llamen *alumno*, *alum_asig* y *asignatura*, seleccione cada una de ellas y modifique el nombre, bien desde la ventana de propiedades (propiedad *Nombre*), o bien desde el propio diseñador, según muestra la figura siguiente:



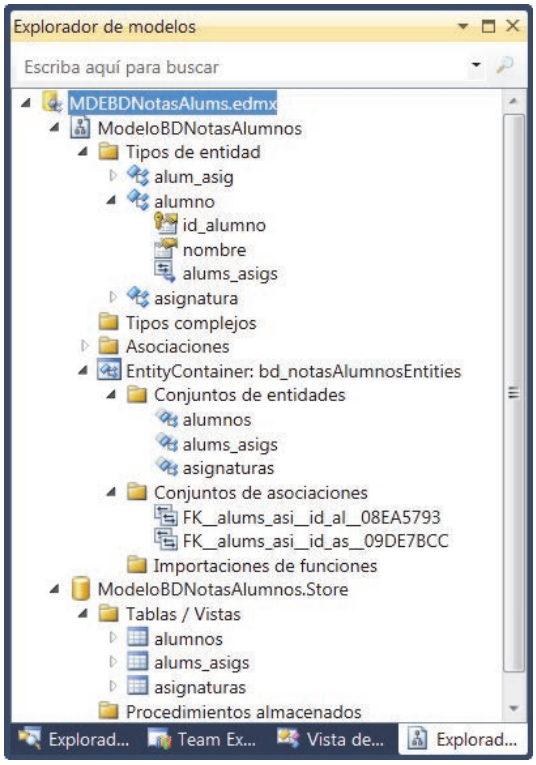
Para la descripción de las relaciones entre tipos de identidades, la unidad fundamental es el *tipo de asociación* o simplemente *asociación*. En un modelo conceptual, una asociación representa una relación entre dos tipos de entidad (como *alumno* y *alum_asig*) especificados por sus dos extremos, cada uno de los cuales especifica también una multiplicidad que indica el número de entidades que pueden estar en ese extremo de la asociación: una (1), cero o una (0..1) o muchas (*). De aquí las propiedades de navegación, como *alums_asigs*, utilizadas para acceder a las entidades situadas en un extremo de una asociación. Según esto, si en un extremo el número de identidades es una, parece más lógico que la propiedad de navegación correspondiente se nombre en singular (una entidad) y en plural, cuando sean muchas (una colección de entidades). Modificamos entonces en este sentido las propiedades de navegación de la entidad *alum_asig*.

Después de todas las modificaciones, el diagrama de entidades queda así:

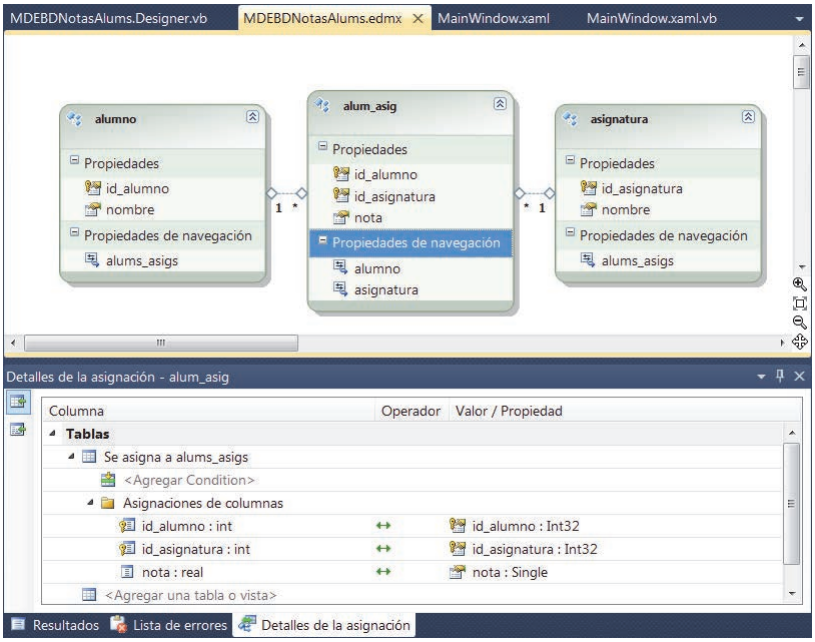


De forma predeterminada, *EF* crea la lógica para actualizar una base de datos (inserciones, actualizaciones y eliminaciones) con los cambios realizados en los datos de las clases de entidad, basándose en el esquema de la tabla (información de columna y de clave principal). Cuando no se desea usar el comportamiento predeterminado, se puede configurar el comportamiento asignando procedimientos almacenados concretos a través del diseñador.

Así mismo, desde el menú contextual del diseñador de entidades podemos mostrar también el *Explorador de modelos* que muestra la figura siguiente, desde el que podemos observar el modelo conceptual y el modelo lógico o de almacenamiento:



También, ejecutando la orden *Ver > Otras ventanas > Detalles de la asignación de Entity Data Model* podrá comprobar los detalles de la asignación C-S para la entidad que seleccione. Si, a continuación, hace clic en otra entidad, la vista de detalles cambiará para mostrar los de esta otra entidad:



Una vez finalizadas las tareas descritas, ya tenemos generado el modelo de objetos y el contexto de objetos (**ObjectContext**), que en nuestro caso está representado por la clase *bd_notasAlumnosEntities*. Se trata de una clase derivada de **ObjectContext** que representa el *contenedor de entidades*(agrupación lógica de los conjuntos de entidades y de los conjuntos de asociaciones) definido en el modelo conceptual. Desde el panel *Vista de clases* puede observar que dicha clase proporciona la funcionalidad necesaria para realizar consultas sobre un modelo EDM que devuelve entidades como objetos, así como para crear, actualizar y eliminar objetos de entidad.

Finalmente, para poder utilizar este modelo de entidades en el proyecto *ApEntidades*, habrá que añadir al mismo una referencia (nodo *References*) a la biblioteca *BDNotasAlumnos*. Otras cosas que también tenemos que hacer son copiar en el proyecto *ApEntidades* el fichero de configuración *App.Config* del proyecto *BDNotasAlumnos*, ya que este fichero contiene la cadena de conexión para acceder a la base de datos, y añadir una referencia a **System.Data.Entity**.

Las clases de entidad y el contexto de objetos

En el explorador de modelos, véase la figura en la página anterior, observamos que se ha generado una clase de entidad por cada entidad del modelo conceptual y otra clase *bd_notasAlumnosEntities* derivada de **ObjectContext**. Las clases de en-

tidad constituyen el modelo de objetos y la otra, el contexto de objetos que provee la funcionalidad para consultar y trabajar con las entidades como objetos.

A su vez, cada clase de entidad define una propiedad por cada columna de la tabla asociada, con el mismo nombre, así como otras propiedades, denominadas propiedades de navegación, para definir las relaciones que existen entre las tablas (éstas devuelven un **EntityCollection(Of** *TEntidad***)** para las relaciones 1..* o un **EntityReference(Of** *TEntidad***)** para las relaciones *..1). Todas las clases han sido definidas en el fichero *MDEBDNotasAlums.Designer.vb*; écheles una ojeada.

La clase *bd_notasAlumnosEntities* derivada de **ObjectContext** contiene la información de la *cadena de conexión*, la cual es proporcionada a través de su constructor, varias propiedades, por ejemplo, *alumnos*, y varios métodos, por ejemplo, el método **SaveChanges** que envía los datos actualizados del modelo de objetos de *EF* a la base de datos, a los que se puede llamar para realizar las operaciones deseadas sobre la base de datos. Fíjese también en las propiedades *alumnos*, *asignaturas* o *alums_asigs* de esta clase, retornan un objeto de tipo **ObjectSet(Of** *TEntidad***)** que representa un conjunto de entidades de tipo *TEntidad* (o dicho de otra forma, una secuencia de objetos "filas de la tabla") utilizado para realizar operaciones de creación, lectura, actualización y eliminación (**ObjectSet** se deriva de **ObjectQuery** que representa una consulta). Como la colección de objetos implementa la interfaz **IQueryable** que hereda de **IEnumerable**, podremos aplicar LINQ sin problemas. Por ejemplo, la propiedad *alumnos* devuelve un objeto **ObjectSet(Of** *alumno***)** que se corresponde con una secuencia de objetos de la clase de entidad *alumno*, cada uno de los cuales se corresponde con una fila de la tabla *alumnos*.

Según lo expuesto, para que una aplicación dotada de un contexto de objetos, como lo es *bd_notasAlumnosEntities*, pueda acceder a la base de datos correspondiente, deberá crear un objeto **ObjectContext**. Por ejemplo:

```
' Obtener el contexto de objetos
Dim contextoDeObjs As New bd_notasAlumnosEntities()
' Obtener la tabla alumnos
Dim TablaAlumnos As ObjectQuery(Of alumno) = _
  contextoDeObjs.alumnos
' Realizar una consulta
Dim alums = From alum In TablaAlumnos _
    Where alum.id_alumno = 1234567 _
    Select New With {.Nombre = alum.nombre}
If alums.Count() > 0 Then
  Console.WriteLine(alums.First().Nombre)
End If
```

¿Cuánto tiempo permanece abierta la conexión con la base de datos? Normalmente, una conexión permanece abierta mientras se utilizan los resultados de la consulta. Por ejemplo:

```
Dim alums = From alum In contextoDeObjs.alumnos  _
    Select alum
Console.WriteLine(contextoDeObjs.Connection.State.ToString()) ' escribe: Closed
For Each alum In alums
  ' ...
  Console.WriteLine(contextoDeObjs.Connection.State.ToString()) ' escribe: Open
Next
Console.WriteLine(contextoDeObjs.Connection.State.ToString()) ' escribe: Closed
```

Ahora bien, si espera que los resultados van a tardar bastante tiempo en procesarse, y no le causa ningún problema que se almacenen en memoria caché, aplique el método **ToList** a la consulta:

```
For Each alum In alums.ToList()
  ' ...
  Console.WriteLine(contextoDeObjs.Connection.State.ToString()) ' escribe: Closed
Next
```

Propiedades de navegación

Al importar un modelo relacional, el asistente para el EDM deduce de los metadatos de la base de datos las características de las asociaciones entre entidades y genera las correspondientes propiedades de navegación. Por lo tanto, una propiedad de navegación da acceso a las entidades que hay en los extremos de la asociación que representan. Por ejemplo, en una asociación entre las entidades *alumno* y *alum_asig*, la entidad *alumno* puede declarar una propiedad de navegación denominada *alums_asigs* para representar los objetos *alum_asig* asociados a un objeto *alumno* determinado. Resumiendo, desde un objeto *alumno* o *alum_asig*, las propiedades de navegación permiten localizar el objeto en el otro extremo de la asociación, lo cual simplifica enormemente las consultas en *eSQL*.

En el código del fichero *MDEBDNotasAlums.Designer.vb* podemos observar la clase de entidad *alumno* que está asociada a la tabla *alumnos* de la base de datos. Esta clase, que representa una fila de la tabla, define por cada columna de la tabla *alumnos* una propiedad del mismo nombre, *id_alumno* y *nombre*, que devuelve o asigna el valor correspondiente al atributo vinculado con la misma y una propiedad más, *alums_asigs* (llamada propiedad de navegación), que define la relación entre las tablas, en este caso, entre *alumnos* y *alums_asigs* (en la primera *id_alumno* es la clave primaria que, a su vez, es la clave externa en la segunda), que devuelve un objeto **EntityCollection(Of** *alum_asig***)** que se corresponde con una secuencia de objetos *alum_asig*. El ejemplo siguiente muestra cómo, gracias a

esta propiedad de navegación, es posible acceder a la notas de las asignaturas de un alumno de una forma muy simple:

```
Dim actas = _
    From alum In contextoDeObjs.alumnos _
    From al_as In alum.alums_asigs _
    Group New With {alum.nombre, al_as.nota} _
    By al_as.id_asignatura Into gr = Group

For Each grupo In actas
  Console.WriteLine(vbLf & "Asignatura: " & grupo.id_asignatura)
  For Each al In grupo.gr
    Console.WriteLine((al.nombre & " ") & al.nota)
  Next
Next
```

Este ejemplo muestra por cada asignatura los alumnos matriculados en la misma con sus notas correspondientes.

Obsérvese que la expresión *alum.alums_asigs* devuelve un objeto **EntityCollection(Of** *alum_asig***)** con las asignaturas y notas del alumno *alum*.

La propiedad *alums_asigs* que define la relación entre las tablas *alumnos* y *alums_asigs* nos permite para un objeto *alumno* acceder a los datos que figuran de este objeto en la tabla relacionada *alums_asigs* (relación 1:N).

Obsérvese la simplificación de la expresión de consulta (asignaturas en las que se ha matriculado *alum*) *from al_as in alum.alums_asigs* respecto a esta otra, que da el mismo resultado:

```
Join al_as In contextoDeObjs.alums_asigs _
           On alum.id_alumno Equals al_as.id_alumno _
```

Un análisis análogo lo haríamos con respecto a la clase de entidad *asignatura*. Y, ¿qué sucede con la clase de entidad *alum_asig*? Observamos que esta clase, que está asociada a la tabla *alums_asigs* de la base de datos, también define una propiedad por cada campo de la tabla y, además, dos propiedades más: *alumno* y *asignatura*, que definen las relaciones de la tabla *alums_asigs* con las tablas *alumnos* y *asignaturas* (en la primera *id_alumno* e *id_asignatura* son claves externas y, respectivamente, en las segundas son claves primarias); estas dos propiedades devuelven un objeto *TEntidad* (*alumno* o *asignatura*), a través de una referencia de tipo **EntityReference(Of** *TEntidad***)**, que se corresponde con la entidad del extremo de la relación con multiplicidad 1 en este caso. El ejemplo siguiente muestra cómo, gracias a estas dos propiedades, es posible acceder a las asignaturas de las que se ha matriculado un alumno de una forma muy simple:

```
Dim idAlumno As Integer = 1234567

Dim alumAsigs = _
    From al_as In contextoDeObjs.alums_asigs _
    Where al_as.id_alumno = idAlumno _
    Group New With {.NombAsig = al_as.asignatura.nombre} _
    By al_as.alumno.nombre Into gr = Group

For Each grupo In alumAsigs
  Console.WriteLine(vbLf & "Alumno: " & grupo.nombre)
  For Each x In grupo.gr
    Console.WriteLine(x.NombAsig)
  Next
Next
```

Las expresiones *al_as.asignatura* y *al_as.alumno* haciendo uso de **EntityReference** devuelven, respectivamente, un objeto *asignatura* y otro *alumno*.

Las propiedades *alumno* y *asignatura* que definen las relaciones entre la tabla *alums_asigs* y las tablas *alumnos* y *asignaturas* nos permiten para un objeto *alum_asig* acceder a los datos que figuran de este objeto en la tabla relacionada *alumnos* o *asignaturas* (relación N:1).

Otra solución al ejemplo anterior puede ser la siguiente:

```
Dim idAlumno As Integer = 1234567

Dim alums = _
  From al In contextoDeObjs.alumnos _
  Where al.id_alumno = idAlumno _
  Select al
Dim asigs = _
  From al_as In contextoDeObjs.alums_asigs _
  Where al_as.id_alumno = idAlumno _
  Select al_as.asignatura

For Each alum In alums
  Console.WriteLine(alum.nombre)
Next
For Each asig In asigs
  Console.WriteLine(asig.nombre)
Next
```

Mostrar datos en una interfaz gráfica

Los datos del ejemplo anterior pueden ser fácilmente mostrados en una interfaz gráfica como la siguiente:

```
<Grid>
  <StackPanel>
    <TextBlock Name="tbNomAlum" />
    <ListBox Name="listaAsigs"/>
  </StackPanel>
</Grid>
```

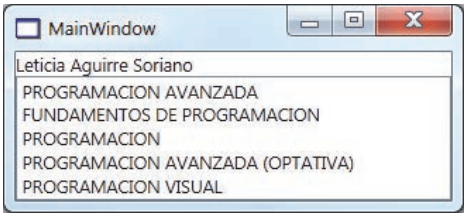
Los datos a mostrar, según el ejemplo del apartado anterior, provienen de: el nombre del alumno de la colección *alums* y las asignaturas de las que está matriculado, de la colección *asigs*. Por lo tanto, para mostrar el nombre en el **TextBlock** bastaría escribir:

```
tbNomAlum.DataContext = alums
tbNomAlum.Text = alums.First().nombre
```

Y para mostrar la lista de asignaturas en el **ListBox** bastaría escribir:

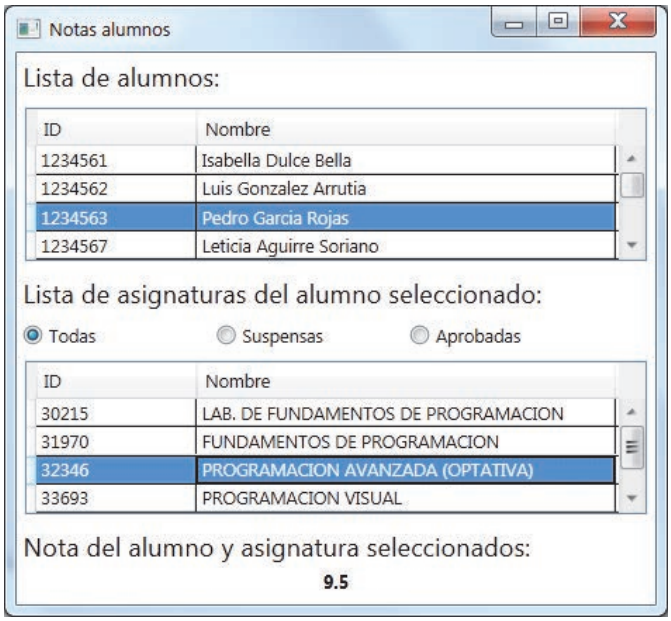
```
listaAsigs.ItemsSource = asigs
listaAsigs.DisplayMemberPath = "nombre"
```

El resultado sería el siguiente:



Una aplicación con interfaz gráfica

El paso siguiente es ver cómo trabaja una aplicación con interfaz gráfica contra la base de datos utilizando este modelo de objetos. Como ejemplo, vamos a generar la aplicación "Notas alumnos" que ya conoce del capítulo *Acceso a una base de datos* y así podrá comparar las tecnologías utilizadas en una y otra versión. La aplicación, inicialmente, mostrará una interfaz que permita a un alumno acceder a la base de datos para conocer la nota obtenida en cualquiera de las asignaturas en las que está matriculado.



A la vista de la interfaz anterior, el alumno seleccionará su nombre en la primera lista (este control muestra los nombres de la tabla *alumnos*), lo que hará que se muestren en la otra lista las asignaturas de las que está matriculado. Después, seleccionará la asignatura de la cual quiere conocer la nota y ésta le será mostrada en un control de texto.

Cree una nueva solución, *ApNotasAlumnos*, con un proyecto del mismo nombre. A continuación añada la biblioteca de clases *BDNotasAlumnos* que generamos anteriormente. Después, para poder utilizar este modelo de entidades en el proyecto *ApNotasAlumnos*, añada al mismo una referencia a la biblioteca de clases *BDNotasAlumnos*. También hay que copiar en el proyecto *ApNotasAlumnos* el fichero de configuración *App.Config* del proyecto *BDNotasAlumnos*, que contiene la cadena de conexión para acceder a la base de datos, y añadir una referencia a **System.Data.Entity**.

A continuación, construya la interfaz gráfica, tal como explicamos en el capítulo *Acceso a una base de datos*, y añada los manejadores de datos que allí utilizamos. Omitimos mostrar este código porque, excepto el cuerpo de los manejadores, es el mismo que expusimos en el capítulo *Acceso a una base de datos* (en el CD del libro puede verlo).

Vincular controles con el origen de datos

Ya tenemos generado el modelo de objetos representación de la base de datos, esto es, la capa de acceso a datos (DAL), y diseñada la interfaz gráfica. Ahora, partiendo de este modelo, crearemos un proveedor de datos para utilizarlo como origen de datos en la primera lista, *dgAlumnos*, la que muestra los nombres de los alumnos. Para ello, añada una nueva carpeta al proyecto *ApNotasAlumnos*, denominada por ejemplo *BLL*, y agregue a la misma la siguiente clase:

```
Imports BDNotasAlumnos

Class CProveedorDatosAlumno
  Private contextoDeObjs As bd_notasAlumnosEntities

  Public Sub New()
    contextoDeObjs = New bd_notasAlumnosEntities()
  End Sub

  Public Function ObtenerAlumnos() As List(Of alumno)
    ' Consulta LINQ
    Dim alums = _
      From alum In contextoDeObjs.alumnos _
      Select alum
    Return alums.ToList()
  End Function
End Class
```

La variable *alums* hace referencia a un objeto que implementa la interfaz **IQueryable(Of** *alumno***)** y **ToList** devuelve una colección de tipo **List(Of** *alumno***)**.

Un objeto de la clase *CProveedorDatosAlumno* será el proveedor de datos del **DataGrid** *dgAlumnos*. Dicho proveedor lo construiremos como un recurso *pdAlumnos* de la ventana, según se indica a continuación:

```
<Window.Resources>
  <ObjectDataProvider x:Key="pdAlumnos"
    ObjectType="{x:Type local:CProveedorDatosAlumno}"
    MethodName="ObtenerAlumnos">
  </ObjectDataProvider>
</Window.Resources>
```

Observe que el recurso *pdAlumnos* proporciona la colección de alumnos obtenida de la tabla *alumnos* por medio de una consulta LINQ que realiza el método *ObtenerAlumnos* de la clase *CProveedorDatosAlumno* que da lugar al proveedor. Para que el **DataGrid** *dgAlumnos* muestre los datos *id_alumno* y *nombre* de cada uno de los objetos *alumno* de la colección, asignamos dicho proveedor al contexto

de datos del contenedor del **DataGrid**, para así poder vincular dicha colección con la propiedad **ItemsSource** del **DataGrid**, y los atributos *id_alumno* y *nombre* de cada objeto de la colección con sus respectivas columnas en el **DataGrid**:

```
<StackPanel DataContext="{StaticResource pdAlumnos}">
  <TextBlock FontSize="16" Margin="4" Text="Lista de alumnos:"/>
  <DataGrid Name="dgAlumnos" Margin="5" MaxHeight="95"
            AutoGenerateColumns="False"
            ItemsSource="{Binding}"
            SelectionChanged="dgAlumnos_SelectionChanged">
    <DataGrid.Columns>
      <DataGridTextColumn Header="ID"
            Binding="{Binding Path=id_alumno}" MinWidth="100"/>
      <DataGridTextColumn Header="Nombre"
            Binding="{Binding Path=nombre}" MinWidth="260"/>
    </DataGrid.Columns>
  </DataGrid>
</StackPanel>
```

Si ejecuta ahora la aplicación, observará cómo la ventana muestra ya la lista de alumnos.

Una alternativa al proveedor de datos anterior sería crear un objeto **CollectionViewSource**. Recuerde, no es recomendable crear objetos de la clase **CollectionView**, sino que para crear una vista de colección para una colección que sólo implementa **IEnumerable**, lo recomendado es crear un objeto **CollectionViewSource**, agregar la colección a su propiedad **Source** y obtener la vista de colección de su propiedad **View** (véase el capítulo *Enlace de datos y WPF*). Trabajar con una vista de colección facilita agregar y quitar elementos en lugar de trabajar directamente con la colección de origen, y desacopla el origen de datos (y por lo tanto la lógica de la colección) de la interfaz gráfica, por lo que será mucho más fácil cambiar posteriormente las fuentes. Por ejemplo, podemos crear el objeto **CollectionViewSource** como un recurso de la aplicación, así:

```
<Window.Resources>
  <CollectionViewSource x:Key="pdAlumnos" />
</Window.Resources>
```

Una vez definido el recurso de tipo **CollectionViewSource**, asignamos a su propiedad **Source** la colección de objetos *alumno*:

```
Public Class MainWindow
  Private proDatos As New CProveedorDatosAlumno()

  Public Sub New()
    ' ...
```

```
    Dim alums = DirectCast(Me.Resources("pdAlumnos"),
                           CollectionViewSource)
    alums.Source = proDatos.ObtenerAlumnos()
  End Sub
  ' ...
End Class
```

Continuemos con la aplicación. Cuando el usuario seleccione un nombre en la primera lista, en la segunda lista deben mostrarse las asignaturas de la tabla *asignaturas* de las que ese alumno se ha matriculado. ¿Cómo obtenemos esas asignaturas y las notas correspondientes? Pues a través de la siguiente expresión de consulta:

```
Dim alumAsigs = _
  From al_as In contextoDeObjs.alums_asigs _
  Where al_as.id_alumno = idAlumno _
  Select New With {al_as.asignatura.id_asignatura, _
                   al_as.asignatura.nombre, _
                   al_as.nota}
```

Observe que en la consulta hay un parámetro variable que se corresponde con el identificador *idAlumno* del alumno seleccionado y que será obtenido, como veremos un poco más adelante, de la fila seleccionada en el **DataGrid** *dgAlumnos*. Cada objeto de la colección *alumAsigs* es de un tipo anónimo con tres atributos: *id_asignatura*, *nombre* y *nota*.

Construyamos entonces otro proveedor de datos que nos proporcione la colección de objetos producidos por esta consulta. Para ello, añada a la carpeta *BLL* del proyecto una nueva clase *CProveedorAsignaturasNotas* que implemente un método *ObtenerAsignaturasNotas* que devuelva la colección de objetos:

```
Imports BDNotasAlumnos

Class CProveedorAsignaturasNotas
  Private contextoDeObjs As bd_notasAlumnosEntities

  Public Sub New()
    contextoDeObjs = New bd_notasAlumnosEntities()
  End Sub

  Public Function ObtenerAsignaturasNotas(idAlumno As Integer) As Object
    Dim alumAsigs = _
      From al_as In contextoDeObjs.alums_asigs _
      Where al_as.id_alumno = idAlumno _
      Select New With {al_as.asignatura.id_asignatura, _
                       al_as.asignatura.nombre, _
                       al_as.nota}
    Return alumAsigs.ToList()
```

```
  End Function
End Class
```

En ocasiones, nos puede interesar colocar las entidades en un **ObservableCollection** por las ventajas que esto reporta, según estudiamos en el capítulo *Enlace de datos y WPF*. Por ejemplo, para que el método anterior devolviera un objeto de esa clase, simplemente tendríamos que invocar al constructor **ObservableCollection** pasando como argumento la colección *alumAsigs*. El objeto así construido contendría una copia de los elementos de la colección especificada.

```
Public Function ObtenerAsignaturasNotas(idAlumno As Integer) _
As ObservableCollection(Of Object)
  Dim alumAsigs = _
    From al_as In contextoDeObjs.alums_asigs _
    Where al_as.id_alumno = idAlumno
    Select New With {al_as.asignatura.id_asignatura, _
                     al_as.asignatura.nombre, _
                     al_as.nota}
  Dim asigs As New ObservableCollection(Of Object)(alumAsigs)
  Return asigs
End Function
```

La colección de asignaturas proporcionada por el proveedor *CProveedorAsignaturasNotas* corresponde al alumno seleccionado y será el origen de datos para el segundo **DataGrid**, *dgAsignaturas*, de la interfaz gráfica:

```
<StackPanel Name="spAsignaturas" Grid.Row="1">
  <TextBlock FontSize="16" Margin="4"
    Text="Lista de asignaturas del alumno seleccionado:"/>
  <DataGrid Name="dgAsignaturas" Margin="5" MaxHeight="95"
            AutoGenerateColumns="False"
            ItemsSource="{Binding}" CanUserAddRows="False">
    <DataGrid.Columns>
      <DataGridTextColumn Header="ID"
        Binding="{Binding Path=id_asignatura}" MinWidth="100"/>
      <DataGridTextColumn Header="Nombre"
        Binding="{Binding Path=nombre}" MinWidth="260"/>
    </DataGrid.Columns>
  </DataGrid>
</StackPanel>
```

Para que el **DataGrid** *dgAsignaturas* muestre los datos *id_asignatura* y *nombre* de cada uno de los objetos de la colección devuelta por el método *ObtenerAsignaturasNotas* del proveedor *CProveedorAsignaturasNotas*, tenemos que asignar dicha colección al contexto de datos del contenedor del **DataGrid**, *spAsignaturas*, y vincular dicha colección con la propiedad **ItemsSource** del **Data-**

Grid, y los atributos *id_ asignatura* y *nombre* de cada objeto de la colección con sus respectivas columnas en el **DataGrid**.

Ahora bien, la colección a la que nos hemos referido en el párrafo anterior cambia para cada alumno seleccionado. Según esto, ¿cuándo asignamos la colección al contexto de datos de *spAsignaturas*? Pues cada vez que el usuario seleccione un nuevo nombre en el **DataGrid** *dgAlumnos*. ¿Y dónde hacemos esa asignación? Como la acción "seleccionar una nueva fila" genera el evento **SelectedChanged**, será el método que responda a este evento el que genere la colección de asignaturas correspondiente al alumno seleccionado y el que la asigne al contexto de datos del **DataGrid**. Por lo tanto, añada este método y complételo así:

```
Private Sub dgAlumnos_SelectionChanged(sender As System.Object, e
As System.Windows.Controls.SelectionChangedEventArgs)
  If e.AddedItems.Count = 0 Then Return
  Dim alum As alumno = TryCast(e.AddedItems(0), alumno)
  If alum Is Nothing Then Return
  Dim pAsigsNotas As New CProveedorAsignaturasNotas()
  asigs = pAsigsNotas.ObtenerAsignaturasNotas(alum.id_alumno)
  spAsignaturas.DataContext = asigs
End Sub
```

El código anterior requiere declarar el siguiente atributo en la clase *MainWindow* a la que pertenece el método:

```
Private asigs As Object
```

Ahora el **DataGrid** *dgAsignaturas* ya muestra el nombre de las asignaturas en las que se ha matriculado el alumno seleccionado y, además, tenemos acceso también a las notas correspondientes; recuerde que cada objeto de la colección *alumAsigs* es de un tipo anónimo con tres atributos: *id_asignatura*, *nombre* y *nota*. Pues bien, ahora cuando el usuario seleccione una de las asignaturas mostradas, se tiene que visualizar la nota correspondiente a esa asignatura en el **TextBlock** *tbNota* que hemos colocado en el fondo de la ventana. La acción de seleccionar un elemento en el **DataGrid** hace que dicho elemento quede referenciado por su propiedad **SelectedItem**. Entonces, esta propiedad será el proveedor de datos para *tbNota*, según muestra el código escrito a continuación; esto es, asignamos al contexto de datos del contenedor del **TextBlock** la propiedad **SelectedItem** de *dgAsignaturas* y vinculamos la propiedad **Text** de *tbNota* con el atributo *nota* del elemento correspondiente a la asignatura seleccionada:

```
<StackPanel Grid.Row="2"
  DataContext="{Binding SelectedItem, ElementName=dgAsignaturas}">
  <TextBlock FontSize="16" Margin="4"
    Text="Nota del alumno y asignatura seleccionados:"/>
  <TextBlock Name="tbNota"
```

```
    Text="{Binding Path=nota}"
    HorizontalAlignment="Center" FontWeight="Bold" />
</StackPanel>
```

Filtros

Según estudiamos en el capítulo *Enlace de datos y WPF*, el método **GetDefaultView** de **CollectionViewSource** devuelve la vista predeterminada del origen especificado, vista que permite navegar, filtrar, ordenar y agrupar datos en una colección. Por ejemplo, la siguiente línea de código devuelve la vista correspondiente a la colección *asigs* correspondiente al alumno seleccionado:

```
Dim vista As ICollectionView = CollectionViewSource.GetDefaultView(asigs)
```

En otras palabras, una vista no es más que una representación de los datos de un objeto, que implementa la interfaz **ICollectionView**, que permite ordenación, filtrado, búsqueda, edición y navegación.

Por ejemplo, para mostrar las filas del **DataGrid** *dgAsignaturas* ordenadas por el nombre de la asignatura simplemente tendríamos que añadir al método *dgAlumnos_SelectionChanged* de nuestra solución el código indicado a continuación:

```
Dim vista As ICollectionView =
  CollectionViewSource.GetDefaultView(asigs)
If vista.SortDescriptions.Count = 0 Then
  vista.SortDescriptions.Add( _
    New SortDescription("nombre", ListSortDirection.Ascending))
End If
```

También podemos utilizar la vista para aplicar filtros personalizados para que sólo se muestren ciertas filas. Como ejemplo, vamos a añadir tres botones de opción a la ventana de nuestra aplicación para permitir al usuario mostrar: todas las asignaturas del alumno seleccionado, las asignaturas que tiene suspensas o las asignaturas que tiene aprobadas. Una vez añadidos los botones, controlamos el evento **Checked** de cada uno de ellos para establecer en la vista el filtro adecuado para que sólo se muestren las filas deseadas. Por ejemplo, para mostrar las asignaturas suspensas el filtro que tendríamos que aplicar a la vista sería:

```
If vista Is Nothing Then Return
vista.Filter = Function(elemento As Object) elemento.nota < 5
```

Y para mostrar las asignaturas aprobadas el filtro que tendríamos que aplicar a la vista sería:

```
If vista Is Nothing Then Return
vista.Filter = Function(elemento As Object) elemento.nota >= 5
```

Finalmente, para mostrar todas las asignaturas el filtro que tendríamos que aplicar a la vista sería:

```
If vista Is Nothing Then Return
vista.Filter = Nothing
```

Complete los controladores de los botones de opción con estos filtros y la aplicación está finalizada. La conclusión final que tenemos que sacar es que utilizando *Entity Framework* hemos sido capaces de acceder a una base de datos sin necesidad de tener que conocer los proveedores de datos de ADO.NET y el lenguaje de consultas SQL.

Contextos de corta duración

En la aplicación que hemos iniciado estamos usando un contexto de larga duración; en nuestro caso, al ser un atributo de la clase *MainWindow* durará el tiempo que la aplicación se esté ejecutando, lo que puede suponer un abuso de recursos difícil de soportar en muchas ocasiones. Por eso, puede ser conveniente utilizar contextos de corta duración, algo habitual, por ejemplo, en aplicaciones distribuidas. La forma más habitual de añadir un contexto de corta duración es mediante una sentencia **Using**. Por ejemplo:

```
Public Class MainWindow
  Public Sub New()
    ' ...
    Dim proDatos As New CProveedorDatosAlumno()
    Using proDatos
      Dim alums = DirectCast(Me.Resources("pdAlumnos"),
        CollectionViewSource)
      alums.Source = proDatos.ObtenerAlumnos()
    End Using
  End Sub
  ' ...
End Class
```

La sentencia **Using** utilizada en el código anterior obliga a que la clase *CProveedorDatosAlumno* implemente la interfaz **IDisposable**.

REALIZAR CAMBIOS EN LOS DATOS

Los cambios que se efectúen en los objetos que forman la base de datos orientada a objetos virtual, imagen del modelo de datos de una base de datos relacional, no

se aplican a esta última hasta que se llame explícitamente al método **SaveChanges** de **ObjectContext**.

Al llamar a **SaveChanges** se desencadena el siguiente proceso:

1. *EF* examina el conjunto de objetos para determinar si ha habido cambios (los servicios de objetos realizan el seguimiento de los cambios realizados en estos objetos). Si hubo cambios, los objetos con cambios que dependen de otros se ordenan según sus dependencias.

2. *EF* inicia una transacción para encapsular la serie de órdenes individuales.

3. Los cambios en los objetos se convierten uno a uno en sentencias SQL y se envían al servidor. En este punto, cualquier error detectado por el gestor de la base de datos hace que: se detenga el proceso de envío, se inicie una excepción y se reviertan todos los cambios de la base de datos, como si no se hubiese enviado nada. Además, como **ObjectContext** mantiene un registro completo de todos los cambios, se puede intentar corregir el problema y llamar de nuevo a **SaveChanges**. Para cada objeto y relación de un contexto determinado, *EF* crea una estructura de datos, objeto de la clase **ObjectStateEntry**, para almacenar su estado y hacer el seguimiento de los cambios. Este conjunto de estructuras está accesible a través de la propiedad **ObjectStateManager** del contexto de objetos.

El siguiente ejemplo muestra de una forma genérica cómo llevar a efecto los cambios sobre una base de datos. Primero se realiza la consulta y después se hacen los cambios (modificar, añadir o borrar) sobre los resultados obtenidos para, a continuación, enviar esos cambios a la base de datos. Cuando una consulta se ejecuta dentro de un contexto de objetos, los objetos devueltos se asocian automáticamente a dicho contexto de objetos.

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Consulta/nuevo objeto
  ' ...
  Try
    ' Realizar cambios sobre la consulta obtenida
    ' ...
    ' Cambios realizados:
    Dim osm As ObjectStateManager=contextoDeObjs.ObjectStateManager
    Dim sCambios As String = _
          "Modificados: " & osm.GetObjectStateEntries( _
            EntityState.Modified).Count().ToString() & _
          ", añadidos: " & osm.GetObjectStateEntries( _
            EntityState.Added).Count().ToString() & _
          " y borrados: " & osm.GetObjectStateEntries( _
            EntityState.Deleted).Count().ToString()
```

```
    Console.WriteLine(sCambios)

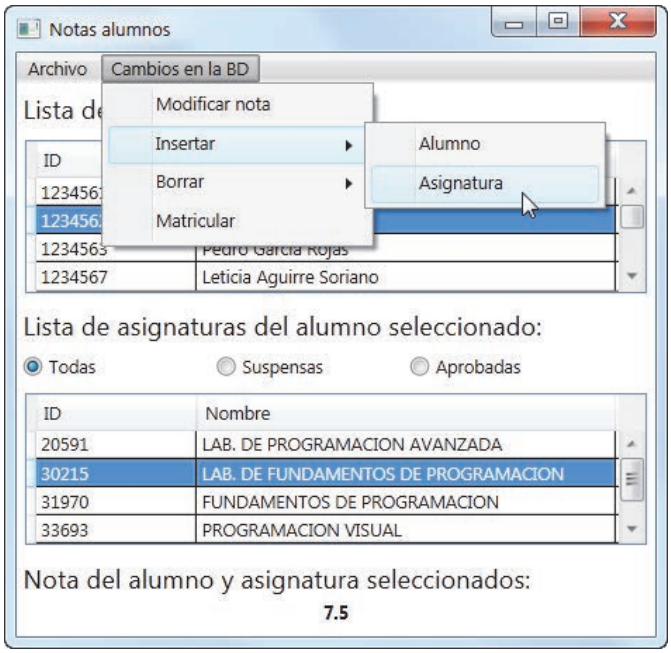
    ' Enviar los cambios a la base de datos
    Dim nCambios As Integer = contextoDeObjs.SaveChanges()
    Console.WriteLine("Se realizaron " & nCambios.ToString() & _
                      " cambios")
  Catch ex As OptimisticConcurrencyException
    MessageBox.Show(ex.Message)
    ' Alternativa: hacer ajustes e intentarlo otra vez.
    ' contextoDeObjs.SaveChanges();
    contextoDeObjs.Refresh(RefreshMode.ClientWins, asigNota)
    contextoDeObjs.SaveChanges()
  Catch ex As Exception
    MessageBox.Show(ex.Message)
  End Try
End Using
```

Cuando no se han realizado cambios, los resultados en *sCambios* serán de la siguiente manera: *Modificados: 0, añadidos: 0 y borrados: 0*, y *nCambios* (número de objetos que tenían el estado **Modified**, **Added** o **Deleted** cuando se llamó al método **SaveChanges**) valdrá cero.

El método **SaveChanges** sin argumentos ejecuta las sentencias adecuadas para intentar implementar los cambios en la base de datos. Si ocurre un error se procede como se ha indicado anteriormente en el punto 3. Hay otra versión, con un argumento (**True** o **False**). Un valor **True** indica que se restablezca el seguimiento de cambios en el contexto de objetos (es el valor por omisión) y un valor **False** obligaría a llamar al método **AcceptAllChanges** después de **SaveChanges**.

Antes de que se ejecute el método **SaveChanges**, se genera el evento **SavingChanges**, lo que permitirá implementar, cuando sea necesario, la lógica de negocio personalizada antes de que se guarden los cambios en la base de datos.

A continuación, vamos a exponer cómo se implementan las modificaciones de filas de una tabla, las inserciones de filas y el borrado de las mismas. Para probar dichas operaciones, vamos a añadir un menú a la aplicación análogo al siguiente:



El menú *Archivo* tendrá un solo elemento *Salir* y el menú *Cambios en la BD* tendrá cuatro elementos: *Modificar nota*, *Insertar*, *Borrar* y *Matricular*. A su vez, *Insertar* tendrá los elementos *Alumno* y *Asignatura* y *Borrar* también. En la carpeta *Cap07\Cambios en los datos* del CD que acompaña al libro puede ver el ejercicio completo.

Modificar filas en la base de datos

Puede actualizar las filas de una base de datos modificando los valores de los miembros de los objetos asociados a la colección de objetos *clase_entidad* que devuelve la consulta **ObjectQuery(Of** *clase_entidad***)** y después, enviando los cambios a la base de datos.

Para actualizar una fila de la base de datos, siga estos pasos:

1. Realice una consulta para obtener la fila a modificar.
2. Ejecute la consulta y cambie los valores de las columnas implicadas.
3. Envíe los cambios a la base de datos.

El siguiente ejemplo busca en la tabla *alums_asigs* la fila (objeto *alum_asig*) correspondiente al alumno y asignatura especificados y modifica la nota, salvando después los cambios en la base de datos. La variable *consulta* representa una consulta **ObjectQuery(Of** *alum_asig***)**. Evidentemente, en una aplicación real, los datos implicados en la operación (sección *Obtener los datos*) serán introducidos por

el usuario, por ejemplo, a través de una caja de diálogo según puede ver en el apartado de *Ejercicios resueltos*.

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234567
  Dim idAsig As Integer = 20590
  Dim notaAlum As Single = 7.5F

  ' Consulta para obtener la fila a modificar
  Dim consulta = _
    From al_as In contextoDeObjs.alums_asigs _
    Where al_as.id_alumno = idAlum AndAlso al_as.id_asignatura = idAsig _
    Select al_as

  If consulta.Count() = 0 Then
    MessageBox.Show("La consulta no contiene elementos")
    Return
  End If

  ' Ejecutar la consulta y cambiar los valores
  ' de las columnas implicadas
  For Each al_as As alum_asig In consulta
    al_as.nota = notaAlum
  Next

  ' Enviar los cambios a la base de datos
  Try
    contextoDeObjs.SaveChanges()
    MessageBox.Show("Cambios realizados")
  Catch ex As Exception
    MessageBox.Show(ex.InnerException.Message)
  End Try
End Using
```

Insertar filas en la base de datos

Para insertar filas en una base de datos, se agregan los objetos correspondientes a las mismas al conjunto de entidades especificado del contexto de objetos y después se envían los cambios a la base de datos.

Para insertar una fila en la base de datos siga estos pasos:

1. Cree un nuevo objeto que incluya los datos de esa fila. Para ello, puede utilizar el método *clase_entidad.Create...*, proporcionado por la clase de entidad correspondiente al tipo del objeto.

2. Agregue el nuevo objeto al contexto de objetos. Para ello, el contexto facilita métodos *AddTo...*, para agregar nuevos objetos de cada una de las entidades presentes en el EDM, los cuales, a su vez, invocan al método **AddObject** de **ObjectContext** que tiene como primer argumento el nombre del conjunto de entidades al que se va a añadir el objeto y como segundo el objeto de ese tipo de entidad a añadir. Así mismo, los nuevos objetos serán registrados en el **ObjectStateManager** con el estado *Added*.
3. Envíe el cambio a la base de datos.

El siguiente ejemplo crea un nuevo objeto de tipo *alumno*, se asignan los valores adecuados a sus propiedades y se agrega al conjunto de entidades *alumnos*. Finalmente, se envía el cambio a la base de datos.

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234560
  Dim nomAlum As String = "Alberto García Sánchez"

  ' Crear un nuevo objeto alumno
  Dim alum As alumno = alumno.Createalumno(idAlum, nomAlum)

  ' Añadirlo al conjunto de entidades alumnos
  contextoDeObjs.AddToalumnos(alum)
  ' O bien:
  'contextoDeObjs.AddObject("alumnos", alum);

  ' Enviar los cambios a la base de datos
  Try
    contextoDeObjs.SaveChanges()
    MessageBox.Show("Cambios realizados")
  Catch ex As Exception
    MessageBox.Show(ex.InnerException.Message)
  End Try
End Using
```

Como ejercicio, se propone escribir el código necesario para añadir una nueva asignatura.

Si la operación de inserción no tiene sentido, por ejemplo, porque intentamos insertar una fila en la tabla *alumnos* y el identificador del *alumno* ya existe, se elevará la excepción correspondiente.

Y, ¿cómo añadimos un objeto *alum_asig*? En este caso se trata de un nuevo objeto relacionado con otro objeto del contexto de objetos. Para añadirlo proceda de alguna de las dos formas siguientes:

- Para relaciones de uno a varios o de varios a varios, llame al método **Add** de **EntityCollection(Of** *TEntidad***)** y especifique el objeto relacionado.

- Para relaciones de uno a uno o de varios a uno, establezca la propiedad **Value** de la **EntityReference(Of** *TEntidad***)** en el objeto relacionado.

El siguiente ejemplo crea un nuevo objeto de tipo *alum_asig*, se asignan los valores adecuados, se asignan las relaciones N:1 de este objeto con los objetos *alumno* y *asignatura* respectivos y se agrega al conjunto de entidades *alums_asigs*. Finalmente, se envía el cambio a la base de datos.

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234560
  Dim idAsig As Integer = 34680

  ' Consulta para obtener el alumno a matricular
  Dim consulta1 = _
    From alum In contextoDeObjs.alumnos _
    Where alum.id_alumno = idAlum _
    Select alum

  ' Consulta para obtener la asignatura de la que se va a matricular
  Dim consulta2 = _
    From asig In contextoDeObjs.asignaturas _
    Where asig.id_asignatura = idAsig _
    Select asig

  If consulta1.Count() = 0 OrElse consulta2.Count() = 0 Then
    MessageBox.Show("El alumno o la asignatura no existen")
    Return
  End If
  Dim alumno As alumno = consulta1.First()
  Dim asignatura As asignatura = consulta2.First()

  ' Crear un nuevo objeto alums_asigs
  Dim al_as = alum_asig.Createalum_asig(idAlum, idAsig, 0.0F)

  ' Establecer las relaciones
  al_as.alumnoReference.Value = alumno ' N:1
  al_as.asignaturaReference.Value = asignatura ' N:1
  ' O bien:
  'alumno.alums_asigs.Add(al_as);     ' 1:N
  'asignatura.alums_asigs.Add(al_as); ' 1:N
  ' Añadir el nuevo objeto al conjunto de entidades alums_asigs
  contextoDeObjs.AddToalums_asigs(al_as)

  ' Enviar los cambios a la base de datos
  Try
```

```
    contextoDeObjs.SaveChanges()
    MessageBox.Show("Cambios realizados")
  Catch ex As Exception
    MessageBox.Show(ex.InnerException.Message)
  End Try
End Using
```

Cuando se añade un nuevo objeto secundario, el objeto primario debe existir en el contexto de objetos o en el origen de datos antes de llamar a **SaveChanges**, de lo contrario se producirá una excepción **InvalidOperationException**.

Cuando se generó el EDM se generaron también un par de métodos parciales denominados *OnNombre_propiedadChanging* y *OnNombre_propiedadChanged*, los cuales son invocados cuando se establece una propiedad.

Borrar filas en la base de datos

Para eliminar filas de una base de datos, se quitan los objetos deseados de la colección de objetos *clase_entidad* que devuelve la consulta **ObjectQuery(Of** *clase_entidad***)** correspondiente y después se envían los cambios a la base de datos.

EF no admite operaciones de eliminación en cascada. Por lo tanto, si desea eliminar una fila de una tabla que tiene restricciones, deberá realizar una de las siguientes tareas:

- Escriba código para eliminar primero los objetos secundarios que impiden que se elimine el objeto primario.

- Establezca la regla ON DELETE CASCADE en la restricción FOREIGN KEY de la base de datos.

De acuerdo con el primer punto anterior, para eliminar una fila de la base de datos siga estos pasos:

1. Realice una consulta para obtener los objetos secundarios que impiden que se elimine el objeto primario.
2. Ejecute la consulta para eliminar los objetos secundarios. Para ello, la clase **ObjectContext** proporciona el método **DeleteObject(***entidad***)** que permite marcar el objeto para borrarlo (será registrado en el **ObjectStateManager** con el estado *Deleted*). La fila correspondiente será eliminada del origen de datos cuando se ejecute **SaveChanges**.
3. Realice una consulta para obtener el objeto primario a eliminar.
4. Ejecute la consulta para eliminar el objeto primario.

5. Envíe los cambios a la base de datos.

El siguiente ejemplo elimina el *alumno* especificado. Para ello, busca en la tabla *alums_asigs* las filas correspondientes al alumno (objetos *alum_asig*) y después elimina estos objetos y su objeto primario (de tipo *alumno*), salvando a continuación los cambios en la base de datos.

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234560

  ' Consulta para obtener los objetos secundarios de idAlum
  Dim consulta1 = _
    From alum In contextoDeObjs.alums_asigs _
    Where alum.id_alumno = idAlum _
    Select alum
  If consulta1.Count() <> 0 Then
    For Each al_as In consulta1
      ' Borrar los objetos secundarios
      contextoDeObjs.DeleteObject(al_as)
    Next
  End If

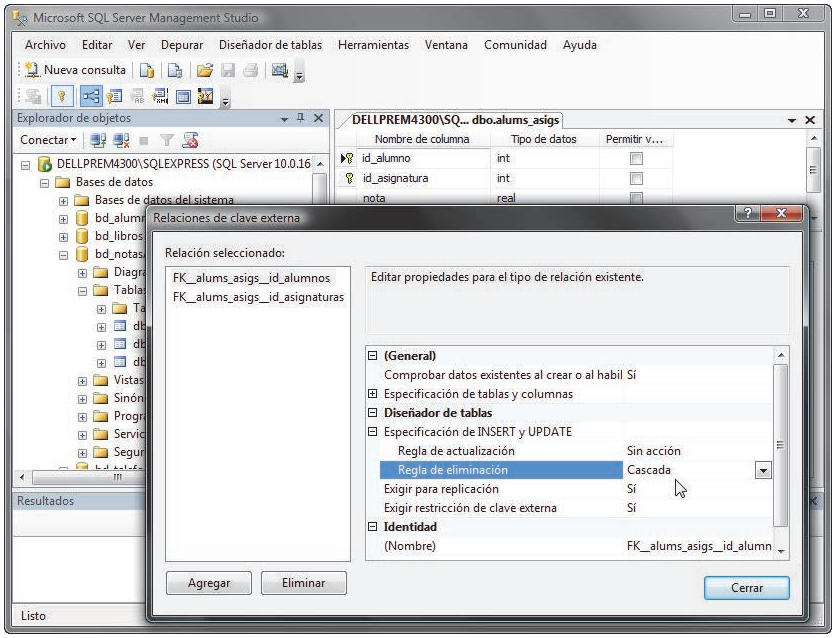
  ' Consulta para obtener el objeto primario idAlum
  Dim consulta2 = _
    From alum In contextoDeObjs.alumnos _
    Where alum.id_alumno = idAlum _
    Select alum
  If consulta2.Count() <> 0 Then
    For Each alum In consulta2
      ' Borrar el objeto primario
      contextoDeObjs.DeleteObject(alum)
    Next
  End If

  If consulta1.Count() = 0 AndAlso consulta2.Count() = 0 Then
    MessageBox.Show("La consulta no contiene elementos")
    Return
  End If

  ' Enviar los cambios a la base de datos
  Try
    contextoDeObjs.SaveChanges()
    MessageBox.Show("Cambios realizados")
  Catch ex As Exception
    MessageBox.Show(ex.InnerException.Message)
  End Try
End Using
```

Como ejercicio, se propone escribir el código necesario para eliminar una asignatura que está asignada a un alumno determinado.

Si quisiéramos realizar operaciones en cascada, tendríamos que establecer la regla ON DELETE CASCADE en la restricción FOREIGN KEY de la base de datos. Esto, en nuestro caso, lo podemos hacer utilizando el administrador de SQL Server. Por ejemplo, si queremos que al borrar un alumno de la tabla *alumnos* se borren también todas las filas correspondientes a este alumno en la tabla *alums_asigs*, es necesario cambiar la *Regla de eliminación* de la relación de clave externa *FK__alums_asigs__id_alumnos* en la base de datos SQL Server asignándole el valor *Cascada* según muestra la figura siguiente.



El código para realizar esta operación sería ahora el siguiente:

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234560

  ' Consulta para obtener los objetos secundarios de idAlum
  Dim consulta = _
    From alum In contextoDeObjs.alumnos _
    Where alum.id_alumno = idAlum _
    Select alum

  If consulta.Count() <> 0 Then
    For Each alum In consulta
```

```
      ' Habilitadas las operaciones de borrado en cascada
      ' Borrar el objeto primario y los secundarios
      contextoDeObjs.DeleteObject(alum)
    Next
  End If

  ' Enviar los cambios a la base de datos
  Try
    contextoDeObjs.SaveChanges()
    MessageBox.Show("Cambios realizados")
  Catch ex As Exception
    MessageBox.Show(ex.InnerException.Message)
  End Try
End Using
```

Problemas de concurrencia

Entity Framework implementa un modelo de concurrencia o simultaneidad optimista, lo cual significa que durante una sesión de trabajo no se bloquean los datos en el origen de datos para que otra sesión simultánea no pueda interferir. El resultado es que los cambios se guardan en la base de datos sin comprobar la concurrencia de forma predeterminada. Por lo tanto, para las propiedades de las entidades del modelo conceptual que puedan experimentar un alto grado de concurrencia, se recomienda definirlas con su atributo **ConcurrencyMode** igual a *fixed*. Cuando una propiedad tiene fijado el modo de concurrencia o simultaneidad, el contexto de objetos comprueba si esa columna cambió en la base de datos antes de guardar los datos en ella y en el caso de que la columna hubiera cambiado, generará una excepción **OptimisticConcurrencyException**.

Como ejemplo, vamos a simular dos sesiones diferentes trabajando sobre la misma fila de la base de datos con el fin de modificar la nota de una asignatura de un alumno:

```
Using contexto1DeObjs As New bd_notasAlumnosEntities(),
      contexto2DeObjs As New bd_notasAlumnosEntities()
  ' SESIÓN 1: contexto1DeObjs
  ' Obtener los datos
  Dim idAlum As Integer = 1234567
  Dim idAsig As Integer = 20590

  ' Consulta para obtener la fila a modificar
  Dim consulta1 = _
    From al_as In contexto1DeObjs.alums_asigs _
    Where al_as.id_alumno = idAlum AndAlso al_as.id_asignatura = idAsig _
    Select al_as

  ' Ver el código T-SQL
```

```
    MessageBox.Show(DirectCast(consulta1, ObjectQuery).ToTraceString())

    ' Ejecutar la consulta y cambiar la nota
    consulta1.First().nota = consulta1.First().nota + 0.5F

    ' SESIÓN 2: contexto2DeObjs
    ' Obtener los datos: mismo alumno y misma asignatura
    ' pero diferente nota
    idAlum = 1234567
    idAsig = 20590

    ' Consulta para obtener la fila a modificar
    Dim consulta2 = _
      From al_as In contexto2DeObjs.alums_asigs _
      Where al_as.id_alumno = idAlum AndAlso al_as.id_asignatura = idAsig _
      Select al_as

    ' Ejecutar la consulta y cambiar la nota
    consulta2.First().nota = consulta2.First().nota + 0.75F

    ' SESIÓN 1: enviar los cambios a la base de datos
    Try
      contexto1DeObjs.SaveChanges()
    Catch ex As Exception
      MessageBox.Show(ex.InnerException.Message)
    End Try

    ' SESIÓN 2: enviar los cambios a la base de datos
    Try
      contexto2DeObjs.SaveChanges()
    Catch ex As Exception
      MessageBox.Show(ex.InnerException.Message)
    End Try
End Using
```

- Se inicia la sesión 1, se obtienen los datos de una fila determinada y se asigna un nuevo valor a la nota.

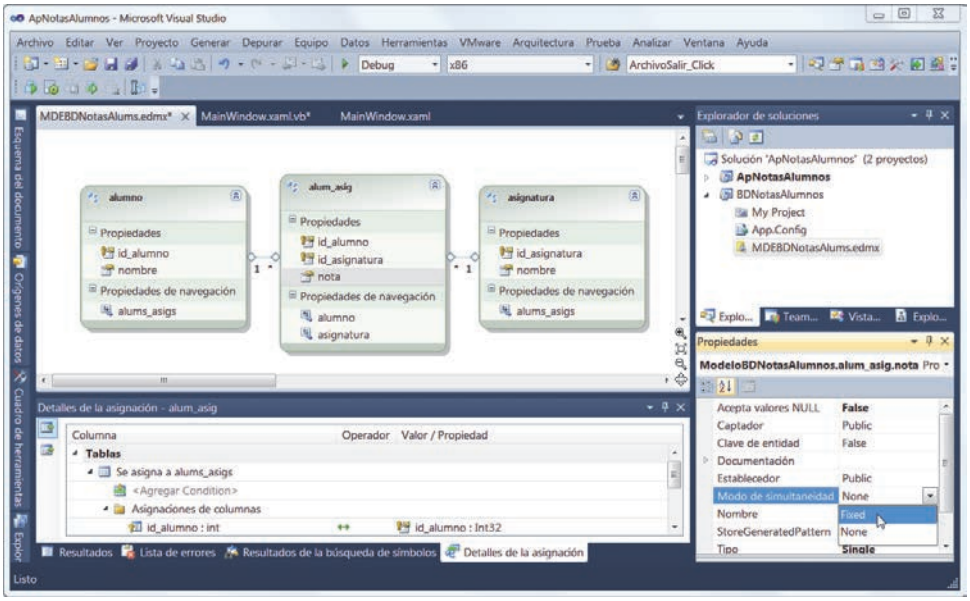
- Antes de que la sesión 1 envíe los datos a la base de datos, se inicia la sesión 2, se obtienen los datos de la misma fila y se asigna un nuevo valor a la nota.

- La sesión 1 envía los datos a la base de datos y se graba la nueva nota.

- La sesión 2, desconociendo que la nota ha sido modificada en el origen de datos por otra sesión, envía los datos a la base de datos y se graba la nueva nota.

El resultado es la última nota grabada: *nota+=0,75*. Pero, ¿es esto lo que queríamos? ¿Qué operación deseamos dar por válida? *nota+=0,5* o *nota+=0,75*.

En un caso como el presentado, para poder elegir entre una de las dos opciones, la solución pasa por cambiar la propiedad *Modo de simultaneidad* de la columna *nota* de la entidad *alumns_asigs*. Para ello, abra el diseñador de entidades, seleccione la columna *nota* en la entidad *alumns_asigs* y cambie su propiedad *Modo de simultaneidad* al valor *Fixed* según muestra la figura siguiente.



Ahora, si hay un conflicto de concurrencia en la columna *nota* se generará una excepción **OptimisticConcurrencyException** (en nuestro ejemplo se producirá en la sesión 2). Para solucionar este conflicto de concurrencia, se recomienda llamar al método **Refresh(***modo-refresco***,** *entidad***)** de **ObjectContext**. El modo de refresco (**RefreshMode**) controla cómo se propagan los cambios:

- Opción *StoreWins*. La opción *almacén gana* indica que los cambios realizados en la *entidad* deben remplazarse por valores procedentes del origen de datos.

- Opción *ClientWins*. La opción *cliente gana* indica que los cambios realizados en la *entidad* no se remplazan por valores procedentes del origen de datos. Por lo tanto, en la siguiente llamada a **SaveChanges**, estos cambios se envían al origen de datos.

Resumiendo, cuando se produce una excepción **OptimisticConcurrencyException**, se debe controlar llamando a **Refresh** y especificando si el conflicto se debe resolver prevaleciendo los datos del origen de datos (*StoreWins*) o los datos del objeto del contexto de objetos (*ClientWins*).

Según lo expuesto, modifique el código anterior para que prevalezcan los datos del origen de datos según se muestra a continuación:

```
' SESIÓN 1: enviar los cambios a la base de datos
Try
  contexto1DeObjs.SaveChanges()
Catch ex As OptimisticConcurrencyException
  MessageBox.Show(ex.Message)

  contexto1DeObjs.Refresh(RefreshMode.StoreWins, consulta1)
  contexto1DeObjs.SaveChanges()
End Try

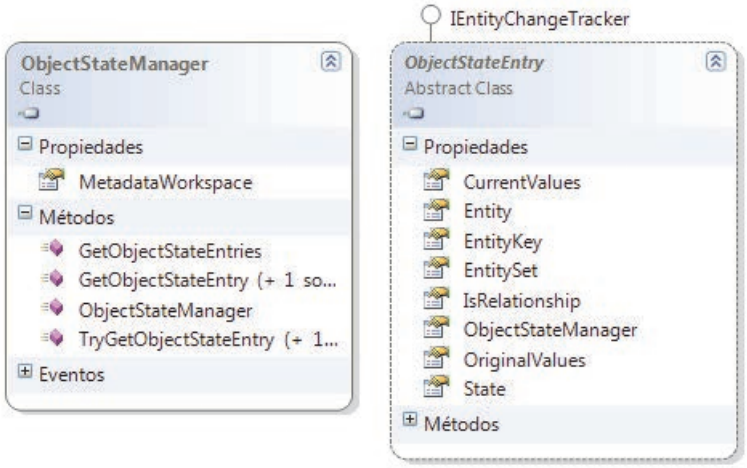
' SESIÓN 2: enviar los cambios a la base de datos
Try
  ' La siguiente operación generará una excepción porque
  ' los datos en la base de datos han cambiado después de
  ' que la sesión 2 los obtuviera
  contexto2DeObjs.SaveChanges()
Catch ex As OptimisticConcurrencyException
  MessageBox.Show(ex.Message)
  ' El conflicto puede resolverse refrescando el contexto 2
  ' aplicando la regla: almacén gana (StoreWins)
  contexto2DeObjs.Refresh(RefreshMode.StoreWins, consulta2)
  ' y salvando los cambios de nuevo
  contexto2DeObjs.SaveChanges()
End Try
```

Cuando se ejecute este código, la sesión 1 grabará *nota+=0,5* en la base de datos y cuando la sesión 2 intente realizar la operación **SaveChanges**, se verificará que la *nota* ha cambiado respecto de la de partida, por lo que se generará la excepción que resolverá el conflicto dando la razón al almacén (base de datos), con lo que el resultado será el que había: *nota+=0,5*.

El seguimiento de cambios

Hemos visto cómo podemos añadir, modificar y suprimir entidades y aplicar estos cambios a la base de datos. Pero estas operaciones, según hemos podido comprobar en el apartado anterior, requieren que se realice un seguimiento de los cambios en el marco de entidades para conocer no sólo los valores actuales, sino los valores originales y así poder compararlos. Pues bien, este seguimiento es controlado por un objeto de la clase **ObjectStateManager** del contexto de objetos. Para ello, cada vez que se ejecuta una consulta en el contexto de objetos, por defecto, para cada una de las entidades que se obtienen se crea un objeto asociado de la clase **ObjectStateEntry**, para almacenar su estado y hacer el seguimiento de los

cambios. La figura siguiente muestra los atributos que envuelve cada uno de estos objetos:



Observamos que cada objeto **ObjectStateEntry** asociado a una entidad proporciona, básicamente, los valores actuales y los valores originales de las propiedades de esa entidad (objeto o relación: **IsRelationship**), una referencia (**Entity**) a la entidad representada, la clave de la misma (**EntityKey**: permite distinguir a la entidad del resto), **EntitySet** obtiene el **EntitySetBase** del **ObjectStateEntry** para determinar si éste está realizando un seguimiento de un objeto o de una relación y el estado de la entidad, así como una referencia al objeto **ObjectStateManager** que la administra. Este último objeto tiene dos métodos interesantes: **GetObjectStateEntries(***estado***)** y **GetObjectStateEntry(***clave/objeto***)**. El primero devuelve una colección de objetos **ObjectStateEntry** para las entidades (objetos o relaciones) que tienen el estado especificado y el segundo devuelve el objeto **ObjectStateEntry** correspondiente a la clave de entidad o al objeto especificado.

Veamos un ejemplo en el que realizamos una consulta sobre el conjunto de entidades *alums_asigs*, modificamos la nota de la entidad obtenida, utilizando el administrador de estado recuperamos las entidades que han sido modificadas (propiedad **State** = **Modified**) y mostramos datos relevantes de las mismas así como sus valores originales y actuales:

```
Using contextoDeObjs As New bd_notasAlumnosEntities()
  ' Obtener los datos
  Dim idAlum As Integer = 1234567
  Dim idAsig As Integer = 20590

  ' Consulta para obtener la fila a modificar
  Dim consulta1 = _
    From al_as In contextoDeObjs.alums_asigs _
```

```
        Where al_as.id_alumno = idAlum And al_as.id_asignatura = idAsig _
        Select al_as

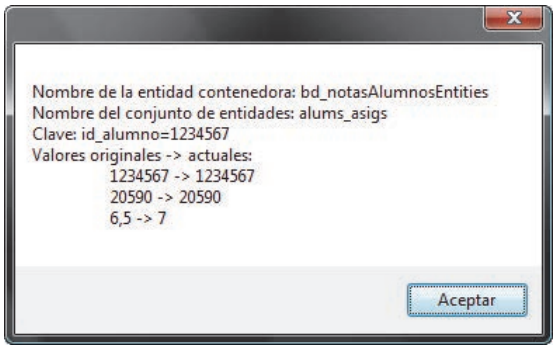
    ' Ejecutar la consulta y cambiar la nota
    consulta1.First().nota = consulta1.First().nota + 0.5F

    Dim admin As ObjectStateManager = _
        contextoDeObjs.ObjectStateManager
    Dim entidadesModificadas = _
        admin.GetObjectStateEntries(EntityState.Modified)

    For Each OSEntry In entidadesModificadas
      Dim info As String = _
           "Nombre de la entidad contenedora: " & _
           OSEntry.EntityKey.EntityContainerName & vbLf & _
           "Nombre del conjunto de entidades: " & _
           OSEntry.EntityKey.EntitySetName.ToString() & vbLf & _
           "Clave: " & _
           OSEntry.EntityKey.EntityKeyValues.First().Key & "=" & _
           OSEntry.EntityKey.EntityKeyValues.First().Value & vbLf & _
           "Valores originales -> actuales:" & vbLf
      For i As Integer = 0 To OSEntry.OriginalValues.FieldCount - 1
        info += vbTab & OSEntry.OriginalValues(i) & " -> " & _
        OSEntry.CurrentValues(i) & vbLf
      Next
      MessageBox.Show(info)
    Next

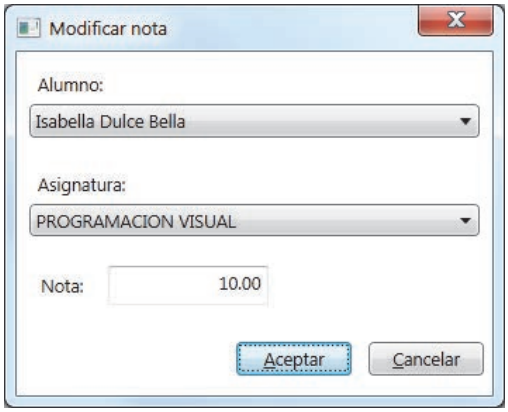
    Try
      contextoDeObjs.SaveChanges()
    Catch ex As Exception
      MessageBox.Show(ex.Message)
    End Try
End Using
```

EntitySetName es el nombre del conjunto de entidades para la entidad a la que pertenece la **EntityKey**. El resultado sería el que muestra la figura siguiente:



EJERCICIOS RESUELTOS

1. Modificar la aplicación anterior, "Cambios en los datos", para que el elemento *Modificar nota* del menú *Cambios en la BD* muestre una caja de diálogo, como la de la figura siguiente, que permita obtener el identificador del alumno, el identificador de la asignatura y la nota obtenida en la misma, datos que utilizaremos para modificar en la base de datos la nota de ese alumno en esa asignatura.



El valor de la caja de texto tendrá que ser validado para garantizar que se trata de un número positivo entre 0 y 10 de tipo **Single**.

La lista de la ventana principal afectada por los cambios deberá ser actualizada después de cualquier cambio. En este ejemplo, la lista a refrescar será la correspondiente al **DataGrid** *dgAsignaturas*.

Empecemos entonces por implementar la actualización de la lista de asignaturas. Para refrescar el contexto de datos del contenedor del **DataGrid** *dgAsignaturas* simplemente tenemos que echar un vistazo al controlador *dgAlumnos_SelectionChanged*. Tomando como referencia ese código, podemos escribir un método privado *RefrescarListaAsigs* que invocaremos cada vez que se realice una modificación y sea necesario actualizar esta lista.

```
Private Sub RefrescarListaAsigs(idAlum As Integer)
  Dim indice As Integer = dgAsignaturas.SelectedIndex
  Dim pAsigsNotas As New CProveedorAsignaturasNotas()
  asigs = pAsigsNotas.ObtenerAsignaturasNotas(idAlum)
  spAsignaturas.DataContext = asigs
  dgAsignaturas.SelectedIndex = indice
End Sub
```

En el código anterior se puede observar que, una vez refrescada la lista, hay que restablecer el índice de la fila que estaba seleccionada para que se muestre la nueva nota.

Entonces, cuando modifiquemos la nota de un alumno será necesario actualizar la lista de asignaturas. ¿Dónde? Pues después de haber enviado los cambios a la base de datos (**SaveChanges**):

```
' Enviar los cambios a la base de datos
Try
  contextoDeObjs.SaveChanges()
  MessageBox.Show("Cambios realizados")
  ' Refrescar dgAsignaturas
  RefrescarListaAsigs(idAlum)
Catch ex As Exception
  MessageBox.Show(ex.InnerException.Message)
End Try
```

A continuación vamos a añadir al proyecto una nueva ventana (una caja de diálogo), denominada *ModificarNota*, con los controles que muestra la figura anterior y que se describen en la tabla siguiente (el diseño de cajas de diálogo fue expuesto en el capítulo *Controles y cajas de diálogo*):

Objeto	**Propiedad**	**Valor**
Etiqueta	Content	Alumno:
Lista desplegable	Name	listaDesAlums
Etiqueta	Text	Asignatura:
Lista desplegable	Name	listaDesAsigs
Etiqueta	Text	Nota:
Caja de texto	Name	ctNota
Botón de pulsación	Content Name IsDefault	_Aceptar btAceptar True
Botón de pulsación	Content Name IsCancel	_Cancelar btCancelar True

¿Cuándo se visualizará esta caja de diálogo? Pues cuando el usuario de la aplicación haga clic en la orden *Modificar nota* del menú *Cambios en la BD*. Para ello, modifique el método vinculado con el evento **Click** de esta orden como se indica a continuación:

```
Private Sub CambiosBDModificarNota_Click(sender As Object,
e As RoutedEventArgs)
```

```
  Using contextoDeObjs As New bd_notasAlumnosEntities()
    ' Datos a obtener del diálogo
    Dim idAlum As Integer
    Dim idAsig As Integer
    Dim notaAlum As Single

    ' Mostrar el diálogo
    Dim dlgModNota As New ModificarNota()
    If dlgModNota.ShowDialog() = True Then
      ' Obtener los datos
      ' idAlum, idAsig y notaAlum
    Else
      Return
    End If

    ' Consulta para obtener la fila a modificar
    ' ...
  End Using
End Sub
```

Según se puede observar en el código anterior, los datos a obtener de la caja de diálogo son los identificadores del alumno (*idAlum*) y de la asignatura (*idAsig*), y la nueva nota a establecer (*notaAlum*). El identificador del alumno será obtenido de la propiedad **SelectedValue** de la lista de alumnos de este diálogo, el identificador de la asignatura, de la propiedad **SelectedValue** de la lista de asignaturas y la nota de la caja de texto *ctNota*. Para facilitar estas operaciones añada a la clase *ModificarNota*, que da lugar a esta caja de diálogo, el dato miembro privado *nota* y las propiedades públicas que se indican a continuación:

```
  Private m_nota As Single = 0.0F

  Public ReadOnly Property Nota() As Single
    Get
      Return m_nota
    End Get
  End Property

  Public ReadOnly Property IdAlumno() As Integer
    Get
      Return CInt(listaDesAlums.SelectedValue)
    End Get
  End Property

  Public ReadOnly Property IdAsignatura() As Integer
    Get
      Return CInt(listaDesAsigs.SelectedValue)
    End Get
  End Property
```

Para que el diálogo pueda interactuar con la base de datos, añada al mismo otro atributo privado que referencie al contexto de dispositivo.

```
Private contextoDeObjs As New bd_notasAlumnosEntities()
```

Cuando el diálogo esté visualizado, *listaDesAlums* deberá mostrar la lista de alumnos y *listaDesAsigs*, la lista de asignaturas. Para ello, añada el controlador del evento **Loaded** de este diálogo y complételo con el código que se indica a continuación:

```
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
  ' Consulta: alumnos
  Dim alums =
    From alum In contextoDeObjs.alumnos
    Select alum
  ' Configurar la lista fija de alumnos
  listaDesAlums.DisplayMemberPath = "nombre"
  listaDesAlums.SelectedValuePath = "id_alumno"
  listaDesAlums.SelectedIndex = 0
  listaDesAlums.ItemsSource = alums

  ' Configurar la lista fija de asignaturas
  listaDesAsigs.DisplayMemberPath = "nombre"
  listaDesAsigs.SelectedValuePath = "id_asignatura"
  listaDesAsigs.SelectedIndex = 0
End Sub
```

El método anterior realiza una consulta sobre la tabla *alumnos* de la base de datos y asigna la colección resultante, *alums*, como origen de datos de la lista *listaDesAlums*. Así mismo, configura esta lista para que muestre el miembro *nombre* del alumno y almacene como valor el miembro *id_alumno* de los objetos de la colección *alums*. También configura la lista *listaDesAsigs* para que muestre el miembro *nombre* de la asignatura y almacene como valor el miembro *id_asignatura* de los objetos de la colección que habrá que obtener desde las tablas *alums_asigs* y *asignaturas* de la base de datos cada vez que se seleccione un nuevo alumno de la lista *listaDesAlums*. Como esta acción genera el evento **SelectionChanged**, escribiremos el controlador de este evento para que obtenga la colección *alumAsigs* de las asignaturas correspondientes al alumno seleccionado, la cual pasará a ser el origen de datos de la lista *listaDesAsigs*:

```
Private Sub listaDesAlums_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  ' Consulta: asignaturas del alumno seleccionado
  Dim id_al As Integer = Convert.ToInt32(listaDesAlums.SelectedValue)
  Dim alumAsigs =
    From al_as In contextoDeObjs.alums_asigs
    Where al_as.id_alumno = id_al
```

```
    Select al_as.asignatura()
  listaDesAsigs.ItemsSource = alumAsigs
End Sub
```

Así mismo, cada vez que se seleccione una asignatura en la lista *listaDesAsigs*, la caja de texto deberá mostrar la nota correspondiente, proceso que escribiremos en el controlador del evento **SelectionChanged** de esta lista:

```
Private Sub listaDesAsigs_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  ' Nota de la asignatura y del alumno seleccionados
  If TryCast(sender, ComboBox).SelectedIndex = -1 Then
    TryCast(sender, ComboBox).SelectedIndex = 0
  End If
  Dim id_al As Integer = Convert.ToInt32(listaDesAlums.SelectedValue)
  Dim id_as As Integer = Convert.ToInt32(listaDesAsigs.SelectedValue)
  Dim consulta =
    From al_as In contextoDeObjs.alums_asigs
    Where al_as.id_alumno = id_al AndAlso al_as.id_asignatura = id_as
    Select al_as.nota
  ctNota.Text = consulta.First().ToString()
End Sub
```

Según lo expuesto, el usuario seleccionará el alumno y la asignatura correspondientes a la nota que desea modificar. Después introducirá la nota en la caja de texto *ctNota*. Para validar esta nota, añada el controlador del botón *Aceptar* y complételo como se indica a continuación:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  If Not NotaNoValida(ctNota.Text) Then
    MessageBox.Show("Nota no válida")
    Return
  End If
  DialogResult = True
End Sub
```

Este controlador verifica si la nota es válida, ejecutando el método *NotaNoValida* escrito a continuación. Si es válida, asigna a la propiedad **DialogResult** del diálogo el valor **True**; el diálogo se cierra y se retorna al método que lo creó. Y si no es válida, se muestra una notificación y se permanece en este diálogo.

```
Private Function NotaNoValida(snota As String) As Boolean
  If Not Single.TryParse(snota, m_nota) Then Return False
  If m_nota < 0 OrElse m_nota > 10 Then Return False
  Return True
End Function
```

El método *NotaNoValida* recibe el contenido de la caja de texto y lo intenta convertir en un valor de tipo **Single** almacenándolo en la variable *nota*. Si la conversión tiene éxito, se verifica que el valor esté entre 0 y 10 y si lo está, se devuelve el valor **True**; si la conversión no tiene éxito porque el contenido de la caja de texto no se ajusta a un número **Single** o porque el valor no está entre 0 y 10, se devuelve el valor **False**.

Finalmente, modifique el controlador del evento **Click** de la orden *Modificar nota* del menú *Cambios en la BD* para que recupere del diálogo *ModificarNota*, por medio de las propiedades que establecimos para tal fin, los datos identificador del alumno (*idAlum*), identificador de la asignatura (*idAsig*) y nota (*notaAlum*), necesarios para realizar la consulta que nos permita obtener la fila de la cual se quiere modificar la nota:

```
Private Sub CambiosBDModificarNota_Click(sender As Object,
e As RoutedEventArgs)
  Using contextoDeObjs As New bd_notasAlumnosEntities()
    ' Datos a obtener del diálogo
    Dim idAlum As Integer
    Dim idAsig As Integer
    Dim notaAlum As Single

    ' Mostrar el diálogo
    Dim dlgModNota As New ModificarNota()
    If dlgModNota.ShowDialog() = True Then
      ' Obtener los datos
      idAlum = dlgModNota.IdAlumno
      idAsig = dlgModNota.IdAsignatura
      notaAlum = dlgModNota.Nota
    Else
      Return
    End If

    ' Consulta para obtener la fila a modificar
    ' ...
  End Using
End Sub
```

Pruebe ahora la aplicación y observe que todo funciona perfectamente. En el apartado siguiente, *Ejercicios propuestos*, le será propuesto que complete de forma análoga el resto de las órdenes del menú *Cambios en la BD*.

RESUMEN

En este capítulo se han expuesto los recursos del lenguaje compatibles con LINQ (***L**anguage **IN**tegrated **Q**uery*). Este lenguaje ofrece la posibilidad de expresar las

operaciones de consulta en el propio lenguaje (Visual Basic, por ejemplo) sobre colecciones de datos de diversa procedencia (objetos en memoria, bases de datos relacionales o documentos XML). Estas consultas son llevadas a cabo mediante lo que se denomina *expresiones de consulta*, por lo que a continuación estudiamos las cláusulas que forman estas expresiones con el fin de aprender a formularlas.

A continuación, introdujimos los proveedores que LINQ proporciona para acceder a los distintos orígenes de datos, para después centrarnos en el proveedor *LINQ to Entities* que permite consultar las entidades que definen el modelo conceptual de *Entity Framework*. ADO.NET *Entity Framework* permite a los desarrolladores crear aplicaciones que acceden a bases de datos elevando el nivel de abstracción del nivel lógico relacional al nivel conceptual, permitiendo así escribir expresiones de consulta independientes de cualquier motor de almacenamiento de datos o esquema relacional determinados.

El siguiente paso fue poner en práctica todo lo expuesto. Para ello, planteamos el diseño de una aplicación que accediera a una base de datos formada por varias tablas relacionadas. Esta aplicación posibilitó aprender cómo conectarse a una base de datos, cómo generar el modelo de entidades, cómo trabajar con las clases de entidad y el contexto de objetos o cómo mostrar datos en una interfaz gráfica.

Finalmente, terminamos este capítulo estudiando cómo realizar cambios en los datos de la base; esto es, modificar filas en la base de datos, insertar filas en la base de datos o borrar filas en la base de datos, lo que nos llevó a estudiar también los problemas de concurrencia.

EJERCICIOS PROPUESTOS

1. Realizar una aplicación, fundamentada en la aplicación “Cambios en los datos”, desarrollada en el apartado anterior, que permita básicamente cuatro acciones:

   a. Poner una nota.
   b. Añadir un alumno.
   c. Añadir una asignatura.
   d. Matricular a un alumno de una asignatura.

CAPÍTULO 8

© F.J.Ceballos/RA-MA

NAVEGACIÓN DE TIPO WEB

Según lo estudiado en los capítulos anteriores, muchas de las aplicaciones Windows tradicionales están organizadas en torno a una ventana con una barra de menús y otra de herramientas, de forma que cuando el usuario hace clic en ellas, las acciones se suceden y otras ventanas aparecen; esto es, puede haber varias ventanas igual de importantes, en las que el usuario empleará más o menos tiempo operando sobre ellas y saltará a otras ventanas siempre que sea necesario. Sin embargo, los desarrolladores de aplicaciones de escritorio se han dado cuenta de que sería beneficioso diseñar cierto tipo de ellas bajo el modelo de navegación de las aplicaciones webs (especialmente cuando se trata de aplicaciones simples y ligeras o cuando lo que buscamos es agilizar el despliegue de la aplicación); este modelo es un modelo basado en páginas y, por lo tanto, muy diferente al modelo tradicional basado en ventanas. Ésta es la razón de por qué WPF incluye su propio sistema de navegación basado en páginas. Esto significa que los usuarios podrán ejecutar la aplicación sin tener que instalarla; simplemente, abrirán su explorador (navegador) y escribirán la localización de la aplicación que desean ejecutar. Este es el modelo de aplicaciones, llamado XBAP (***X**aml **B**rowser **AP**plications* – aplicaciones de explorador XAML de WPF), que vamos a explicar en este capítulo, aplicaciones que se pueden hospedar en Microsoft Internet Explorer o en Firefox por medio del *host* de WPF *PresentationHost.exe*.

WPF, XBAP y Silverlight

WPF, *XBAP* y *Silverlight* son tres tecnologías para el desarrollo de aplicaciones WPF. Una aplicación XBAP combina las características de las aplicaciones webs, esto es, se pueden implementar en un servidor web e iniciarse desde Internet Explorer o Firefox, y de las aplicaciones cliente enriquecidas, esto es, pueden aprovechar las capacidades de WPF. *Silverlight* es una implementación multiplataforma para distintos servidores de .NET Framework con la que se gene-

ran aplicaciones interactivas para la web que soportan multimedia de una forma sencilla, aunque sólo incluye un subconjunto de la tecnología WPF. La tabla siguiente resume las diferencias y similitudes entre las distintas tecnologías:

Aps. WPF estándar	Aps. XBAP	Aps. Silverlight
Son instaladas en la máquina del usuario final.	No son instaladas en la máquina del usuario.	La máquina del usuario final no necesita .Net Framework.
Pueden ser instaladas vía MSI o ClickOnce.	Pueden ser desplegadas vía ClickOnce.	Pueden ejecutarse en cualquier navegador y plataforma.
No pueden instalarse de forma automática nuevas versiones.	Pueden instalarse de forma automática nuevas versiones.	Sólo requieren una descarga de un complemento que se instala en pocos segundos.
Aparecen en el menú de inicio y en agregar o quitar programas.	No aparecen en el menú de inicio ni en agregar o quitar programas.	Se pueden configurar para que se ejecuten fuera del explorador.
Se ejecutan en su propia ventana.	Sólo pueden ejecutarse en IE y Firefox.	Multiplataforma.
S. O. Windows.	S. O. Windows.	Se usan como aplicaciones para Internet.
Se usan como aplicaciones de escritorio.	Se usan como aplicaciones para Intranet.	Se necesita una conexión a Internet.
No se necesita una conexión a Internet.	Se necesita una conexión a Internet.	Pueden utilizar WCF.
Pueden utilizar WCF.	No pueden utilizar WCF.	

Resumiendo, las aplicaciones WPF son, básicamente, aplicaciones cliente de Windows. En cambio, Silverlight y XBAP han sido diseñados para aplicaciones webs; así que, según la tabla anterior, elegirá uno u otro en función de los requisitos de la aplicación y de dónde se vaya a ejecutar.

NAVEGACIÓN

WPF admite la navegación de tipo web con páginas (clase **Page**) e hipervínculos (clase **Hyperlink**) y se puede implementar mediante:

- Páginas compiladas en una aplicación independiente y hospedadas en una ventana de navegación (**NavigationWindow**).

- Páginas hospedadas por un marco (clase **Frame**) hospedado en una página que puede ser independiente o bien compilada en una aplicación XBAP o en una aplicación independiente.

- Páginas independientes que se hospedan en un explorador web.

- Páginas compiladas en una aplicación XBAP que se hospedan en un explorador web: Internet Explorer o Firefox.

Para facilitar la navegación, WPF implementa la clase **NavigationService**, que es el motor de navegación compartido para procesar las solicitudes de navegación utilizando **Frame**, **NavigationWindow** y **XBAP**. Esta clase proporciona métodos de navegación para iniciar la navegación, eventos de navegación para interactuar con ella y un historial de navegación, un diario (**Journal**), para memorizar la navegación hacia delante y hacia atrás. Esto es, **NavigationService** encapsula la funcionalidad que permite descargar el contenido dentro del contexto de una navegación de explorador. Este contenido puede ser cualquier tipo de objeto de .NET Framework y fichero HTML. No obstante, la manera preferida para empaquetar el contenido para la navegación son las páginas (**Page**).

Para aprender acerca de la navegación de tipo web, vamos a generar una aplicación independiente de Windows que permita a los usuarios ver los datos (dirección, teléfono, fotografía, etc.) de la persona seleccionada de una lista. La aplicación constará de varias páginas WPF hospedadas en una ventana con el estilo del explorador.

Crear la base de datos

Para mostrar la aplicación trabajando, sin complicar el desarrollo con una capa de acceso a datos contra una determinada base de datos, tema estudiado en los capítulos anteriores, vamos a utilizar un proveedor de datos XML, denominado *xmlTelefonos*, cuya fuente de datos será el fichero *telefonos.xml* de la carpeta *Datos* del proyecto (el enlace de datos lo iniciamos en el capítulo *Introducción a WPF* y lo continuamos, detalladamente, en los capítulos siguientes, así como el estudio de los proveedores de datos).

La estructura de cada nodo *persona* de la lista *listatelefonos* almacenada en el fichero *telefonos.xml* es así:

```
<?xml version="1.0" encoding="utf-8"?>
<listatelefonos>
  <persona>
    <foto>Imagenes/foto.jpg</foto>
    <nombre>Persona 01</nombre>
    <direccion>Dirección 01</direccion>
    <telefono>Teléfono 01</telefono>
    <es_movil>True</es_movil>
  </persona>
  ...
<listatelefonos>
```

Para poner estos datos a disposición de la aplicación, utilizaremos, como hemos dicho, un proveedor de datos XML.

Crear el proyecto

En este apartado, vamos a crear la infraestructura de la aplicación, que incluye la definición de la aplicación, dos páginas (**Page**) y otros recursos. Según esto, cree un nuevo proyecto de tipo *Aplicación WPF* denominado *ApNavegacionWeb*.

Una vez creado el esqueleto de la aplicación, abra el fichero *Application.xaml*, el cual define una aplicación WPF y los recursos de la misma. También se utiliza este fichero para especificar la interfaz de usuario que se mostrará automáticamente cuando se inicie la aplicación; en este caso, *MainWindow.xaml*:

```
<Application x:Class="Application"
             xmlns="http://.../presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

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

Para acceder a los datos XML, una buena opción es el proveedor **XmlDataProvider** estudiado en capítulos anteriores. Según esto, edite el fichero *telefonos.xml* y guárdelo en una carpeta *Datos* del proyecto y las imágenes a las que hace referencia ese fichero, guárdelas en otra carpeta *Imagenes*. Después, añada a la sección de recursos de **Application** un proveedor XML que tenga como origen de datos dicho fichero:

```
<Application.Resources>
  <XmlDataProvider x:Key="xmlTelefonos"
     Source="Datos/telefonos.xml" XPath="listatelefonos/persona"/>
</Application.Resources>
```

El valor *listatelefonos/persona* del atributo **XPath** indica que se seleccionarán todos los elementos *persona* que son hijos de *listatelefonos*.

A continuación, vamos a editar el fichero *MainWindow.xaml*. Este fichero contiene el código XAML que da lugar a la ventana principal de la aplicación que, en nuestro caso, como veremos a continuación, va a mostrar el contenido en dos páginas: una primera página mostrará la lista de personas y una segunda, los datos relacionados con la persona seleccionada en la primera. Según esto, la aplicación navegará a contenido diferente según la interacción del usuario. Por consiguiente, la ventana principal tiene que ser una ventana que admita la navegación por el contenido, facilidad que proporciona **NavigationWindow**, clase que se deriva de **Window**, por lo tanto, hereda todas las propiedades de ésta. Según lo expuesto, cambie el elemento **Window** por **NavigationWindow**, cambie el título (**Title**) para que ahora sea "Lista de teléfonos" y elimine el **Grid**, porque este tipo de ventana no admite contenido directo. El resultado será el siguiente:

```
<NavigationWindow x:Class="MainWindow"
        xmlns="http://.../presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Lista de teléfonos" Height="350" Width="525">

</NavigationWindow>
```

El cambio de **Window** por **NavigationWindow** también lo tendremos que realizar en la clase *MainWindow* definida en el fichero de código subyacente que contiene el código para controlar los eventos declarados en *MainWindow.xaml*. Por lo tanto, edite el fichero *MainWindow.xaml.vb* y haga que la clase *MainWindow* se derive ahora de **NavigationWindow**:

```
Class MainWindow
  Inherits NavigationWindow

End Class
```

NavigationWindow

La clase System.Windows.Navigation.**NavigationWindow** se deriva de **Window** para añadir la funcionalidad necesaria para navegar al contenido y mostrarlo (otro objeto que permite navegación es **Frame**). El contenido puede ser cualquier objeto .NET Framework o fichero HTML, aunque lo más habitual es que sea un objeto de la clase **Page** que representa una página de contenido a la que se puede navegar. Para navegar a un contenido desde **NavigationWindow**, se puede hacer estableciendo la propiedad **Source** de este objeto con el URI correspondiente al contenido al que se desea navegar. Por ejemplo:

```
<NavigationWindow x:Class="MainWindow"
        xmlns="http://.../presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Lista de teléfonos" Height="350" Width="525"
        Source="PagListado.xaml">

</NavigationWindow>
```

También se puede navegar a un contenido mediante programación invocando el método **Navigate**.

Cuando se navega por el contenido, **NavigationWindow** graba el historial de navegación. Para avanzar y retroceder por el historial de navegación, **NavigationWindow** muestra automáticamente una interfaz de usuario de navegación como la mostrada en la figura siguiente. Mostrar o no esta interfaz es configurable a través de la propiedad **ShowsNavigationUI**.



Cada vez que se invoca al método **Navigate** o al método **GoForward** de System.Windows.Navigation.**NavigationService** se añade una entrada al historial de retroceso de navegación y cada vez que se invoca al método **GoBack** se añade una entrada al historial de avance de navegación. Los métodos **CanGoBack** y **CanGoForward** informan de si hay entradas en el historial de retroceso o avance de navegación, respectivamente. Se puede eliminar la entrada más reciente del historial de retroceso de navegación llamando a **RemoveBackEntry**.

También, durante una navegación o después de ella, **NavigationWindow** proporciona información del contenido al que se navega (URI) por medio de su propiedad **Source**, el URI del contenido actual por medio de su propiedad **CurrentSource** y el propio contenido (un objeto) por medio de su propiedad **Content**. Esto no quiere decir que **NavigationWindow** almacene el objeto de contenido en el historial de navegación, ya que consumiría mucha memoria, sino que crea un nuevo objeto de contenido cada vez que se navega a él mediante el historial de navegación, por lo que el estado del contenido no se recuerda de una navegación a la siguiente, aunque en WPF existen técnicas para almacenar parte de un estado de una parte del contenido en el historial de navegación; por ejemplo, mediante propiedades de dependencia o mediante **IProvideCustomContentState** y **CustomContentState**.

Resumiendo, un navegador es una clase que admite la navegación y el historial de navegación. Los dos navegadores de WPF son **NavigationWindow** y **Frame**. Cada uno de estos objetos actúa como anfitrión (*host*) del contenido al que se navega. De la definición se deduce que los tipos de *host*, **NavigationWindow** y **Frame** o un explorador, crean su propio objeto *servicio de navegación* al que se tiene acceso invocando al método **GetNavigationService**. Por ejemplo:

```
' Obtener el servicio de navegación
Dim ns As NavigationService = NavigationService.GetNavigationService(Me)
' Ir al elemento más reciente del historial de retroceso de navegación
ns.GoBack()
```

Page

Un objeto de la clase **Page**, derivada de System.Windows.**FrameworkElement**, encapsula una página de contenido a la que se puede navegar. De la funcionalidad proporcionada por la misma, destacamos las siguientes propiedades:

- **KeepAlive**. Permite administrar la duración de la página indicando si se conserva el objeto **Page** en el historial de navegación. El valor predeterminado para esta propiedad es **False**.

- **NavigationService**. Los tipos de *host* como **NavigationWindow** y **Frame** o un explorador, crean su propio objeto **NavigationService** (servicio de navegación) al que se tiene acceso desde esta propiedad.

- Propiedades para configurar el aspecto de la página como Background, Content, FontFamily, FontSize, Foreground, ShowsNavigationUI, Template, Title o WindowTitle.

- Propiedades para configurar el aspecto de la ventana del *host* como WindowHeight, WindowWidth o WindowTitle. **WindowTitle** obtiene o establece el título del *host* de una **Page**.

Una página se puede hospedar en un objeto **Window**, **NavigationWindow**, **Frame**, **UserControl** o en un explorador.

Para navegar entre las páginas de una aplicación podemos hacerlo mediante declaración utilizando un objeto **Hyperlink** (encapsula un URI – identificador de recursos uniforme), mediante programación invocando al método **NavigationService.Navigate** o visualmente utilizando la interfaz de navegación del *host*. Cuando navegamos entre páginas utilizando un URI, el servicio de navegación mantiene el URI pero no el objeto de contenido (**Page**, por ejemplo). Si se necesita conservar el objeto **Page** en el diario de navegación, puede navegar a dicho ob-

jeto mediante programación, **NavigationService.Navigate**, o puede especificar que WPF conserve el objeto **Page** en el diario, estableciendo la propiedad **KeepAlive** de esa página en **True**.

Añadir páginas a la aplicación

Una vez creado el esqueleto de la aplicación, como siguiente paso, vamos a añadir a la misma las dos páginas mencionadas anteriormente.

La primera página que vamos a añadir será la que se muestre cuando se inicie la aplicación, y su función será mostrar la lista de personas de la que el usuario podrá seleccionar aquélla cuyos datos desee conocer. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Agregar* > *Página* > *Página (WPF)*. Denomine a la página *PagListado*, por ejemplo. Una vez añadida, edite el fichero *PagListado.xaml* y establezca el título del *host* con el valor "Listado de personas" según se puede ver a continuación:

```
<Page x:Class="PagListado"
      xmlns="http://.../presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://.../markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      WindowTitle="Listado de personas">

    <Grid>

    </Grid>
</Page>
```

Lo anterior no enseña que por cada página perteneciente a la aplicación creamos un objeto **Page** que, normalmente, estará soportado por el fichero de código XAML correspondiente y por el fichero de código Visual Basic subyacente. Las líneas de color gris pueden ser eliminadas, sólo sirven para trabajar durante el diseño; en este caso, para limitar el tamaño de la página en el diseñador (por ejemplo puede establecer el alto y el ancho de la página para que tengan los mismos valores que *MainWindow*). Los elementos identificados como **mc:Ignorable** no generarán errores al procesarlos en un procesador XAML porque si no se pueden resolver como un tipo subyacente o una construcción de programación, se omiten; esto es lo que ocurre en el ejemplo con los elementos pertenecientes al espacio de nombres **d**.

El siguiente paso, que ya vimos en el apartado anterior, sería editar el fichero *MainWindow.xaml* para establecer la propiedad **Source** de **NavigationWindow** al

valor “PagListado.xaml”. Esto hace que esta página sea la primera que se muestre cuando se inicie la aplicación.

Siguiendo un procedimiento análogo, añada otra página, *PagDatos*, con el título “Datos personales” para el *host*, según muestra el código siguiente:

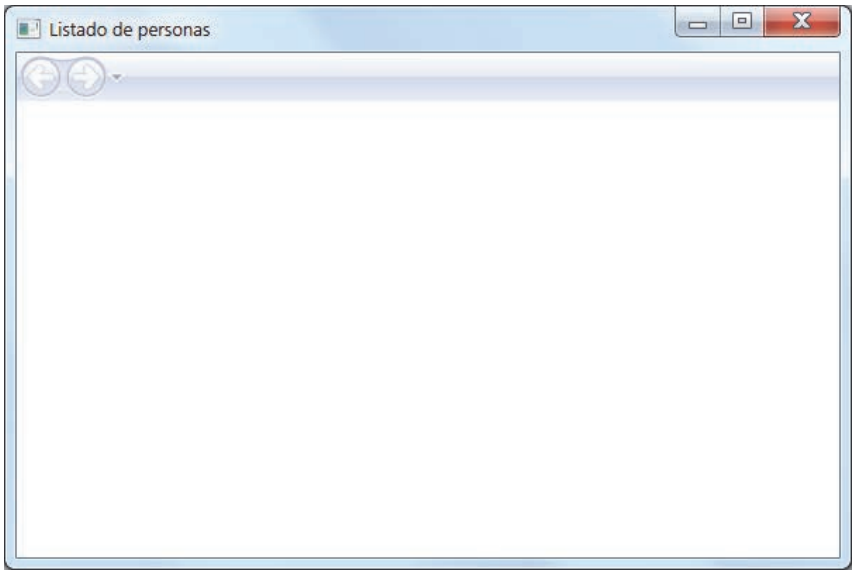
```
<Page x:Class="PagDatos"
      xmlns="http://.../presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://.../markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      WindowTitle="Datos personales">

    <Grid>

    </Grid>
</Page>
```

Esta página, como la anterior, tiene su propio fichero de código subyacente para controlar la lógica para responder a los datos proporcionados por el usuario.

Si ahora compila y ejecuta la aplicación, observará que se muestra la ventana **NavigationWindow** con su barra de navegación mostrando la página de contenido *PagListado*:



Diseño de la interfaz gráfica

Según hemos visto en los capítulos anteriores, los controles de diseño: Canvas, DockPanel, Grid, StackPanel y WrapPanel, proporcionan un modo ordenado de colocar los elementos de la interfaz gráfica y de controlar el tamaño y la posición de los mismos cuando dicha interfaz cambia de tamaño. De todos ellos, el que hemos utilizado con mayor frecuencia ha sido el control de diseño **Grid**, porque es el más idóneo en la mayoría de los casos por la versatilidad que ofrece.

Trabajando con páginas la técnica empleada sigue siendo la misma. Por ejemplo, vamos a diseñar la página *PagListado* para que tome el aspecto mostrado en la figura siguiente; se trata de un diseño en el que observamos una rejilla con dos columnas y cuatro filas. La columna 0 tiene un ancho de 240 y la columna 1 tiene un ancho de * (espacio restante disponible). Las filas 0 y 2 tienen de alto * (reciben una proporción ponderada del espacio restante disponible) y las filas 1 y 3 tienen *Auto* de alto (se ajustan al tamaño de su contenido).



En la columna 1, fila 0, hemos puesto un **Label** (“Ver datos personales”), en la columna 1, fila 1, hemos puesto un **Border** con un **Label** (“Nombres”), en la columna 1, fila 2, hemos puesto un **ListBox** y en la columna 1, fila 3, hemos puesto un **Button**. Finalmente, se ha añadido una imagen como fondo del **Grid** y se ha aplicado estilo a cada uno de los controles.

Como se puede observar, esta página muestra una lista de personas para que los usuarios de la aplicación puedan seleccionar el nombre de aquélla cuyos datos,

por ejemplo el teléfono, desean conocer. Fíjese en que la lista sólo muestra el nombre de cada persona. Para ello, tenemos que vincular el origen de datos *xmlTelefonos* con la lista, por medio de su propiedad **ItemsSource**:

```
<ListBox Name="lstPersonas" Grid.Column="1" Grid.Row="2"
         ItemsSource="{Binding Source={
                                StaticResource xmlTelefonos}}" />
```

Este enlace proporciona objetos *persona*; el resultado es que cada elemento de la lista es una representación de cadena de cada objeto *persona* de la colección subyacente. Por lo tanto, para que cada elemento de la lista muestre sólo el nombre, hay que redefinir la plantilla de datos asociada con cada uno de ellos a través de la propiedad **ItemTemplate** de la lista:

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <StackPanel>
      <TextBlock Text="{Binding XPath=nombre}" />
    </StackPanel>
  </DataTemplate>
</ListBox.ItemTemplate>
```

Esta plantilla especifica que cada elemento de la lista estará formado por un bloque de texto vinculado al atributo *nombre* del objeto *persona*. El resto de los atributos de este objeto simplemente se ignoran.

Según lo expuesto, abra la página *PagListado.xaml* y agregue el código XAML siguiente entre las etiquetas del **Grid**:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="240" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="Auto"/>
    <RowDefinition />
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>

  <Grid.Background>
    <ImageBrush ImageSource="Imagenes/telefono.jpg"/>
  </Grid.Background>

  <!-- Título -->
  <Label Grid.Column="1"
```

```
          Style="{StaticResource EstiloTextoCabecera}">
    Ver datos personales
  </Label>

  <!-- Lista de personas -->
  <Border Grid.Column="1" Grid.Row="1"
          Style="{StaticResource EstiloCabeceraDeLista}">
    <Label Content="Nombres"
           Style="{StaticResource EstiloTextoCabeceraDeLista}"/>
  </Border>
  <ListBox Name="lstPersonas" Grid.Column="1" Grid.Row="2"
           Style="{StaticResource EstiloLixtBox}"
           ItemsSource="{Binding Source={
                                 StaticResource xmlTelefonos}}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding XPath=nombre}" />
        </StackPanel>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>

  <!-- Botón mostrar datos -->
  <Button Name="btMostrar" Grid.Column="1" Grid.Row="3"
          Content="Mostar datos"
          Style="{StaticResource EstiloButton}" />
</Grid>
```

Una interfaz gráfica utiliza estilos para permitir la reutilización de las distintas apariencias en diversos elementos. Esta posibilidad de reutilizar los estilos ayuda a simplificar la creación y la administración del código XAML. Dicho de una forma sencilla, un estilo define las propiedades y sus valores para los distintos elementos de la interfaz.

Vamos a definir los estilos que hemos aplicado en el código anterior, a los controles de *PagListado*, en un ámbito accesible desde todas las páginas, como es la sección de recursos de la aplicación. Según esto, abra el fichero *Application.xaml* y agregue el código XAML siguiente entre las etiquetas *Application.Resources*:

```
<Application.Resources>
  ...

  <!-- Estilo texto cabecera -->
  <Style x:Key="EstiloTextoCabecera">
    <Setter Property="Label.VerticalAlignment"
            Value="Center"></Setter>
    <Setter Property="Label.FontFamily" Value="Arial"></Setter>
```

```
    <Setter Property="Label.FontWeight" Value="Bold"></Setter>
    <Setter Property="Label.FontSize" Value="20"></Setter>
    <Setter Property="Label.Foreground" Value="#0088dd"></Setter>
  </Style>

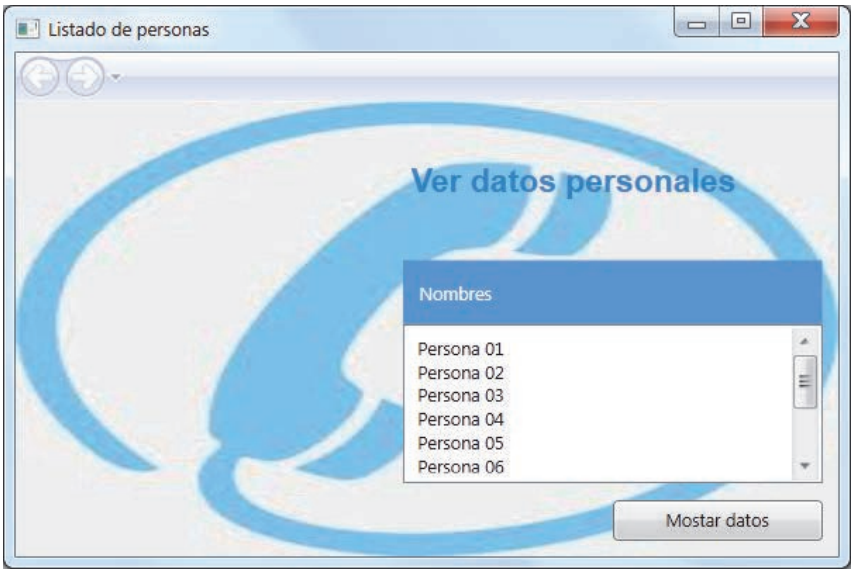
  <!-- Estilo cabecera de la lista -->
  <Style x:Key="EstiloCabeceraDeLista"
         TargetType="{x:Type Border}">
    <Setter Property="Height" Value="40" />
    <Setter Property="Margin" Value="0,0,10,0"/>
    <Setter Property="Padding" Value="5" />
    <Setter Property="Background" Value="#4F99E4" />
  </Style>

  <!-- Estilo texto de la cabecera de la lista -->
  <Style x:Key="EstiloTextoCabeceraDeLista"
         TargetType="{x:Type Label}">
    <Setter Property="Foreground" Value="White" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="HorizontalAlignment" Value="Left" />
  </Style>

  <!-- Estilo LixtBox -->
  <Style x:Key="EstiloLixtBox" TargetType="{x:Type ListBox}">
    <Setter Property="Margin" Value="0,0,10,0" />
    <Setter Property="Padding" Value="5" />
  </Style>

  <!-- Estilo Button -->
  <Style x:Key="EstiloButton" TargetType="{x:Type Button}">
    <Setter Property="Width" Value="130" />
    <Setter Property="Height" Value="25" />
    <Setter Property="Margin" Value="0,10,10,10" />
    <Setter Property="HorizontalAlignment" Value="Right" />
  </Style>
</Application.Resources>
```

Compile y ejecute la aplicación, y observe los resultados.



Si recordamos, esta página se muestra porque hemos fijado en el código XAML la propiedad **Source** de *MainWindow*, derivada de **NavigationWindow**, con el valor *PagListado*. No obstante, esta operación también puede realizarse desde el fichero de código subyacente así:

```
Public Sub New()
  InitializeComponent()
  Me.Content = New PagListado()
End Sub
```

Cuando finaliza la ejecución del constructor *MainWindow* se han creado dos objetos: un objeto *MainWindow* (derivado de **NavigationWindow**) y un objeto *PagListado* (derivado de **Page**), para después asignar la página de contenido a la propiedad **Content** de *MainWindow*.

El siguiente paso es realizar el diseño de la página *PagDatos* que va a mostrar los datos de la persona seleccionada, para que tome el aspecto mostrado en la figura siguiente. Se trata de un diseño en el que observamos una rejilla con dos columnas y cuatro filas. La columna 0 tiene un ancho de 240 y la columna 1 tiene un ancho de * (espacio restante disponible). Las filas 0 y 1 tienen de alto * (reciben una proporción ponderada del espacio restante disponible) y las filas 2 y 3 tienen *Auto* de alto (se ajustan al tamaño de su contenido).

En la columna 1, fila 0, hemos puesto un **Label** (“Datos personales de:”), en la columna 1, fila 1, hemos puesto un **Image** para mostrar el atributo foto del objeto *persona*, en la columna 1, fila 2, hemos puesto un **Border** con un **Label** para mostrar el atributo nombre y en la columna 1, fila 3, hemos puesto un **Border** con

otro **Grid** secundario con dos columnas y tres filas para mostrar las etiquetas y los valores de los atributos dirección, teléfono y si éste es o no un dispositivo móvil. Finalmente, se ha añadido una imagen como fondo del **Grid** principal y se ha aplicado estilo a cada uno de los controles.



Observe que la finalidad de este formulario es mostrar los atributos *foto*, *nombre*, *direccion*, *telefono* y *es_movil* de la persona seleccionada en la lista de la página anterior. Para ello, tendremos también que vincular los controles correspondientes de esta página con los atributos mencionados, según puede verse en el código siguiente. La fuente de datos será el objeto *persona* seleccionado.

Según lo expuesto, abra la página *PagDatos.xaml* y agregue el código XAML siguiente entre las etiquetas del **Grid**:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="240" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
```

```
    <Grid.Background>
      <ImageBrush ImageSource="Imagenes/telefono.jpg"/>
    </Grid.Background>

    <!-- Título -->
    <Label Grid.Column="1"
           Style="{StaticResource EstiloTextoCabecera}">
      Datos personales de:
    </Label>
    <!-- Foto -->
    <Image Grid.Column="1" Grid.Row="1" Source="{Binding XPath=foto}"
            HorizontalAlignment="Left"  Height="64" Width="64"/>
    <!-- Nombre -->
    <Border Grid.Column="1" Grid.Row="2"
            Style="{StaticResource EstiloCabeceraDeLista}">
      <Label Content="{Binding XPath=nombre}"
             Style="{StaticResource EstiloTextoCabeceraDeLista}"/>
    </Border>
    <!-- Dirección, teléfono y si es o no móvil -->
    <Border Grid.Column="1" Grid.Row="3"
            Style="{StaticResource EstiloBorder}">
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition/>
          <RowDefinition/>
          <RowDefinition/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="0" Text="Dirección: "
                   Style="{StaticResource EstiloTextBlock}"/>
        <TextBlock Grid.Row="0" Grid.Column="1"
                   Text="{Binding XPath=direccion}"
                   Style="{StaticResource EstiloControl}"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Text="Teléfono: "
                   Style="{StaticResource EstiloTextBlock}"/>
        <TextBlock Grid.Row="1" Grid.Column="1"
                   Text="{Binding XPath=telefono}"
                   Style="{StaticResource EstiloControl}"/>
        <TextBlock Grid.Row="2" Grid.Column="0" Text="Móvil: "
                   Style="{StaticResource EstiloTextBlock}"/>
        <CheckBox Grid.Row="2" Grid.Column="1"
                  Style="{StaticResource EstiloControl}"
                  IsChecked="{Binding XPath=es_movil}" />
      </Grid>
    </Border>
  </Grid>
```

Igual que hicimos con la página anterior, vamos a definir los estilos que hemos aplicado en el código anterior, a los controles de *PagDatos*, en la sección de recursos de la aplicación. Según esto, abra el fichero *Application.xaml* y agregue el código XAML siguiente entre las etiquetas *Application.Resources*:

```
<Application.Resources>
  ...

  <!-- Estilo Border -->
  <Style x:Key="EstiloBorder" TargetType="{x:Type Border}">
    <Setter Property="BorderBrush" Value="Blue" />
    <Setter Property="BorderThickness" Value="2" />
    <Setter Property="Margin" Value="0,0,10,10" />
    <Setter Property="Padding" Value="7" />
  </Style>

  <!-- Estilo TextBlock -->
  <Style x:Key="EstiloTextBlock" TargetType="{x:Type TextBlock}">
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="Margin" Value="5" />
  </Style>

  <!-- Estilo Control -->
  <Style x:Key="EstiloControl">
    <Setter Property="Control.VerticalAlignment" Value="Center" />
    <Setter Property="Control.HorizontalAlignment" Value="Left" />
    <Setter Property="Control.Margin" Value="5" />
  </Style>
</Application.Resources>
```

Lógica de negocio

Finalizada la capa de presentación, vamos a escribir la lógica de negocio para que las acciones de los usuarios sobre la interfaz tengan la respuesta adecuada.

Cuando un usuario ejecute la aplicación le será mostrada la página *PagListado* que visualiza la lista de personas. A continuación, seleccionará la persona de la lista de la cual desea conocer sus datos y pulsará el botón "Mostrar datos". ¿Qué proceso tiene que desencadenar esta acción? Pues navegar a la página *PagDatos* pasándole el objeto *persona* seleccionado para mostrar los datos de esta persona. Después, el usuario podrá retroceder a la página anterior o cerrar la aplicación. ¿Desde dónde ejecutamos ese proceso? Pues desde el controlador del evento **Click** del botón. Por lo tanto, añada este controlador en el fichero de código subyacente de *PagListado* y edítelo como se indica a continuación:

```
Private Sub btMostrar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim pagDatos As New PagDatos(Me.lstPersonas.SelectedItem)
  Me.NavigationService.Navigate(pagDatos)
End Sub
```

Observe que para navegar a una página lo que hacemos es obtener el servicio de navegación que está utilizando el *host* de la página que se está mostrando e invocar a su método **Navigate** pasando como argumento el objeto a cuyo contenido se desea navegar asíncronamente.

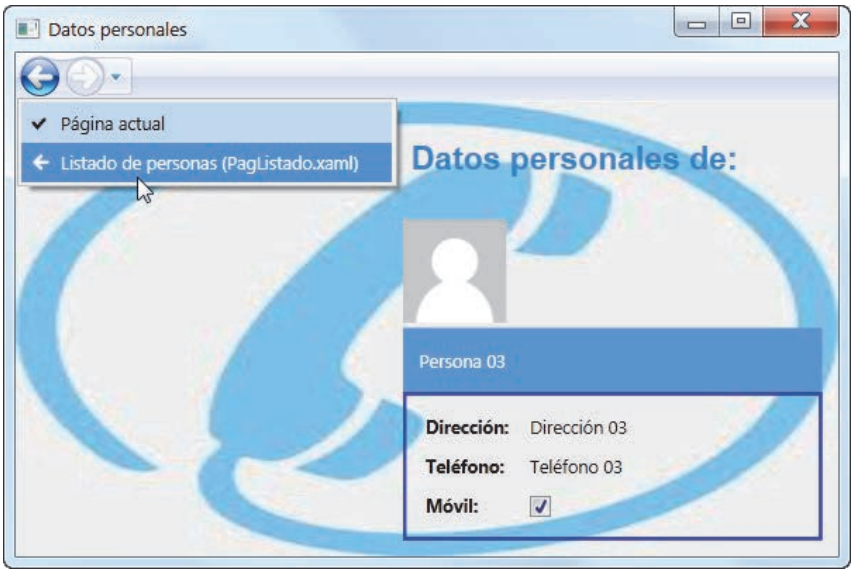
En este caso, además, al crear el objeto *PageDatos* pasamos como parámetro a su constructor el elemento seleccionado de la lista (objeto *persona*), con el fin de mostrar sus datos. Esto implica que la clase *PageDatos* debe tener un constructor con un parámetro que admita el argumento pasado, objeto que será el origen de datos para los controles de *PageDatos*. Por lo tanto, abra el fichero de código subyacente de la página *PageDatos* y añada este constructor a dicha clase:

```
Class PagDatos
  Public Sub New()
    InitializeComponent()
  End Sub

  Public Sub New(persona As Object)
    Me.New()
    Me.DataContext = persona
  End Sub
End Class
```

Observe que el constructor con un parámetro invoca al constructor sin parámetros para construir la página y después asigna al contexto de datos de la misma el objeto *persona* pasado como argumento.

Ejecute la aplicación y pruebe a navegar en ambas direcciones, haciendo clic en el botón “Mostrar datos” y utilizando los botones de la interfaz de navegación mostrada automáticamente por **NavigationWindow**. Esta interfaz, además de los botones, muestra un menú con los sitios a los que se puede navegar.



Pasar datos entre páginas

En el apartado anterior, utilizamos el constructor de la clase *PagDatos* para pasar datos a esta página desde la página *PagListado*. Otra alternativa para pasar datos entre dos páginas, aunque un poco más laboriosa, es definir propiedades que soporten los datos a pasar. Por ejemplo:

1. Podemos definir en *PagDatos* la propiedad:

```
Public Property Persona() As Object
```

2. Asignar, en el controlador *btMostrar_Click*, a esta propiedad la persona seleccionada en la lista:

```
Private Sub btMostrar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim pagDatos As New PagDatos()
  pagDatos.Persona = Me.lstPersonas.SelectedItem
  Me.NavigationService.Navigate(pagDatos)
End Sub
```

3. Y asignar el valor de esta propiedad al contexto de datos de *PagDatos* cuando se cargue esta página (evento **Loaded**):

```
Private Sub Page_Loaded(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Me.DataContext = Persona
End Sub
```

Duración y diario de las páginas

En los apartados *NavigationWindow* y *Page* al hablar del diario de navegación dijimos que, para optimizar memoria, el comportamiento predeterminado del diario es almacenar los metadatos (datos que describen otros datos) de **Page** en cada entrada del diario en lugar de una referencia a dicho objeto. Así, cuando se navega a una entrada del diario, se utilizan esos metadatos para crear un nuevo objeto **Page**. Por consiguiente, cada objeto **Page** visitado dura desde que se navega al mismo (se construye y se carga) hasta que desde él se navega a otra parte (se descarga). Puede implementar los controladores de los eventos **Loaded** y **Unloaded** para que, junto con el constructor, muestren mensajes del hecho ocurrido: objeto construido, cargado o descargado. Por ejemplo, si navega a *PagListado* > *PagDatos* > *PagListado*, observará que los hechos se suceden así:

```
PagListado construida
PagListado cargada
PagDatos construida
PagListado descargada
PagDatos cargada
PagListado construida
PagDatos descargada
PagListado cargada
```

Si ahora, en este estado, vuelve a la *PagDatos* mediante el botón de la interfaz de navegación, no mediante el botón "Mostrar datos", lo que sucede es:

```
PagListado descargada
PagDatos cargada
```

¿Qué observamos? Que la página *PagDatos* no se ha vuelto a construir de nuevo, hecho que sí ocurre con *PagListado*, simplemente se ha cargado. ¿Por qué? Porque la navegación a *PagDatos* se hizo invocando al método **Navigate**. Entonces, cuando invocamos al método **Navigate** con un objeto, no con un URI, el servicio de navegación no sabe cómo volver a crear el objeto, razón por la que lo guarda en la caché, para poder recuperarlo, lo que tendría el mismo efecto que establecer **KeepAlive** en **True** para esta página. Por tanto, si un objeto **Page** necesita iniciarse cada vez que se navega al mismo, deberá mover ese código de iniciación del constructor al controlador del evento **Loaded**.

Resumiendo, de forma predeterminada, WPF no almacena objetos **Page** en el historial de navegación, sino que se crea un nuevo objeto cada vez que se navega a esa página partiendo de los metadatos relacionados con la entrada de página almacenada en el historial de navegación. Pero también es posible que WPF retenga los objetos **Page** en el historial de navegación, aunque esto puede ser caro desde un punto de vista de consumo de memoria.

No obstante, cuando sea necesario, se puede usar una propiedad de dependencia del diario para almacenar y recuperar el estado de la página mediante el historial de navegación. Una propiedad de dependencia del diario es una propiedad de dependencia que tiene el metadato **Journal**. Esta técnica es muy útil cuando necesitemos almacenar un elemento de estado de página en el objeto **Page**.

Por ejemplo, si aplicamos esta técnica a la propiedad *Persona* definida anteriormente (aunque no sea necesario en este ejemplo; sólo como aclaración de lo explicado), el resultado sería éste:

```
Class PagDatos
  Private Shared PersonaProperty As DependencyProperty

  Shared Sub New()
    Dim metadatos As New FrameworkPropertyMetadata()
    metadatos.Journal = True
    PersonaProperty = DependencyProperty.Register(
      "PersonaProperty", GetType(Object),
      GetType(PagDatos), metadatos, Nothing)
  End Sub

  Public Property Persona() As Object
    Get
      Return GetValue(PersonaProperty)
    End Get
    Set(value As Object)
      SetValue(PersonaProperty, value)
    End Set
  End Property
  '...
End Class
```

Observamos que la propiedad de dependencia ha sido marcada con la indicación *metadatos.Journal = True* para poder restaurar su estado entre navegaciones (varios controles WPF, por ejemplo **TextBox**, utilizan este mecanismo para mantener su estado entre navegaciones).

Este mecanismo obliga a seriar el valor de esa propiedad cuando se abandona la página, para poderlo recuperar cuando se vuelva a la página. Por lo tanto, el tipo de la propiedad tiene que estar marcado como seriable.

Hyperlinks

La forma más fácil de navegar desde una página a otra es utilizando un **Hyperlink**: un objeto que implementa la propiedad **NavigateUri** que especifica el URI del contenido al que se desea navegar al hacer clic en ese hipervínculo. Sin em-

bargo, esta forma de navegación sólo es válida si el elemento primario directo o indirecto del **Hyperlink** es un *host* de navegación (**NavigationWindow**, **Frame** o un explorador).

Por ejemplo, supongamos que la propiedad **Source** de *MainWindow*, derivada de **NavigationWindow**, hace referencia en el código XAML a una página *Menu.axml* (en lugar de a *PagListado.axml*) definida así:

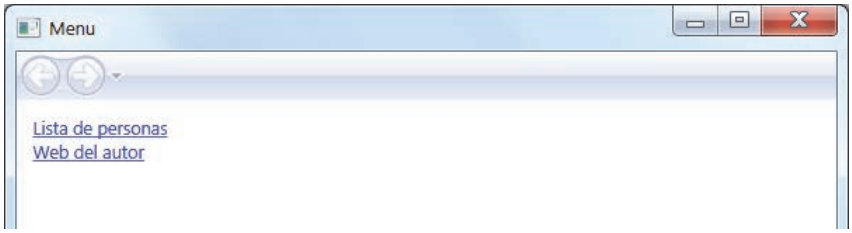
```
<Page x:Class="Menu"
      ...
      WindowTitle="Menu">

  <Grid Margin="10">
    <TextBlock>
      <Hyperlink NavigateUri="PagListado.xaml">
        Lista de personas
      </Hyperlink>
      <LineBreak />
      <Hyperlink NavigateUri="http://www.fjceballos.es">
        Web del autor
      </Hyperlink>
      <LineBreak />

      <!-- Otras opciones -->

    </TextBlock>
  </Grid>
</Page>
```

Esta página mostrará un enlace, entre otros, por medio de un objeto **Hyperlink** que permitirá navegar a la página *PagListado.xaml*. El URI se corresponde con una ruta relativa que especifica dónde está ubicada la página; en este caso, en la misma carpeta que la aplicación. Si ejecuta ahora la aplicación, el resultado sería el siguiente:



Ahora el usuario haría clic sobre la opción “Lista de personas”, lo que le conduciría a la página *PagListado.xaml*. Lo que ocurriría a continuación sería lo ya expuesto en los apartados anteriores. La segunda opción le conduciría al contenido de la web del autor.

Ahora bien, puede suceder que el intento de navegar a un determinado contenido falle porque el ordenador no está en línea o porque el contenido no está disponible. En este caso, se lanzará una excepción, la relacionada con el error ocurrido. Para manipular esta excepción y prevenir que nuestra aplicación se cierre inesperadamente, necesitamos capturarla. Esto lo haremos a través del controlador del evento **NavigationFailed** del objeto **Application** así:

```
Private Sub Application_NavigationFailed(sender As Object,
e As System.Windows.Navigation.NavigationFailedEventArgs)
  If TypeOf e.Exception Is Exception Then
    MessageBox.Show("El URI " & e.Uri.ToString() &
                    " no está disponible.")
    ' Permitir que la aplicación se siga ejecutando
    e.Handled = True
  End If
End Sub
```

También, la página especificada en el objeto **Hyperlink** podría estar en otro ensamblado, esto es, no pertenecer al ensamblado de la aplicación. Por ejemplo, en ocasiones puede resultar útil colocar todas las páginas en una biblioteca, para que sea sencillo reutilizarlas en distintas aplicaciones. En este caso, el URI tiene que especificar la biblioteca (el nombre del ensamblado) a la que pertenece la página. Por ejemplo, el código XAML anterior también podría escribirse así:

```
<Hyperlink NavigateUri="/ApNavegacionWeb;component/PagListado.xaml">
  Lista de personas
</Hyperlink>
```

En este ejemplo observamos que ahora el URI especifica en primer lugar el nombre del ensamblado, *ApNavegacionWeb*, y después la ruta de la página.

En otras ocasiones necesitaremos navegar a una determinada parte de una página con una gran cantidad de contenido. Esto puede hacerlo con la navegación de fragmento: *pagina.axml#nombre_fragmento*, donde *nombre_fragmento* coincidirá con el nombre del control de la página al que se desea navegar. Por ejemplo:

```
<TextBlock> Ir a la
  <Hyperlink NavigateUri="PagListado.xaml#lstPersonas">
    lista
  </Hyperlink> de personas
</TextBlock>
```

Frame

Un marco (**Frame**) proporciona la posibilidad de colocar una página dentro de una ventana o de otra página, aportando flexibilidad porque, de esta forma, pode-

mos reutilizar la misma página de diferentes formas dependiendo del tipo de aplicación que estemos creando.

Para alojar una página dentro de otra o dentro de una ventana, simplemente necesitamos utilizar un objeto de la clase **Frame**. Un **Frame** es un control de contenido que soporta navegación. El contenido puede ser cualquier objeto de .NET Framework y ficheros HTML, sin embargo, la manera preferida de empaquetar contenido para la navegación, como ya hemos visto, son los objetos **Page**. Este control define una propiedad **Source** que hace referencia al identificador de recursos uniforme (URI) del contenido que se desea mostrar.

Por ejemplo, podríamos utilizar un **Frame** para alojar la página menú en un objeto **Window**, en vez de en un **NavigationWindow**:

```
<Window x:Class="MainWindow"
        ...
        Title="Lista de teléfonos" Height="350" Width="525">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Frame BorderBrush="Blue" BorderThickness="1"
           Source="Menu.xaml">
    </Frame>
  </Grid>
</Window>
```

Este código mostraría en el panel de diseño un control **Frame** que incluye, de forma predeterminada, una interfaz de usuario de navegación (su propiedad **NavigationUIVisibility** vale *Visible*), según muestra la figura siguiente:



El ejemplo anterior, utilizando un **Frame**, añade navegación sólo a parte de la ventana, a la correspondiente a la fila 0 del **Grid**.

Otra propiedad de interés de **Frame** es **JournalOwnership**. Esta propiedad permite especificar si **Frame** administra su propio historial de navegación o cede la administración del mismo a un navegador primario (**NavigationWindow** o **Frame**).

Funciones de página

Hasta ahora, hemos visto cómo pasar información a una página (creábamos un objeto de la página, lo configurábamos si era necesario y lo pasábamos como argumento invocando al método **NavigationService.Navigate**), pero no hemos visto cómo retornar información desde una página. En aplicaciones WPF a base de ventanas esto lo hacíamos abriendo un cuadro de diálogo, pero bajo el sistema de navegación de WPF basado en páginas, aplicar esta técnica, si es que se puede (las aplicaciones XBAP no permiten ventanas flotantes), se considera un mal estilo de programación. El mecanismo que WPF proporciona para retornar información desde una página son las *funciones de página*.

Este estilo de navegación es similar a llamar a un método para realizar algún proceso y devolver un valor, lo que se considera programación estructurada. Análogamente, este estilo de navegación se conoce como *navegación estructurada*, y WPF lo proporciona a través de la clase genérica **PageFunction(Of *T*)**.

PageFunction(Of *T*) es una clase genérica que se deriva de **Page** y representa un tipo especial de página en el sentido de que para navegar a ella hay que seguir un protocolo análogo al seguido para llamar a un método. La sintaxis XAML para definir una página de este tipo es la siguiente:

```
<PageFunction x:TypeArguments="T">
  Contenido
</PageFunction>
```

donde *T* es el tipo del valor devuelto por la página; el tipo predeterminado es **String**. Sin embargo, este tipo puede ser fácilmente modificado cambiando el valor del atributo **TypeArguments** en el elemento **<PageFunction>**.

La página que llame a otra de tipo **PageFunction(Of *T*)** puede ser de tipo **Page** o de tipo **PageFunction(Of *T*)**. Haciendo un símil, una página **PageFunction(Of *T*)** es análoga a un diálogo mientras que una página **Page** es análoga a una ventana.

Como ejemplo, vamos a volver sobre la aplicación anterior para añadir a la misma una nueva página de tipo **PageFunction(Of *CPersona*)** que solicite al usuario los datos de una persona que, supuestamente, queremos añadir a nuestra base de datos. Para ello, abra la aplicación y, en primer lugar, añada una nueva clase *CPersona* que represente al objeto que deseamos añadir:

```
Public Class CPersona
  Private m_foto As String
  Private m_nombre As String
  Private m_direccion As String
```

```
Private m_telefono As String
Private es_movil As Boolean

Public Sub New(Optional f As String = Nothing,
Optional n As String = Nothing, Optional d As String = Nothing,
Optional t As String = Nothing, Optional m As Boolean = False)
  m_foto = f
  m_nombre = n
  m_direccion = d
  m_telefono = t
  es_movil = m
End Sub

Public Property Foto() As String
  Get
    Return m_foto
  End Get
  Set(value As String)
    m_foto = value
  End Set
End Property

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

Public Property Direccion() As String
  Get
    Return m_direccion
  End Get
  Set(value As String)
    m_direccion = value
  End Set
End Property

Public Property Telefono() As String
  Get
    Return m_telefono
  End Get
  Set(value As String)
    m_telefono = value
  End Set
End Property
```

```
  Public Property EsMovil() As Boolean
    Get
      Return es_movil
    End Get
    Set(value As Boolean)
      es_movil = value
    End Set
  End Property
End Class
```

Después, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Agregar > Nuevo elemento > Función de página (WPF)*. Denomine a la nueva página *PagFunAgregarPersona*, por ejemplo, y haga clic en el botón *Aceptar*. Modifique el atributo **TypeArguments** para que especifique *CPersona* como el tipo del valor devuelto por la página; esto implica añadir el espacio de nombres correspondiente a esta clase.

```
<PageFunction
    xmlns="http://.../presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:ApNavegacionWeb"
    x:Class="PagFunAgregarPersona"
    x:TypeArguments="local:CPersona"
    xmlns:mc="http://.../markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300"
    WindowTitle="Página agregar persona">
    <Grid>

    </Grid>
</PageFunction>
```

En este código observamos que la página **PageFunction** retornará un objeto de la clase *CPersona*. Esto exige modificar también el fichero de código subyacente en el mismo sentido:

```
Class PagFunAgregarPersona
  Inherits PageFunction(Of CPersona)

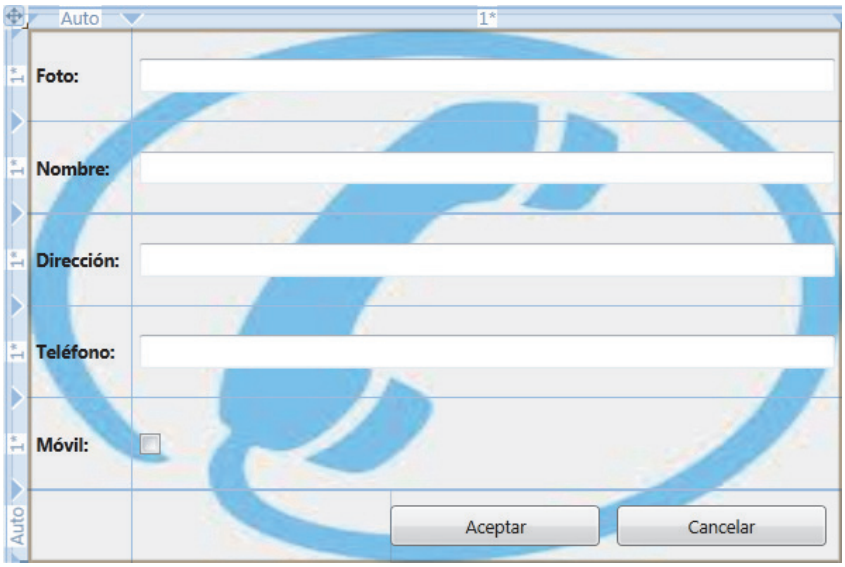
End Class
```

Diseño

El siguiente paso es realizar el diseño de la página *PagFunAgregarPersona* para que permita introducir los datos para una nueva persona; su aspecto puede ser el mostrado en la figura siguiente. Se trata de un diseño en el que observamos una rejilla con dos columnas y seis filas. La columna 0 tiene un ancho *Auto* y la co-

lumna 1 tiene un ancho de * (espacio restante disponible). Las filas 0 a 4 tienen de alto * (reciben una proporción ponderada del espacio restante disponible) y la fila 5 tiene *Auto* de alto (se ajusta al tamaño de su contenido).

En la columna 0, filas 0 a 4, hemos puesto un **Label** que especifique el dato que hay que introducir en el control que hay justo a continuación; en la columna 1, filas 0 a 3, hemos puesto un **TextBox** para introducir los datos del objeto *CPersona* indicados por las etiquetas; en la columna 1, fila 4, hemos puesto un **CheckBox** para introducir el valor del atributo *es_movil* y en las columnas 0 y 1, fila 5, hemos puesto dos controles **Button**, uno para aceptar los datos y otro para cancelar la operación. Finalmente, se ha añadido una imagen como fondo del **Grid** y se ha aplicado estilo a cada uno de los controles.



Observe que la finalidad de esta página es solicitar al usuario los atributos *foto*, *nombre*, *direccion*, *telefono* y *es_movil* de una persona, que, supuestamente, deseamos añadir a la base de datos (en este ejercicio nos limitaremos a escribir, durante el proceso de depuración, los datos obtenidos; más adelante, en otro ejercicio, completaremos la operación).

La técnica que vamos a seguir es definir el nuevo objeto *CPersona* como un atributo de la página, asignarlo al contexto de datos de la página y enlazar sus propiedades con los correspondientes controles de la página.

```
Class PagFunAgregarPersona
  Inherits PageFunction(Of CPersona)
  Private nuevaPersona As New CPersona()
```

```
  Public Sub New()
    InitializeComponent()
    Me.DataContext = nuevaPersona
  End Sub

  ' ...
End Class
```

Según lo expuesto, abra la página *PagFunAgregarPersona.xaml* y agregue el código XAML siguiente entre las etiquetas del **Grid**:

```
<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>

  <Grid.Background>
    <ImageBrush ImageSource="Imagenes/telefono.jpg"/>
  </Grid.Background>

  <TextBlock Grid.Row="0" Grid.Column="0" Text="Foto: "
             Style="{StaticResource EstiloTextBlock}"/>
  <TextBox Name="ctFoto"  Grid.Row="0" Grid.Column="1"
           Style="{StaticResource EstiloTextBox}"
           Text="{Binding Path=Foto}"/>
  <TextBlock Grid.Row="1" Grid.Column="0" Text="Nombre: "
             Style="{StaticResource EstiloTextBlock}"/>
  <TextBox Name="ctNombre" Grid.Row="1" Grid.Column="1"
           Style="{StaticResource EstiloTextBox}"
           Text="{Binding Path=Nombre}"/>
  <TextBlock Grid.Row="2" Grid.Column="0" Text="Dirección: "
             Style="{StaticResource EstiloTextBlock}"/>
  <TextBox Name="ctDireccion" Grid.Row="2" Grid.Column="1"
           Style="{StaticResource EstiloTextBox}"
           Text="{Binding Path=Direccion}"/>
  <TextBlock Grid.Row="3" Grid.Column="0" Text="Teléfono: "
             Style="{StaticResource EstiloTextBlock}"/>
  <TextBox Name="ctTelefono" Grid.Row="3" Grid.Column="1"
           Style="{StaticResource EstiloTextBox}"
           Text="{Binding Path=Telefono}"/>
```

```
  <TextBlock Grid.Row="4" Grid.Column="0" Text="Móvil: "
             Style="{StaticResource EstiloTextBlock}"/>
  <CheckBox Name="cvEsMovil" Grid.Row="4" Grid.Column="1"
            Style="{StaticResource EstiloControl}"
            Text="{Binding Path=EsMovil}"/>
  <WrapPanel Grid.Row="5" Grid.ColumnSpan="2"
             HorizontalAlignment="Right">
    <Button Name="btAceptar" Content="Aceptar"
            Style="{StaticResource EstiloButton}"
            Click="btAceptar_Click" />
    <Button Name="btCancelar" Content="Cancelar"
            Style="{StaticResource EstiloButton}"
            Click="btCancelar_Click" />
  </WrapPanel>
</Grid>
```

Lógica de negocio

Una función de página tiene que manejar toda su navegación desde el código subyacente. Para ello, la clase **PageFunction** proporciona la funcionalidad siguiente:

- Cuando la función de página finaliza su proceso, tiene que llamar a su método **OnReturn** para volver a la página que la llamó (que la invocó).

- **OnReturn** acepta un parámetro de tipo **ReturnEventArgs(Of** *T***)** que se utiliza para devolver un valor a través de su propiedad **Result**. Este parámetro será **Nothing** cuando no se desee devolver ningún valor.

- Para obtener el valor que la página devolvió, hay que controlar el evento **Return**.

Resumiendo, para devolver un valor, la función de página crea un objeto **ReturnEventArgs(Of** *T***)** pasando como argumento el valor a devolver que quedará referenciado por la propiedad **Result** de ese objeto (en nuestro caso, el valor a devolver es el objeto *CPersona* que hemos construido con los datos introducidos) y llama a **OnReturn** para volver a la página que la llamó. Según lo expuesto, los controladores de los botones "Aceptar" y "Cancelar" serán así:

```
Private Sub btAceptar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Aceptar la información introducida
  OnReturn(New ReturnEventArgs(Of CPersona)(nuevaPersona))
End Sub

Private Sub btCancelar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  ' Operación cancelada
```

```
  OnReturn(Nothing)
End Sub
```

Después, la página de llamada puede recuperar ese valor del objeto **ReturnEventArgs(Of *T*)** por medio del segundo parámetro del controlador del evento **Return** que genera el método **OnReturn**.

En nuestro ejemplo, la página de llamada es *PagListado*. Para navegar desde esta página a la página *PagFunAgregarPersona*, añada a *PagListado* un nuevo botón "Agregar persona" y edite el controlador de su evento **Click** así:

```
Private Sub btAgregar_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim pagFunAgregarPersona As New PagFunAgregarPersona()
  AddHandler pagFunAgregarPersona.Return,
             AddressOf pagFunAgregarPersona_Return
  Me.NavigationService.Navigate(pagFunAgregarPersona)
End Sub
```

Este método se ejecutará cuando el usuario haga clic en el botón "Agregar persona" de *PagListado*. A partir de este instante los hechos se suceden así: se navega a la página *PagFunAgregarPersona*, el usuario introduce en esta página los datos correspondientes a una persona y hace clic en el botón "Aceptar" o "Cancelar", instante en el que se ejecuta el método **OnReturn**, que, entre otras cosas, genera el evento **Return**, y se vuelve a la página *PagListado*.

Entonces, para recuperar la información introducida, la página de llamada, *PagListado*, tiene que controlar ese evento **Return**, de ahí que hayamos añadido en el controlador del evento **Click** del botón "Agregar persona" el código que vincula ese evento con el método que se tiene que ejecutar cuando se produzca.

Según lo expuesto, sólo queda añadir a *PagListado* el controlador *pagFunAgregarPersona_Return* del evento **Return**. Este controlador tiene un segundo parámetro de tipo **ReturnEventArgs(Of *T*)** con una propiedad **Result** que proporciona la información devuelta por la página de función. La función de este controlador sería agregar la persona, *e.Result*, a la base de datos, pero, de momento, nos limitaremos a mostrar en la ventana de resultados del entorno de desarrollo, los atributos del nuevo objeto *CPersona*:

```
Private Sub pagFunAgregarPersona_Return(sender As Object,
e As ReturnEventArgs(Of CPersona))
  ' Aquí se agregaría la persona "e.Result" a la base de datos
  If e IsNot Nothing Then
    System.Diagnostics.Debug.WriteLine(e.Result.Foto)
    System.Diagnostics.Debug.WriteLine(e.Result.Nombre)
    System.Diagnostics.Debug.WriteLine(e.Result.Direccion)
```

```
    System.Diagnostics.Debug.WriteLine(e.Result.Telefono)
    System.Diagnostics.Debug.WriteLine(e.Result.EsMovil)
  End If
End Sub
```

De forma predeterminada, se conserva un objeto **PageFunction(Of *T*)** en el historial de navegación después de que la función de página finaliza su proceso, esto es, cuando **OnReturn** finaliza, lo que posibilita al usuario navegar a la función de página. Si no queremos que esto suceda, podemos eliminar ese objeto del historial estableciendo su propiedad **RemoveFromJournal** a **True**.

APLICACIÓN XBAP

El desarrollo de aplicaciones XBAP es similar al desarrollo de aplicaciones cliente enriquecidas y, al igual que las aplicaciones webs, se pueden instalar en un servidor web e iniciarse con Internet Explorer o Firefox.

La forma más sencilla de crear una aplicación XBAP es utilizando la plantilla *Aplicación de explorador WPF* de Visual Studio 2010. Por ejemplo, utilizando esta plantilla, cree una nueva aplicación WPF denominada *AplicacionXBAP*. Analicemos a continuación el código generado:

```
<Page x:Class="Page1"
      xmlns="http://.../xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://.../markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      Title="Page1">
    <Grid>

    </Grid>
</Page>
```

Este código nos muestra que una aplicación XBAP está basada en un objeto **Page**, una página, a diferencia de una aplicación WPF enriquecida que está basada en un objeto **Window**. Esto es, por cada página perteneciente a la aplicación crearemos un objeto **Page** que, normalmente, estará soportado por el fichero de código XAML correspondiente y por el fichero de código Visual Basic subyacente. Las líneas de color gris pueden ser eliminadas, sólo sirven para trabajar durante el diseño, en este caso, para limitar el tamaño de la página en el diseñador. Los elementos identificados como **mc:Ignorable** no generarán errores al procesarlos en un procesador XAML porque si no se pueden resolver como un tipo subyacen-

te o una construcción de programación, se omiten; esto es lo que ocurre en el ejemplo con los elementos pertenecientes al espacio de nombres **d**.

Si establece las propiedades **Width** y **Height** de **Page**, la página mostrará el diseño realizado con esas medidas centrado en la página.

Ahora observemos el código que inicia la aplicación, fichero *Application.xaml*:

```
<Application x:Class="Application"
             xmlns="http://.../xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="Page1.xaml">
    <Application.Resources>

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

Vemos que, en este caso, la propiedad **StartupUri**, que establece la interfaz gráfica que se muestra automáticamente cuando se inicia una aplicación, tiene asignada la página *Page1*.

Cuando ejecute esta aplicación, simplemente se mostrará la página *Page1* en el explorador. Para comprobarlo, complete el código XAML de *Page1.xaml* así:

```
<Grid>
  <TextBlock>Esta es la página 1</TextBlock>
</Grid>
```

Finalmente, abra su sistema de ficheros y observe la carpeta donde se han generado los ficheros resultantes de la compilación. Observará un fichero ejecutable (*.exe*) que contiene el código compilado, el manifiesto de la aplicación (*.manifest*) que contiene los metadatos asociados a la aplicación y el manifiesto de despliegue de la aplicación (*.xbap*) que contiene la información que *ClickOnce* emplea para descargar e iniciar la aplicación basada en Windows ubicada en un servidor web, que en el caso de Windows es IIS (*Internet Information Services*).

Una vez creado el esqueleto de la aplicación XBAP, completar dicha aplicación no difiere de lo estudiado en los apartados anteriores, cuando utilizamos la clase **NavigationWindow**.

Publicar la aplicación

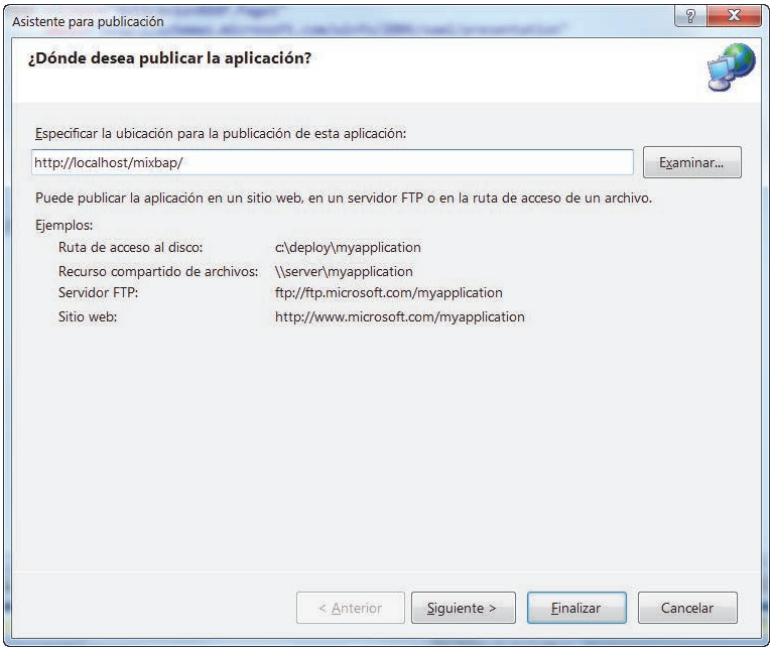
Publicar una aplicación XBAB es parecido a publicar una aplicación web; significa poner la aplicación en un servidor de Internet a disposición de los usuarios de

la red mundial, aunque el escenario más común para este tipo de aplicaciones es Intranet. Para ello, la máquina que va a trabajar como servidor debe proporcionar un sitio web. ¿Cómo se crea un sitio web? Cuando se instala IIS en Windows se configura un sitio web de manera predeterminada. Este sitio se localiza en *C:\Inetpub\wwwroot*.

Entonces, para publicar una aplicación XBAP, simplemente hay que moverla al directorio *\Inetpub\wwwroot*. Pero, evidentemente, en ese servidor habrá más de una aplicación instalada; por eso, lo normal es que cada aplicación esté contenida en su propio directorio, por ejemplo en *\Inetpub\wwwroot\mixbap*.

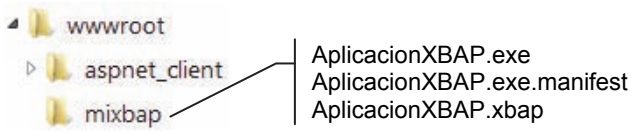
Pero IIS requiere que este directorio sea un directorio virtual vinculado con el directorio físico de su sistema de ficheros que contiene la aplicación web. Esto es, un directorio virtual se utiliza para incluir en IIS el contenido de la aplicación web sin tener que mover o copiar los ficheros desde el directorio físico de la aplicación al directorio raíz del servidor web. No obstante, es bastante habitual que ese directorio físico sea un subdirectorio de *\Inetpub\wwwroot* con el mismo nombre que el directorio virtual, quedando ambos, físico y virtual, solapados.

Todo el proceso descrito resultará mucho más sencillo si utilizamos el *asistente para publicación* que proporciona Visual Studio. Para lanzar este asistente, haga clic con el botón secundario del ratón sobre el nombre del proyecto y elija la opción *Publicar* (tiene que haber iniciado Visual Studio como administrador). Después siga los pasos indicados por el asistente.



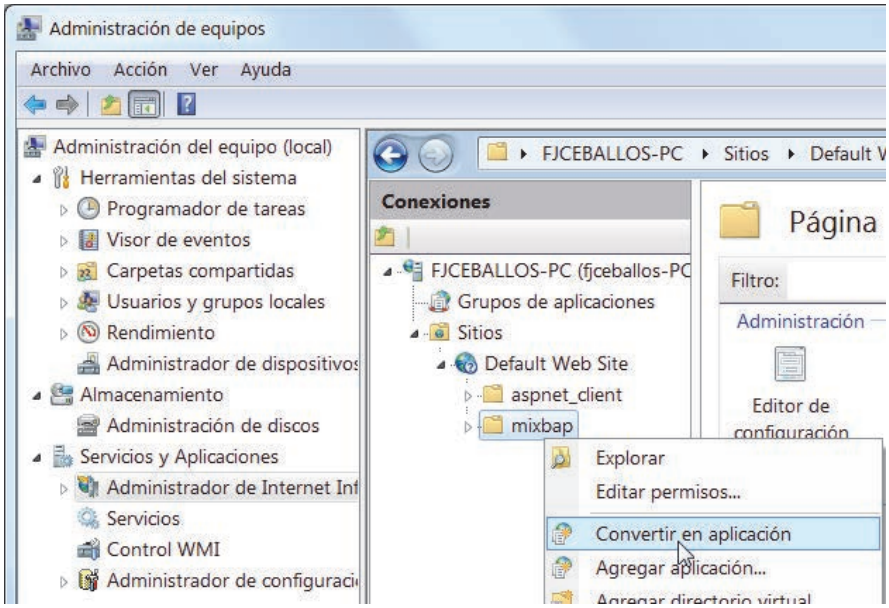
Cuando haya finalizado el trabajo con el asistente, podrá verificar que se ha creado una carpeta (en el ejemplo *mixbap*) virtual vinculada con la carpeta física que contiene los ficheros que componen la aplicación XBAP. También comprobará que se ha creado un fichero *publish.htm* que lanzará la ejecución de la aplicación en el explorador.

Veamos cómo realizar este proceso manualmente, lo que nos ayudará a entenderlo mejor. Para preparar la aplicación XBAP para su despliegue, copie el fichero *.exe* y los manifiestos asociados (*.manifest* y *.xbap*) en una carpeta en el sitio web predeterminado (*Inetpub\wwwroot*). Por ejemplo, la figura siguiente muestra cómo publicar la aplicación *AplicacionXBAP* en el sitio web predeterminado, en una carpeta *mixbap*:



El fichero *.exe* contiene el código compilado (código IL) como cualquier otra aplicación .Net. El fichero XML *.exe.manifest* especifica los requerimientos de la aplicación y el fichero *.xbap* es otro fichero XML que especifica el punto de entrada a la aplicación; en otras palabras, éste es el fichero que el usuario necesita solicitar en el explorador para instalar la aplicación XBAP.

Después, complete esta tarea especificando el nombre del directorio virtual. Para ello, abra el administrador de IIS, seleccione la carpeta *mixbap* en el sitio web, haga clic sobre ella con el botón secundario del ratón y seleccione la opción *Convertir en aplicación* y acepte los valores predeterminados.



Para terminar, opcionalmente, cree en la misma carpeta *mixbap* una página HTML que contenga un hipervínculo para abrir el manifiesto de despliegue (fichero con la extensión *.xbap*). Otra opción sería utilizar un **<iframe>** dentro de la página HTML. Por ejemplo, para la aplicación anterior esta página HTML podría ser así:

```
<html>
  <head></head>
  <body>
    <a href="AplicacionXBAP.xbap">
      Clic aquí para ejecutar la aplicación
    </a>
  </body>
</html>
```

Guarde el código, por ejemplo, en un fichero denominado *publish.htm*. Ahora puede abrir el explorador y ejecutar la aplicación XBAP utilizando la siguiente URL (el nombre del servidor en su máquina será *localhost*):

```
http://miservidor/mixbap/publish.htm
```

Evidentemente, puede invocar directamente al fichero *.xbap* así:

```
http://miservidor/mixbap/AplicacionXBAP.xbap
```

Cuando se lanza la ejecución de la aplicación, observará una barra de progreso mientras se descarga el fichero *.xbap*; esto se corresponde esencialmente con la

instalación de la aplicación que copia dicho fichero en la caché de aplicaciones descargadas.

Cuando desde Visual Studio estamos depurando una aplicación XBAP, cada vez que la compilamos y ejecutamos, Visual Studio siempre vuelve a cargar la última versión en el explorador. Pero esto no sucede así si solicitamos la ejecución de la aplicación, evidentemente después de volverla a publicar, directamente desde el explorador; en este caso, se ejecuta la versión anterior. Esto es debido a que el número de versión entre la versión almacenada en memoria caché (la versión anterior) y la nueva versión sigue siendo el mismo, y no se descarga la nueva versión, sino que, en su lugar, se carga la versión almacenada en la caché. La solución a este problema es limpiar la caché de las aplicaciones descargadas ejecutando la siguiente orden desde la línea de órdenes:

```
mage.exe -cc
```

La utilidad **mage.exe** se localiza en la instalación de Visual Studio o Windows SDK de su ordenador.

Finalmente, indicar que cuando se crea una nueva aplicación XBAP, Visual Studio también incluye un archivo de certificado generado automáticamente con un nombre como *ApplicationNombreAp_TemporaryKey.pfx*. Este certificado contiene un par de claves pública/privada que se utilizan para añadir una firma al fichero *.xbap*. Cuando se publica una actualización de la aplicación, habrá que firmar con la misma clave para asegurar que la firma digital se mantiene constante. Si desea utilizar una clave propia en lugar de utilizar la clave temporal, puede hacerlo a través de la página *Firma* de las propiedades del proyecto.

Seguridad

De manera predeterminada, las aplicaciones XBAP se ejecutan en un recinto de seguridad de confianza parcial, que se limita al conjunto de permisos predeterminado de la zona Internet (las limitaciones son comparables a las impuestas al código JavaScript que pueda aparecer en una página HTML). Con ello, las aplicaciones XBAP se aíslan de manera eficaz del equipo cliente. No obstante, una aplicación XBAP puede elevar los privilegios hasta plena confianza dependiendo de la zona de seguridad de la dirección URL de implementación y la configuración de seguridad del cliente. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Propiedades > Seguridad*. Después, especifique los permisos de seguridad de acceso del código que requiere la aplicación *ClickOnce* para poderse ejecutar.

No obstante, en general, si no es estrictamente necesario, no se recomienda conceder permisos de plena confianza por dos razones importantes:

- Una aplicación con permisos de plena confianza tiene acceso sin restricciones a los recursos, por ejemplo, al sistema de ficheros y al registro, lo que la hace que sea susceptible a un ataque de código malintencionado.

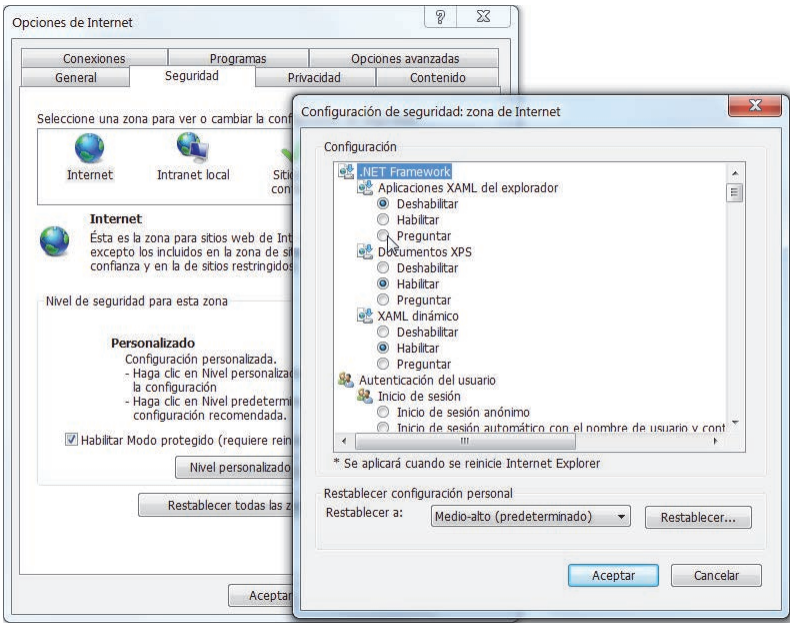
- Cuando una aplicación requiera permisos de plena confianza, se le solicitará al usuario final que conceda estos permisos a la aplicación.

Por estas razones, incluya sólo los permisos requeridos por la aplicación. La página *Seguridad* de las propiedades del proyecto contiene una casilla *Habilitar configuración de seguridad de ClickOnce*. Cuando esta casilla está activada, las solicitudes de permiso de seguridad se agregan al manifiesto de despliegue de la aplicación (*.xbap*). Por lo tanto, si durante la instalación se solicitaran más permisos de los predeterminados para la zona donde se despliega la aplicación, se le pedirá al usuario que conceda los permisos necesarios.

Según lo expuesto, si publicamos la aplicación XBAP en un servidor remoto, puede suceder que, cuando un usuario intente ejecutarla desde su navegador,

```
http://servidor_remoto/mixbap/publish.htm
```

obtenga un error y no la pueda ejecutar, simplemente por temas de seguridad (de forma predeterminada la ejecución está sólo habilitada para la zona de Intranet). Esto es, la configuración de seguridad de nuestro equipo determina el acceso que se permite a cualquier software de exploración web. Por ejemplo, Internet Explorer permite configurar las funciones que pueden ser ejecutadas por él para cada una de las zonas (Internet, Intranet, sitios de confianza y sitios restringidos). Según esto, podemos permitir ejecutar una aplicación XBAP publicada en un servidor remoto si configuramos la zona de Internet así: abra la ventana de opciones de Internet mostrada en la figura siguiente y seleccione *Seguridad > Nivel personalizado > .NET Framework > Aplicaciones XAML de explorador > Habilitar|Preguntar*.



ACCESO A UNA BASE DE DATOS DESDE UNA XBAP

Para ver de una forma más real lo que realmente podemos hacer con las aplicaciones XBAP, vamos a realizar una aplicación de este tipo que trabaje con los datos de una base de datos SQL Server. Dicha base de datos tendrá una tabla de personas definidas por los atributos: identificador de la persona, nombre, dirección, teléfono, observaciones y foto. La aplicación mostrará persona a persona en una página los datos de las personas almacenadas en la base de datos, permitirá añadir nuevas personas, borrarlas o modificarlas, y añadirá posibilidades de clasificación y filtrado de la información mostrada.

Crear la base de datos

Para empezar, construiremos la base de datos *bd_telefonos2* con una tabla *telefonos* definida con los campos mostrados a continuación. El campo *idpersona* será la clave primaria, entero, empezará en 1 y será autoincrementado de 1 en 1:

```
CREATE TABLE telefonos (
  idpersona     INTEGER IDENTITY(1,1) PRIMARY KEY,
  nombre        VARCHAR(30) NOT NULL,
  direccion     VARCHAR(30) NOT NULL,
  telefono      VARCHAR(12) NOT NULL,
  observaciones VARCHAR(240),
  foto          IMAGE )
```

	Nombre de columna	Tipo de datos	Permitir valores NULL
🔑	idpersona	int	☐
	nombre	varchar(30)	☐
	direccion	varchar(30)	☐
	telefono	varchar(12)	☐
	observaciones	varchar(240)	☑
	foto	image	☑

Una vez construida la base de datos, utilizando *Entity Framework*, asignaremos el modelo de datos relacional a un modelo de objetos expresado en el lenguaje de programación Visual Basic, para después realizar las consultas sobre este modelo de objetos. Estas consultas serán convertidas por *Entity Framework* a SQL y enviadas a la base de datos para su ejecución. Cuando la base de datos devuelva los resultados, *Entity Framework* los vuelve a convertir en objetos expresados en el propio lenguaje Visual Basic.

El siguiente paso es crear la estructura básica de la aplicación XBAP que accederá a la base de datos. Para ello, según explicamos anteriormente, abra Visual Studio y cree un nuevo proyecto de tipo *Visual Basic > Windows > Aplicación de explorador WPF*, denominado *ApXBAPtelefonos*.

Conectarse a la base de datos

Abra el *Explorador de servidores* (*Explorador de bases de datos*) y añada una conexión a la base de datos *bd_telefonos2* (este proceso ya fue explicado en el capítulo *Acceso a una base de datos*).

Generar el modelo de entidades

Según lo estudiado en los capítulos anteriores, un enfoque adecuado en el desarrollo de aplicaciones es separar la capa de acceso a datos (DAL) de la capa de presentación. Entonces, como siguiente paso, añada un nuevo proyecto *Biblioteca de clases* a la solución *ApXBAPtelefonos*, por ejemplo *BDtelefonos*, y a continuación agregue a este proyecto un EDM denominado *MDEBDtelefonos*.

Como ya estudiamos, para abrir el *asistente para el EDM* agregue un nuevo elemento de tipo *ADO.NET Entity Data Model* al proyecto *BDtelefonos*. En nuestro caso denominaremos al modelo EDM *MDEBDtelefonos.edmx* y se generará desde la base de datos *bd_telefonos2*, tabla *telefonos*. Cuando hayamos finalizado este proceso, la ventana de diseño de Visual Studio mostrará el *diseñador de enti-*

dades que muestra la figura siguiente, donde se observa la clase *telefono* con los atributos correspondientes a las columnas de cada fila de la tabla *telefonos*:



De forma predeterminada, *Entity Framework* crea la lógica para actualizar una base de datos (inserciones, actualizaciones y eliminaciones) con los cambios realizados en los datos de las clases de entidad, basándose en el esquema de la tabla (información de columna y de clave principal).

Para poder utilizar este modelo de entidades en el proyecto *ApXBAPtelefonos*, añada al mismo una referencia a la biblioteca de clases *BDtelefonos*. También hay que copiar en el proyecto *ApXBAPtelefonos* el fichero de configuración *App.Config* del proyecto *BDtelefonos*, que contiene la cadena de conexión para acceder a la base de datos, y añadir una referencia a **System.Data.Entity**.

Finalmente, asigne a la aplicación XBAP permisos de plena confianza. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Propiedades* > *Seguridad* > *Aplicación de plena confianza*. De esta forma, la aplicación tendrá acceso sin restricciones a los recursos, por ejemplo, al sistema de ficheros, con lo que podrá acceder a la base de datos.

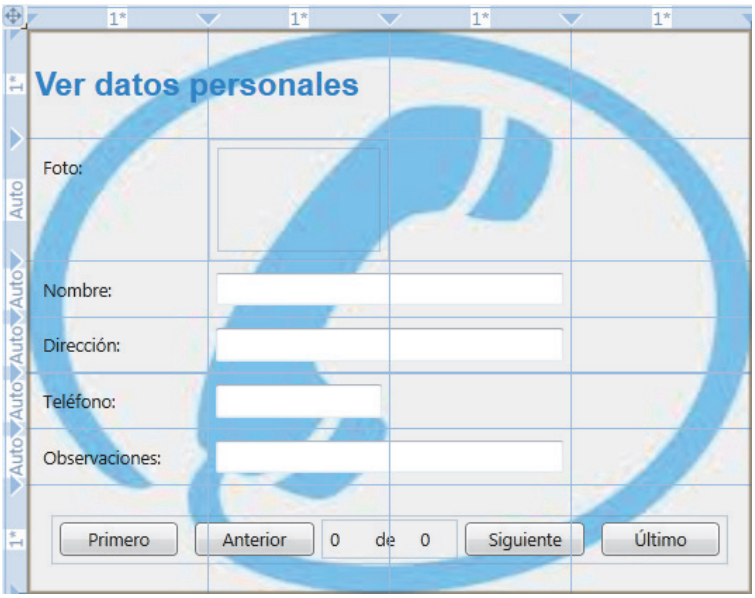
Una vez construido el modelo de entidades, el siguiente paso es diseñar la interfaz gráfica de la aplicación *ApXBAPtelefonos*. Los controles de esta interfaz se proveerán de datos de este modelo de entidades.

Interfaz gráfica

La aplicación, inicialmente, mostrará una interfaz que permita mostrar uno a uno los datos de cada una de las personas.

El proyecto contiene ya una página denominada *Page1*. Cambie este nombre para que ahora la página se denomine *PagMostrar*; haga este cambio también en el fichero de código subyacente y en el fichero *Application.xaml*.

El siguiente paso es realizar el diseño de la página *PagMostrar* que va a mostrar los datos de la persona a la cual navegue el usuario por medio de los botones de avanzar y retroceder que colocaremos en esta página. Para que el diseño tome el aspecto mostrado en la figura siguiente configure la rejilla con cuatro columnas y seis filas y, después, coloque los controles como los ve en la figura.



No abundaremos en más explicaciones sobre este diseño, puesto que es similar a los realizados anteriormente en este mismo capítulo.

Vincular controles con el origen de datos

Ya tenemos generado el modelo de entidades que representa la base de datos, esto es, la capa de acceso a datos (DAL), y diseñada la interfaz gráfica. Ahora, partiendo de este modelo, crearemos un proveedor de datos para utilizarlo como origen de datos en la página *PagMostrar*. Para ello, añada una nueva carpeta al proyecto *ApXBAPtelefonos*, denominada por ejemplo *BLL*, y agregue a la misma la siguiente clase:

```
Imports BDtelefonos

Public Class CProveedorDatosTelefono
  Private m_ContextoDeObjs As bd_telefonos2Entities
  Public Property ContextoDeObjs() As bd_telefonos2Entities
    Get
      Return m_ContextoDeObjs
    End Get
```

```
    Private Set(value As bd_telefonos2Entities)
      m_ContextoDeObjs = value
    End Set
  End Property

  Public Sub New()
    ContextoDeObjs = New bd_telefonos2Entities()
  End Sub

  Public Function ObtenerTelefonos() As List(Of telefono)
    ' Consulta LINQ
    Dim tfnos = From tfno In ContextoDeObjs.telefonos
                Select tfno
    Return tfnos.ToList()
  End Function
End Class
```

Nota: durante el diseño se produce un error debido a que la aplicación es el propio Visual Studio y su fichero de configuración, obviamente, no contiene la cadena de conexión especificada en *App.Config*. Ver la solución en el código proporcionado en el CD, ya que se sale del contexto de nuestra exposición.

La variable *tfnos* hace referencia a un objeto que implementa la interfaz **IQueryable(Of** *alumno***)** y **ToList** devuelve una colección de tipo **List(Of** *telefono***)**.

Un objeto de la clase **ObjectDataProvider** será el proveedor de datos de las páginas de esta aplicación. Para ello lo construiremos como un recurso *pdTelefonos* de la aplicación, según se indica a continuación:

```
<Application.Resources>
  <ObjectDataProvider x:Key="pdTelefonos"
    ObjectType="{x:Type local:CProveedorDatosTelefono}"
    MethodName="ObtenerTelefonos">
  </ObjectDataProvider>
</Application.Resources>
```

Dicho objeto encapsula la lista de teléfonos (colección **List(Of** *teléfono***)**) devuelta por el método *ObtenerTelefonos* de la clase *CProveedorDatosTelefono*. Esto implica añadir también el espacio de nombres correspondiente a esta clase:

```
<Application x:Class="Application"
             xmlns="http://.../presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ApXBAPtelefonos"
             StartupUri="PagMostrar.xaml">
```

Esa colección, **List(Of** *teléfono***)**, referenciada por la propiedad **Data** del **ObjectDataProvider**, serán los datos sobre los que operarán las páginas de la aplicación. Entonces, para que la página *PagMostrar* pueda mostrar los objetos *telefono* de la colección, asignaremos a su contexto de datos el objeto proveedor de los datos dado por el recurso *pdTelefonos* así:

```
Class PagMostrar
  Private odp As ObjectDataProvider = Nothing
  Private Sub Page_Loaded(sender As Object, e As RoutedEventArgs)
    odp = DirectCast(Me.FindResource("pdTelefonos"), ObjectDataProvider)
    Me.DataContext = odp
  End Sub
End Class
```

Un breve vistazo a la clase *CProveedorDatosTelefono* nos recuerda que el recurso *pdTelefonos* proporciona la colección de teléfonos obtenida de la tabla *telefonos* por medio de una consulta LINQ que realiza el método *ObtenerTelefonos* de dicha clase. Para que los controles de la página *pagMostrar* muestren los datos *nombre*, *direccion*, *telefono1*, *observaciones* y *foto* de cada uno de los objetos *telefono* de la colección, hemos asignado dicho proveedor al contexto de datos de la página, para así poder vincular los atributos de esos objetos con sus respectivos controles en la página:

```
<Image Grid.Column="1" Grid.Row="1" Name="imFoto"
       Height="64" Margin="5"
       Source="{Binding Path=foto}"/>
<TextBox Grid.Column="1" Grid.Row="2" Grid.ColumnSpan="2"
         Name="ctNombre" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=nombre}"/>
<TextBox Grid.Column="1" Grid.Row="3" Grid.ColumnSpan="2"
         Name="ctDireccion" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=direccion}"/>
<TextBox Grid.Column="1" Grid.Row="4" Name="ctTelefono"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=telefono1}"/>
<TextBox Grid.Column="1" Grid.Row="5" Grid.ColumnSpan="2"
         Name="ctObservaciones"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=observaciones}"/>
```

Si ejecuta ahora la aplicación, observará cómo la ventana muestra ya los datos del primer objeto de la colección.

El siguiente paso es programar los controladores de los botones “Primero”, “Anterior”, “Siguiente” y “Último” para que muestren el objeto de la colección acorde a lo especificado por su título. Estas operaciones, según vimos en el capítulo *Enlace de datos y WPF*, son proporcionadas por la vista de la colección de

objetos. Entonces, para facilitar que cada controlador tenga acceso a esa vista, añada el siguiente método a la clase *PagMostrar*:

```
Private Function ObtenerVista() As ICollectionView
  Return CollectionViewSource.GetDefaultView(odp.Data)
End Function
```

A continuación, añada los controladores de los botones citados anteriormente:

```
<WrapPanel Grid.ColumnSpan="4" Grid.Row="6"
           Style="{StaticResource EstiloWrapPanel}">
  <Button Name="btPrimero" Content="Primero" Margin="5"
          Click="btPrimero_Click" />
  <Button Name="btAnterior" Content="Anterior" Margin="5"
          Click="btAnterior_Click" />
  <WrapPanel VerticalAlignment="Center" ItemWidth="28">
    <TextBlock Name="btxPosicionReg" Text="0" Margin="5"/>
    <TextBlock Text="de" Margin="5"/>
    <TextBlock Name="btxTotalReg" Text="0" Margin="5"/>
  </WrapPanel>
  <Button Name="btSiguiente" Content="Siguiente" Margin="5"
          Click="btSiguiente_Click" />
  <Button Name="btUltimo" Content="Último" Margin="5"
          Click="btUltimo_Click" />
</WrapPanel>
```

Después, edite estos controladores como se indica a continuación:

```
Private Sub btPrimero_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToFirst()
End Sub

Private Sub btAnterior_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToPrevious()
  If vista.IsCurrentBeforeFirst Then vista.MoveCurrentToFirst()
End Sub

Private Sub btSiguiente_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToNext()
  If vista.IsCurrentAfterLast Then vista.MoveCurrentToLast()
End Sub

Private Sub btUltimo_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  vista.MoveCurrentToLast()
End Sub
```

Si ahora ejecuta la aplicación, observará cómo la ventana permite ya mostrar, uno a uno, los datos de todos los objetos de la colección.

El siguiente paso es hacer que los controles de texto que están entre estos botones "Anterior" y "Siguiente" indiquen la posición del registro actual y el número total de registros. Esta operación hay que realizarla cuando se carga la página y cada vez que cambia el elemento actual de la vista, que está referenciado por la propiedad **CurrentItem**. Para ello, debemos saber que cada vez que cambia el elemento actual de la vista se genera el evento **CurrentChanged**. Según esto, vamos a añadir el controlador *vista_CurrentChanged* para manejar este evento. Lo haremos cuando se cargue la página *PagMostrar*:

```
Private Sub Page_Loaded(sender As Object, e As RoutedEventArgs)
  odp = DirectCast(Me.FindResource("pdTelefonos"), ObjectDataProvider)
  Me.DataContext = odp

  ' Manejar el evento CurrentChanged de la vista
  Dim vista As ICollectionView = ObtenerVista()
  AddHandler vista.CurrentChanged, AddressOf vista_CurrentChanged
  ActualizarPosicionReg(TryCast(vista, CollectionView))
End Sub

Private Sub vista_CurrentChanged(sender As Object, e As EventArgs)
  ActualizarPosicionReg(TryCast(sender, CollectionView))
End Sub

Private Sub ActualizarPosicionReg(vista As CollectionView)
  If vista.CurrentItem Is Nothing Then Return
  Dim posicionActual As String =
    Convert.ToString(vista.CurrentPosition + 1)
  Me.btxPosicionReg.Text = posicionActual
  Me.btxTotalReg.Text = vista.Count.ToString()
End Sub
```

En el código anterior podemos ver que es el método *ActualizarPosicionReg* el que actualiza los controles de texto que indican la posición del registro actual y el número total de registros. Este método se invoca cuando se carga la página por primera vez y cada vez que nos movemos a otro registro.

Controles de usuario

Un control de usuario es un componente reutilizable que se puede agregar a la barra de herramientas como cualquier control. Puede crearlo con Visual Studio o con Expression Blend y puede contener otros controles, recursos y animación, igual que una aplicación. La única diferencia es que el elemento raíz tiene que ser un objeto **UserControl** en lugar de un objeto **Window** o **Page**.

Un control de usuario puede ser un fichero *.dll* (una biblioteca) o un fichero *.xaml*. En el primer caso tendrá que crear un proyecto de tipo *Biblioteca de controles de usuario de WPF* y añadir una referencia a esta biblioteca en el proyecto donde se vaya a usar. En el segundo caso tendrá que añadir al proyecto sobre el que está trabajando un nuevo elemento de tipo **UserControl**; esta opción es más flexible porque el control de usuario está en el mismo proyecto en el que se usa y los cambios se pueden realizar al instante.

Según lo expuesto, vamos a crear un control de usuario XAML para utilizarlo como filtro para los elementos de la colección de objetos *telefono*. El control, según muestra la figura siguiente, estará formado por un **Border** azul, un **Label** (Filtrar teléfonos:) y un **TextBox** en el que el usuario escribirá el patrón, los dígitos, que deben contener los teléfonos (campo *telefono1*) que deseamos filtrar.



Para ello, añada una nueva carpeta al proyecto denominada, por ejemplo, *ControlesUsuario*. Después, haga clic con el botón secundario del ratón sobre esta carpeta y seleccione *Agregar* > *Control de usuario*. Finalmente, diseñe el control. El resultado podría ser parecido al siguiente:

```
<UserControl
 x:Class="ApXBAPtelefonos.ControlesUsuario.UControlFiltrar"
             xmlns="http://.../presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://.../markup-compatibility/2006"
             xmlns:d="http://.../expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="50" d:DesignWidth="250">
```

```
  <Border Margin="5" Padding="3" Background="#88CCFF"
          CornerRadius="5" >
    <DockPanel>
      <Label DockPanel.Dock="Left" Content="Filtrar teléfonos:"
             FontWeight="Bold" Foreground="White" />
      <TextBox x:Name="ctFiltrar"
               Background="White" Margin="4" />
    </DockPanel>
  </Border>
</UserControl>
```

El fichero de código subyacente permanece como está. Observe que el control pertenece al espacio de nombres *ApXBAPtelefonos*.

El paso siguiente es añadirlo a la página *PagMostrar*. Esto implica especificar también el espacio de nombres al que pertenece el control:

```
<Page x:Class="PagMostrar"
      ...
      xmlns:uc="clr-namespace:ApXBAPtelefonos"
      ... >
  <Grid>
    ...
    <uc:UControlFiltrar x:Name="ucFiltrar"
          Grid.Column="2" Grid.ColumnSpan="2" Grid.Row="4"
          TextBoxBase.TextChanged="ucFiltrar_TextChanged"/>
  </Grid>
</Page>
```

Cuando en XAML sea necesario tener acceso mediante código a un elemento que no tiene una propiedad **Name**, debe especificar **x:Name**. No es nuestro caso, pero ahí queda a título de ejemplo.

La interfaz gráfica ya está terminada. Siguiendo con el desarrollo de la aplicación, ¿qué código y dónde tenemos que escribirlo para filtrar los datos? Sabemos que una vista tiene la capacidad de filtrar los elementos provenientes de la colección subyacente por medio de su propiedad **Filter**. ¿Dónde establecemos este filtro? Pues en el controlador del evento adjunto **TextBoxBase.TextChanged** del control de tipo *UControlFiltrar*, según puede ver en el código anterior; así podremos capturar el texto introducido cada vez que cambia el contenido de su caja de texto.

Agregue este controlador al fichero de código subyacente de *PagMostrar* y edítelo como se indica a continuación:

```
Private Sub ucFiltrar_TextChanged(sender As Object,
e As TextChangedEventArgs)
```

```
  Dim ct As TextBox = TryCast(e.OriginalSource, TextBox)
  If ct Is Nothing Then Return
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CanFilter Then
    vista.Filter = Function(elemento As Object) _
      DirectCast(elemento, telefono).telefono1.Contains(ct.Text)
  End If
End Sub
```

Para especificar las condiciones que deben cumplir los objetos que pertenecerán a la colección filtrada, hay que vincular la propiedad **Filter** de la vista con el método que se utilice para especificar tales condiciones. Como esta propiedad es de tipo **Predicate(Of Object)** podemos implementar un método anónimo acorde al delegado **Predicate** (*Boolean Predicate(T obj)*), según se observa en el código anterior. Observe que la condición es que la subcadena *ct.Text* esté contenida en *telefono1*; todos los elementos que cumplan esta condición pertenecerán a la colección filtrada. Observe también el uso de la propiedad **OriginalSource** para obtener el elemento original donde se desencadenó el evento, que es la caja de texto que contiene el control de usuario referenciado por *sender*.

Modificar registros

Cuando ejecuta la aplicación, observará que puede modificar todos los campos, excepto la foto, y que las modificaciones persisten mientras la aplicación se está ejecutando, pero no se graban las modificaciones cuando la aplicación se cierra. Entonces, nuestro siguiente paso va a ser permitir al usuario modificar el campo *foto* y posteriormente veremos cómo permitirle guardar los cambios si lo desea.

Probablemente, haya registros en la base de datos que tengan el campo *foto* a *Nothing*. Lo que vamos a hacer en estos casos es colocar una imagen ficticia cuando el registro se muestre en la página, al mismo tiempo que se actualiza la posición del registro. Esa imagen la vamos a guardar como un recurso de la aplicación. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Propiedades > Recursos > Agregar archivo existente* y se mostrará el diálogo que le permitirá seleccionar una imagen ficticia. Después, en la ventana de propiedades de la imagen, asigne a su propiedad **Persistence** el valor *Incrustado en .resx* y denomine al recurso *unafoto*.

Según lo expuesto en el párrafo anterior, edite al método *ActualizarPosicionReg* y añada el código siguiente:

```
Private Sub ActualizarPosicionReg(vista As CollectionView)
  ...

  ' Poner una foto ficticia si este campo es Nothing
```

```
  If TryCast(vista.CurrentItem, telefono).foto IsNot Nothing Then
    Return
  End If
  Dim fotoFicticia As Bitmap = My.Resources.Resources.unafoto
  Dim ms As New MemoryStream()
  fotoFicticia.Save(ms, ImageFormat.Jpeg)
  TryCast(vista.CurrentItem, telefono).foto = ms.ToArray()
End Sub
```

Para modificar la foto, tendremos que implementar un código que permita obtener la foto de nuestro sistema de ficheros y asignarla al campo *foto* del registro actual. Al igual que para el resto de los campos, para cambiar la foto haremos clic sobre ese control. Esta acción hará que se genere, entre otros, el evento **MouseDown**. La respuesta a este evento será mostrar la caja de diálogo *Abrir* que permita al usuario seleccionar la imagen.

Según lo expuesto, añada el controlador para el evento **MouseDown**:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1" Margin="5"
       MinWidth="32" MinHeight="32" MaxHeight="64"
       HorizontalAlignment="Left" Source="{Binding Path=foto}"
       MouseDown="imFoto_MouseDown" />
```

Después, edite el controlador *imFoto_MouseDown* así:

```
Private Sub imFoto_MouseDown(sender As Object, e As MouseButtonEventArgs)
  Dim dlg As New OpenFileDialog()
  dlg.Filter = "ficheros jpg (*.jpg)|*.jpg|" &
               "ficheros jpeg (*.jpeg)|*.jpeg|" &
               "ficheros png (*.png)|*.png|" &
               "Todos (*.*)|*.*"
  dlg.ShowDialog()
  If dlg.FileName IsNot Nothing AndAlso dlg.FileName.Length > 0 Then
    Dim bytes As Byte() = File.ReadAllBytes(dlg.FileName)
    Dim vista As ICollectionView = ObtenerVista()
    Dim regActual As telefono = DirectCast(vista.CurrentItem, telefono)
    If regActual IsNot Nothing Then regActual.foto = bytes
  End If
End Sub
```

Obsérvese que este método primero construye el diálogo de tipo *OpenFileDialog*, después lo configura y lo muestra. Tras aceptar el usuario la imagen seleccionada, almacena en una matriz los bytes que dan lugar a la misma y se la asigna al campo *foto* del registro actual, quedando así la foto cambiada.

Sólo queda guardar los cambios en la base de datos, cuestión que veremos a continuación. Posteriormente veremos cómo añadir y borrar un registro. Para rea-

lizar todas estas operaciones, lo primero que vamos a hacer va a ser añadir tres botones, "Guardar", "Añadir" y "Borrar", según muestra la figura siguiente:



Guardar los cambios realizados

Las modificaciones realizadas sobre la colección de objetos resultante de la consulta LINQ realizada a la base de datos por medio del método *ObtenerTelefonos* de la clase *CProveedorDatosTelefono* no tendrán efecto si no se envían a la base de datos, según estudiamos en el apartado *Realizar cambios en los datos* del capítulo *LINQ*. Para ello necesitamos obtener la referencia al contexto de objetos utilizado para hacer la consulta, con el fin de invocar a su método **SaveChanges** y enviar los cambios a la base de datos. Según esto, edite el controlador del evento **Click** del botón "Guardar" como se indica a continuación:

```
Private Sub btGuardar_Click(sender As Object, e As RoutedEventArgs)
  guardarCambios("modificar", Nothing)
End Sub
```

El método anterior invoca al método *guardarCambios* que tiene dos parámetros; el primero indica la operación a realizar sobre la base de datos: "modificar", "añadir" o "borrar", y el segundo hace referencia al objeto implicado en la operación (**Nothing** para modificar).

```
Private Sub guardarCambios(op As String, objTfno As telefono)
  ' Obtener el contexto de objetos
  Dim co As bd_telefonos2Entities = TryCast( _
```

```
    odp.ObjectInstance, CProveedorDatosTelefono).ContextoDeObjs
  ' Operación: "modificar", "añadir" o "borrar"
  Select Case op
    Case "añadir"
      co.AddTotelefonos(objTfno)
      Exit Select
    Case "borrar"
      co.DeleteObject(objTfno)
      Exit Select
  End Select
  ' Enviar los cambios a la base de datos
  Try
    co.SaveChanges()
    System.Diagnostics.Debug.WriteLine("Cambios realizados")
  Catch ex As Exception
    System.Diagnostics.Debug.WriteLine(ex.InnerException.Message)
  End Try
End Sub
```

Añadir un nuevo registro

Según expusimos en el capítulo *Enlace de datos en WPF*, añadir un nuevo registro supone solicitar los datos correspondientes al usuario y añadir el objeto correspondiente a la colección actual que subyace bajo el recurso *pdTelefonos* definido en el fichero *Application.xaml*. Esta operación puede realizarse de diferentes formas; por ejemplo, añadiendo un registro con unos valores ficticios al final de la colección y modificando después los campos, o mostrando una nueva página para la captura de los datos y creando un nuevo objeto a partir de los mismos para añadirlo después a la colección.

La primera forma simplemente requeriría escribir el controlador del botón "Añadir" así:

```
Private Sub btAñadir_Click(sender As Object, e As RoutedEventArgs)
  Dim vista As ICollectionView = ObtenerVista()
  Dim objTelefono As New telefono()
  objTelefono.nombre = "Nombre"
  objTelefono.direccion = "Dirección"
  objTelefono.telefono1 = "000000000"
  TryCast(vista.SourceCollection, List(Of telefono)).Add(objTelefono)
  vista.MoveCurrentToLast()
End Sub
```

Este método crea un nuevo objeto *telefono* con unos datos ficticios y lo añade a la colección y después, moviendo la posición actual al final, lo muestra para que el usuario pueda modificarlo.

No obstante, desde un punto de vista didáctico, aunque lo anterior resulta muy sencillo, vamos a optar por implementar la segunda forma; esto es, agregar una nueva página al proyecto que devuelva el nuevo objeto *telefono* que deseamos añadir a la colección. Según esto, agregue una nueva página de tipo **PageFunction(Of** *teléfono***)** que solicite al usuario los datos de una persona. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Agregar > Nuevo elemento > Función de página (WPF)*. Denomine a la nueva página *PagFunAgregarTelefono*, por ejemplo, y haga clic en el botón *Aceptar*. Modifique el atributo **TypeArguments** para que especifique *telefono* como el tipo del valor devuelto por la página, lo cual implica añadir el espacio de nombres correspondiente a esta clase.

```
<PageFunction
    xmlns="http://.../presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    xmlns:local="clr-namespace:BDtelefonos;assembly=BDtelefonos"
    x:Class="PagFunAgregarTelefono"
    x:TypeArguments="local:telefono"
    xmlns:mc="http://.../markup-compatibility/2006"
    WindowTitle="Página agregar teléfono" Width="450" Height="350"
    Name="PagFAgregarTelefono">
    <Grid>

    </Grid>
</PageFunction>
```

En el código anterior observamos que la página **PageFunction** especifica que retornará un objeto de la clase *telefono*. Esto exige modificar también el fichero de código subyacente en el mismo sentido:

```
Class PagFunAgregarTelefono
  Inherits PageFunction(Of telefono)

  Public Sub New()
    InitializeComponent()
  End Sub
End Class
```

El siguiente paso es realizar el diseño de la página *PagFunAgregarTelefono* para que permita introducir los datos para una nueva persona; su aspecto será el mostrado en la figura siguiente. Es un diseño muy parecido al de *PagMostrar*, por lo que no abundaremos en más detalles.



Observe que la finalidad de este formulario es solicitar al usuario los valores de las propiedades *nombre*, *direccion*, *telefono1*, *observaciones* y *foto* de un nuevo objeto de la clase *telefono* que deseamos añadir a la base de datos. Este objeto, que vamos a denominar *nuevoObjTfno*, lo definiremos como un atributo de la página y lo asignaremos al contexto de datos de la misma. También, iniciaremos dicho objeto con unos valores ficticios que serán mostrados como datos iniciales una vez cargada la página, iniciación que realizaremos en el controlador del evento **Loaded** de la página. Lo expuesto implica escribir el código siguiente:

```
Class PagFunAgregarTelefono
  Inherits PageFunction(Of telefono)
  Private nuevoObjTfno As New telefono()

  Public Sub New()
    InitializeComponent()
    Me.DataContext = nuevoObjTfno
  End Sub

  Private Sub PagFAgregarTelefono_Loaded(sender As Object,
  e As RoutedEventArgs)
    nuevoObjTfno.nombre = "Nombre"
    nuevoObjTfno.direccion = "Dirección"
    nuevoObjTfno.telefono1 = "000000000"
    nuevoObjTfno.observaciones = "Observaciones"
    Dim fotoFicticia As Bitmap = My.Resources.Resources.unafoto
    Dim ms As New MemoryStream()
    fotoFicticia.Save(ms, ImageFormat.Jpeg)
    nuevoObjTfno.foto = ms.ToArray()
  End Sub
```

```
    ' ...
End Class
```

El siguiente paso es enlazar, en el fichero de código XAML, las propiedades del objeto *nuevoObjTfno* de la clase *telefono* con los controles correspondientes de la página *PagFunAgregarTelefono* según se indica a continuación:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1" Margin="5"
       MinWidth="32" MinHeight="32" MaxHeight="64"
       HorizontalAlignment="Left"
       Source="{Binding Path=foto}"
       MouseDown="imFoto_MouseDown" />
<TextBox Grid.Column="1" Grid.Row="2" Grid.ColumnSpan="2"
         Name="ctNombre" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=nombre}"/>
<TextBox Grid.Column="1" Grid.Row="3" Grid.ColumnSpan="2"
         Name="ctDireccion" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=direccion}"/>
<TextBox Grid.Column="1" Grid.Row="4" Name="ctTelefono"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=telefono1}"/>
<TextBox Grid.Column="1" Grid.Row="5" Grid.ColumnSpan="2"
         Name="ctObservaciones"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=observaciones}"/>
```

Los datos solicitados al usuario a través del formulario son introducidos por el mismo en las respectivas cajas de texto, excepto la foto que, para cambiarla, habrá que hacer clic sobre ella y buscarla en el sistema de ficheros de la máquina donde se está ejecutando la aplicación. Esta última operación la realizaremos desde el controlador del evento **MouseDown** de control **Image** según puede verse en el código indicado a continuación:

```
Private Sub imFoto_MouseDown(sender As Object, e As MouseButtonEventArgs)
  Dim dlg As New OpenFileDialog()
  dlg.Filter = "ficheros jpg (*.jpg)|*.jpg|" &
               "ficheros jpeg (*.jpeg)|*.jpeg|" &
               "ficheros png (*.png)|*.png|" &
               "Todos (*.*)|*.*"
  dlg.ShowDialog()

  If dlg.FileName IsNot Nothing AndAlso dlg.FileName.Length > 0 Then
    nuevoObjTfno.foto = File.ReadAllBytes(dlg.FileName)
  End If
End Sub
```

Para devolver un valor, la función de página tiene que construir un objeto **ReturnEventArgs(Of** *teléfono***)** pasando como argumento el valor a devolver, en

nuestro caso, el objeto *nuevoObjTfno* que hemos creado con los datos introducidos, y llamar a su método **OnReturn** para volver a la página que la llamó pasando como argumento el objeto **ReturnEventArgs(Of** *teléfono***)** construido; el objeto *nuevoObjTfno* está referenciado por la propiedad **Result** de este objeto. Según lo expuesto, los controladores de los botones "Aceptar" y "Cancelar" serán así:

```
Private Sub btAceptar_Click(sender As Object, e As RoutedEventArgs)
  ' Aceptar la información introducida
  OnReturn(New ReturnEventArgs(Of telefono)(nuevoObjTfno))
End Sub

Private Sub btCancelar_Click(sender As Object, e As RoutedEventArgs)
  ' Operación cancelada
  OnReturn(Nothing)
End Sub
```

Después, la página de llamada, *PagMostrar*, recuperará el valor devuelto por la función de página de la propiedad **Result** del objeto **ReturnEventArgs(Of** *teléfono***)** referenciado por el segundo parámetro del controlador del evento **Return** que genera el método **OnReturn**.

Para navegar desde la página *PagMostrar* a la página *PagFunAgregarTelefono*, edite el controlador del botón "Añadir" de la misma así:

```
Private Sub btAñadir_Click(sender As Object, e As RoutedEventArgs)
  Dim pagFunAgregarTelefono As New PagFunAgregarTelefono()
  AddHandler pagFunAgregarTelefono.Return,
    AddressOf pagFunAgregarTelefono_Return
  Me.NavigationService.Navigate(pagFunAgregarTelefono)
End Sub
```

Este método se ejecutará cuando el usuario haga clic en el botón "Añadir" de *PagListado*. A partir de este instante los hechos se suceden así: se navega a la página *PagFunAgregarPersona*, el usuario introduce en esta página los datos correspondientes a una persona y hace clic en el botón "Aceptar" o "Cancelar", instante en el que se ejecuta el método **OnReturn**, que, entre otras cosas, genera el evento **Return**, y se vuelve a la página *PagMostrar*.

Entonces, para recuperar la información introducida, la página de llamada, *PagMostrar*, tiene que controlar ese evento **Return**, de ahí que hayamos añadido en el controlador del evento **Click** del botón "Añadir" este código:

```
AddHandler pagFunAgregarTelefono.Return,
  AddressOf pagFunAgregarTelefono_Return
```

Según lo expuesto, lo siguiente que tenemos que hacer es añadir a *PagListado* el controlador *pagFunAgregarTelefono_Return* del evento **Return**. La función de este controlador es agregar el objeto de la clase *telefono*, *e.Result*, a la colección de objetos proporcionada por el proveedor de datos, que subyace bajo la vista proporcionada por el método *ObtenerVista*, y a la base de datos:

```
Private Sub pagFunAgregarTelefono_Return(sender As Object,
e As ReturnEventArgs(Of telefono))
  If e IsNot Nothing Then
    Dim vista As ICollectionView = ObtenerVista()
    ' Añadir el objeto a la colección
    TryCast(vista.SourceCollection, List(Of telefono)).Add(e.Result)
    vista.MoveCurrentToLast()
    ' Añadir el objeto a la base de datos
    guardarCambios("añadir", e.Result)
  End If
End Sub
```

Si ahora ejecuta la aplicación, observará que se produce un error. Este error se produce justo cuando el método anterior invoca al método *ObtenerVista*. ¿Por qué? Porque el atributo *odp*, de tipo **ObjectDataProvider**, es **Nothing**. ¿Qué ha sucedido para que este atributo haya pasado a valer **Nothing**? En el apartado *Duración y diario de las páginas* de este mismo capítulo dijimos que cada objeto **Page** visitado dura desde que se navega al mismo (se construye y se carga) hasta que desde él se navega a otra parte (se descarga), por lo que el estado del contenido no se recuerda de una navegación a la siguiente; esto es lo que ha ocurrido con *PagMostrar*. Cuando se navega a la página *PagFunAgregarTelefono* y se retorna a *PagMostrar*, lo primero que sucede es que se ejecuta el constructor de *PagMostrar*, con lo que, entre otras cosas, el atributo *odp* se define iniciado a **Nothing**. No obstante, e independientemente de otras soluciones, cuando sea necesario almacenar y recuperar el estado de la página mediante el historial de navegación, se puede establecer **KeepAlive** en **True** para esta página o bien usar una propiedad de dependencia del diario. Esta última opción, si quisiéramos aplicarla a *odp*, no podríamos hacerlo porque el tipo **ObjectDataProvider** no está marcado como seriable, pero sí podemos aplicar la primera opción:

```
<Page x:Class="PagMostrar"
      ...
      Loaded="Page_Loaded"
      KeepAlive="True">
```

Pruebe ahora a ejecutar la aplicación y observará que todo funciona correctamente.

Borrar un registro

Según expusimos en el capítulo *Enlace de datos en WPF*, para borrar el elemento actualmente mostrado simplemente tendremos que eliminarlo de la colección subyacente de la vista. Para ello, edite el controlador del evento **Click** del botón "Borrar" como se indica a continuación:

```
Private Sub btBorrar_Click(sender As Object, e As RoutedEventArgs)
  ' Quitar de la colección el elemento mostrado
  Dim vista As ICollectionView = ObtenerVista()
  If vista.CurrentPosition > -1 Then
    Dim tfno As telefono = DirectCast(vista.CurrentItem, telefono)
    If tfno IsNot Nothing Then
      Dim siguientePosActual As Integer = (vista.CurrentPosition - 1)
      TryCast(vista.SourceCollection,
        List(Of telefono)).Remove(tfno)
      vista.Refresh()
      If siguientePosActual > -1 Then
        vista.MoveCurrentToPosition(siguientePosActual)
      End If
      ' Borrar el objeto de la base de datos
      guardarCambios("borrar", tfno)
    End If
  End If
End Sub
```

Este método quita de la colección y de la base de datos el elemento actualmente mostrado y muestra el elemento anterior al eliminado. La propiedad **CurrentPosition** valdrá -1 cuando no haya elementos en la vista para mostrar y el método **Refresh** vuelve a crear la vista.

EL CONTROL WEBBROWSER

Según lo que hemos visto hasta ahora, utilizando páginas podemos crear aplicaciones WPF con estilo de navegación web, utilizando XBAP podemos ejecutar la aplicación WPF en el explorador, igual que una página web, y utilizando un control **Frame** podemos poner una página web HTML en una ventana WPF. En este último caso, perdemos todo el control sobre el contenido; esto es, no se puede inspeccionar cuándo el usuario se desplaza a una nueva página, no hay forma de llamar a métodos JavaScript en una página web HTML o no hay forma de permitir a esos métodos que llamen al código WPF. Pues bien, aquí es donde entra en escena el control **WebBrowser**.

Esto es, un **Frame** es una buena opción cuando se necesita un contenedor que pueda cambiar sin problemas entre WPF y el contenido HTML y el **WebBrowser**

es una mejor opción cuando se necesita examinar el modelo de objetos de una página, controlar la navegación de páginas o permitir la interacción entre el código JavaScript y WPF.

RESUMEN

En este capítulo hemos estudiado el modelo de navegación de WPF. Se trata de un modelo basado en páginas y, por lo tanto, muy diferente al modelo tradicional basado en ventanas.

WPF admite la navegación de tipo web con páginas (clase **Page**) e hipervínculos (clase **Hyperlink**) y se puede implementar mediante páginas hospedadas en una ventana de navegación, **NavigationWindow**, hospedadas por un marco (**Frame**) hospedado en una página, o bien hospedadas en un explorador web.

Los dos navegadores de WPF son **NavigationWindow** y **Frame**, entendiendo por navegador una clase que admite la navegación y el historial de navegación. Cada uno de estos objetos actúa como anfitrión (*host*) del contenido al que se navega. Estudiamos la manera de utilizar estos navegadores y realizamos un ejemplo de acceso a datos que facilitara la comprensión de este tipo de aplicaciones.

También estudiamos el paso de datos entre páginas y cómo retornar información desde una página. Hablamos de la duración de las páginas y del historial de navegación y vimos que de forma predeterminada WPF, para optimizar memoria, no almacena objetos **Page** en el historial de navegación, sino que se crea un nuevo objeto cada vez que se navega a esa página partiendo de los metadatos relacionados con la entrada de página almacenada en el historial de navegación.

Después, como una alternativa al desarrollo de aplicaciones cliente enriquecidas, estudiamos el modelo XBAP que nos permite crear aplicaciones WPF con estilo web que, al igual que las aplicaciones webs, se pueden instalar en un servidor web e iniciarse con Internet Explorer o Firefox.

Finalmente construimos una aplicación XBAP con acceso a una base de datos SQL Server. La capa de acceso a datos se desarrolló basándose en el modelo de entidades de ADO.NET proporcionado por Entity Framewok. La aplicación permitía navegación, además de las operaciones típicas sobre una base de datos: selección, modificación, inserción y borrado de registros.

CAPÍTULO 9

© F.J.Ceballos/RA-MA

SILVERLIGHT

A pesar de que la web es el medio más popular para aplicaciones de negocios, hay algunas cosas que las aplicaciones webs ASP.NET no pueden hacer tan bien como las aplicaciones de escritorio, incluso utilizando JavaScript y Ajax, tales como usar animación, sonido, reproducción de vídeo y gráficos en 3D, aunque quizás todo esto no sea tan importante como la seguridad o la escalabilidad (las aplicaciones webs desarrolladas con ASP.NET fueron vistas en mi otro libro *Enciclopedia de Microsoft Visual Basic*). Estas limitaciones son las que viene a solucionar la tecnología *Silverlight* de Microsoft.

La tecnología Silverlight es un competidor directo de Flash. Al igual que Flash, Silverlight permite crear contenido interactivo que se ejecuta en el cliente, con soporte para gráficos dinámicos, contenido multimedia y animación, que va mucho más allá del HTML ordinario y también, al igual que Flash, Silverlight se implementa en los exploradores (también llamados navegadores) mediante un complemento ligero (un *plug-in* ligero) desarrollado para diferentes exploradores y sistemas operativos.

Silverlight cuenta con una ventaja importante y es que es una versión reducida de .NET Framework y por lo tanto permite a los desarrolladores escribir código de cliente utilizando Visual Basic o C#. Además, Silverlight incluye un subconjunto de la tecnología WPF que hemos estudiado en los capítulos anteriores, que extiende en gran medida los elementos para crear la interfaz de usuario en el explorador.

Microsoft está apostando firmemente en posicionar Silverlight como la tecnología de referencia para aplicaciones que se ejecuten tanto en la web como en dispositivos móviles (Windows Phone 7).

Resumimos a continuación las características de Silverlight:

- Es una tecnología multiplataforma que se ejecuta en varios exploradores.
- Para ejecutar una aplicación Silverlight, el explorador debe descargar e instalar un pequeño complemento que se actualiza automáticamente.
- Transmite los contenidos de audio y vídeo por secuencias y ajusta el contenido de vídeo en función del tipo de dispositivo: desde ordenadores de sobremesa hasta dispositivos móviles.
- Incluye gráficos que los usuarios pueden arrastrar, girar y acercar o alejar directamente en el explorador.
- La aplicación se puede ejecutar en un explorador o se puede configurar para que se ejecute en su equipo (ejecución fuera del explorador).
- Combina varias tecnologías en una sola plataforma de desarrollo: WPF y XML, JavaScript, integración con ASP.NET AJAX, lenguajes .NET como Visual Basic, puede conectarse a servicios WCF o SOAP (servicio ASMX) y recibir datos XML, JSON o RSS e incluye LINQ.
- Para crear rápidamente aplicaciones Silverlight se puede utilizar Visual Studio y Expression Blend.
- Permite la construcción de aplicaciones RIA (*Rich Internet Applications*) que conjugan lo mejor del mundo web con las ventajas de las aplicaciones WPF de escritorio.

ARQUITECTURA

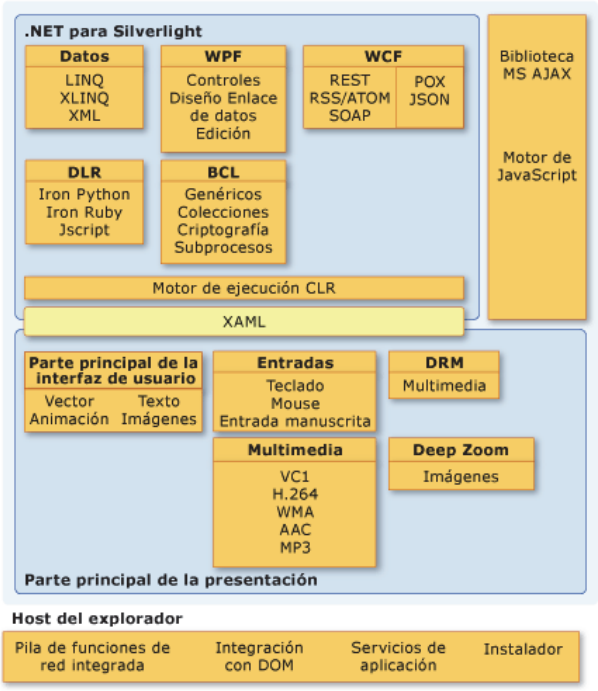
La plataforma de Silverlight se compone de .NET Framework para Silverlight, de un marco de trabajo de presentación básico y de un componente para su instalación y actualización.

.NET Framework para Silverlight es un subconjunto de .NET Framework que contiene componentes y bibliotecas, que incluyen integración de datos (por ejemplo LINQ), controles WPF extensibles que se pueden personalizar aplicando estilos y plantillas, funciones de red, biblioteca de clases comunes (BCL), soporte para lenguajes dinámicos (DLR), el CLR y soporte para servicios WCF. Algunas bibliotecas de Silverlight son ensamblados que no se incluyen en el motor de ejecución de Silverlight, sino que se distribuyen en el SDK de Silverlight para incluirlas directamente en la aplicación; esto es, se empaquetan con esta última y se descargan al explorador.

El marco de trabajo de presentación incluye los componentes y servicios orientados a la interfaz de usuario y a la interacción con el mismo. Esto es, gráficos vectoriales, texto, animaciones e imágenes, acercar imágenes y realizar un movimiento panorámico alrededor de las mismas, gestión de la entrada de datos realizada por el usuario, reproducción de elementos multimedia, capturas de cámara web y de micrófono, controles de interfaz de usuario ligeros y enlaces de datos. También incluye el lenguaje XAML para realizar el diseño.

El componente de instalación y actualización simplifica el proceso de instalar la aplicación y proporciona actualizaciones automáticas.

La figura siguiente muestra estos componentes de la arquitectura de Silverlight:



Desde el punto de vista del desarrollador, como veremos un poco más adelante, Silverlight es un control de usuario que actúa como contenedor de la aplicación. Por suerte, una buena parte de las capacidades de XAML para la definición de interfaces de usuario está presente en Silverlight. Por ejemplo, disponemos de un conjunto enriquecido de controles como Button, Calendar, CheckBox, DataGrid, DatePicker, HyperlinkButton, ListBox, RadioButton, ScrollViewer, RichTextBox, Viewbox, WebBrowser y WebBrowserBrush.

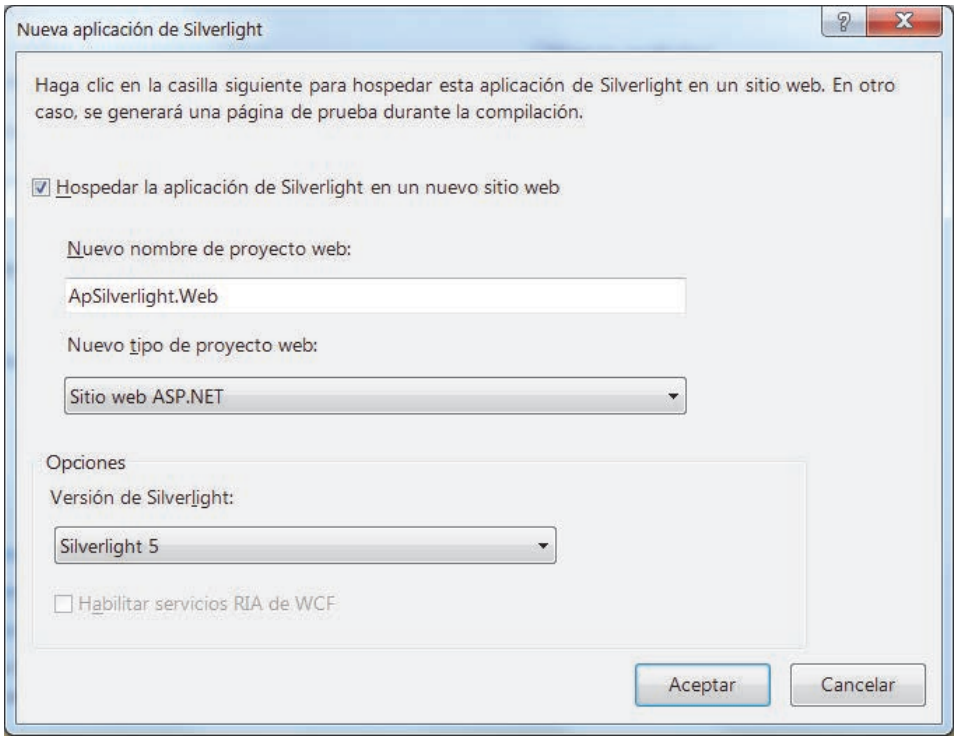
También proporciona otras características como almacenamiento aislado, que permite el acceso seguro del cliente de Silverlight al sistema de ficheros del equipo local para un usuario determinado, programación asíncrona, que lleva a cabo tareas de programación en segundo plano mientras la aplicación queda libre para interactuar con el usuario, cuadro de diálogo *Abrir* seguro, interacción entre HTML y código administrado, seriación, herramientas de compilación para construir el paquete *.xap* que contendrá la aplicación y el punto de entrada, bibliotecas XML para simplificar el trabajo con los datos XML de los servicios webs, impresión de la interfaz de usuario de una aplicación y compatibilidad de ejecución fuera del explorador; precisamente, el control **WebBrowser** es el que proporciona la superficie para mostrar contenido HTML cuando la aplicación se ejecuta fuera del explorador.

CREAR UNA APLICACIÓN SILVERLIGHT

En este apartado vamos a crear una aplicación sencilla para mostrar cómo empezar a programar en Silverlight utilizando Visual Studio. Para ello, necesitamos tener instalado Visual Studio o Visual Web Developer y Silverlight Tools para Visual Studio (este software puede descargarse desde el sitio *http://www.silverlight.net/downloads*).

El primer paso es crear un proyecto de Silverlight. Para ello, abra Visual Studio o Visual Web Developer y seleccione *Archivo > Nuevo > Proyecto > Visual Basic > Silverlight > Aplicación de Silverlight*. Denomine al proyecto, por ejemplo, *ApSilverlight*. También, es una buena opción seleccionar la casilla de verificación *Crear directorio para la solución* para agrupar en una misma carpeta los dos proyectos que se crearán. Después, haga clic en *Aceptar*. Se visualizará un diálogo, que se muestra un poco más adelante, que realiza otra serie de preguntas.

Debido a que Silverlight es una tecnología para clientes que se ejecutan en un navegador web, el diálogo mostrado plantea una pregunta inicial relativa a si se quiere *Hospedar la aplicación de Silverlight en un nuevo sitio web*. La respuesta permite a Visual Studio saber si tiene que crear una aplicación Silverlight más un proyecto ASP.NET que hospede esa aplicación o no tiene que crear este proyecto, que es justo lo que ocurre cuando se desactiva esta opción; en este caso, se crea un proyecto Silverlight independiente y se genera una página de prueba HTML para hospedar la aplicación. Este proyecto independiente se podría asociar más tarde con una aplicación ASP.NET existente.



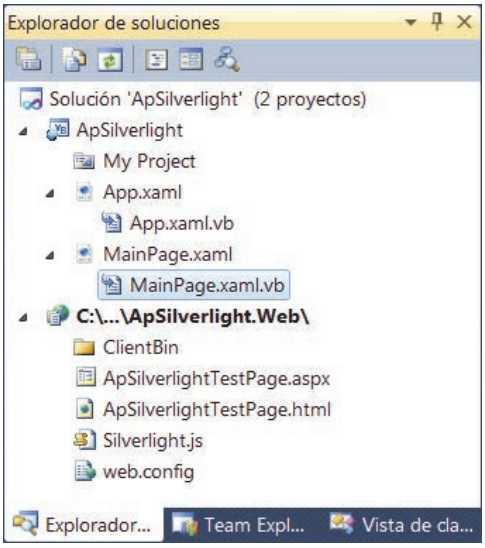
Si se eligió *Hospedar la aplicación de Silverlight en un nuevo sitio web*, puede seleccionar un nuevo tipo para el proyecto web: *Proyecto de aplicación web ASP.NET*, *sitio web ASP.NET* o *proyecto web MVC ASP.NET*. Para el primer tipo se genera un ensamblado (una *dll*) y para el segundo caso no. El tercer tipo utiliza el modelo MVC. En nuestro caso, hemos elegido como tipo de proyecto web *Sitio web ASP.NET*.

En la lista desplegable *Versión de Silverlight*, seleccione la versión de Silverlight que se desea utilizar.

También puede habilitar los servicios para proporcionar funcionalidad a las páginas Silverlight en el lado del servidor, opción *Habilitar servicios RIA de WCF*.

Arquitectura de la aplicación Silverlight

Abra el explorador de soluciones y observe que se han creado dos proyectos: *ApSilverlight* y *ApSilverlight.Web*. El primero da lugar a la aplicación Silverlight y el segundo es el medio utilizado para lanzar la ejecución de dicha aplicación.



El proyecto *ApSilverlight* está definido básicamente por los ficheros *.xaml* y *.vb* que definen el objeto aplicación y la página que se mostrará cuando se ejecute la aplicación Silverlight. Veamos el contenido y la función de estos ficheros.

El fichero *App.xaml* incluye el código XAML que define el objeto **Application** que permite configurar la aplicación Silverlight especificando los recursos aplicables a todas las páginas de la misma, según muestra el código siguiente:

```
<Application xmlns="http://.../presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="ApSilverlight.App">
    <Application.Resources>

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

Y el fichero de código subyacente *App.xaml.vb* permite escribir el código que responda a eventos tales como **Startup**, que se genera cuando se inicia la aplicación, **Exit**, que se genera justo antes de que se cierre la aplicación (no se puede cancelar), o **UnhandledException**, que se genera cuando Silverlight inicia una excepción no controlada.

```
Partial Public Class App
  Inherits Application

  Public Sub New()
    InitializeComponent()
  End Sub
```

```
  Private Sub Application_Startup(ByVal o As Object,
  ByVal e As StartupEventArgs) Handles Me.Startup
    Me.RootVisual = New MainPage()
  End Sub

  ' ...
End Class
```

Obsérvese que la interfaz de usuario de la página principal se fija a través de la propiedad **RootVisual** del objeto *App*.

El fichero *MainPage.xaml* define una página, la que será mostrada inicialmente. Esta página incluirá la colección de controles, imágenes y texto que formarán la interfaz de usuario. Evidentemente, la aplicación puede contener tantas páginas de este tipo como necesitemos. Para añadirlas sólo tiene que hacer clic con el botón secundario del ratón sobre el nombre del proyecto y seleccionar *Agregar > Nuevo elemento > Control de usuario Silverlight*.

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

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

    </Grid>
</UserControl>
```

Y *MainPage.xaml.vb* es el fichero de código subyacente en el que se puede observar que una página Silverlight es un control de usuario (una clase derivada de **UserControl**). Contendrá el código para iniciar la página, para responder a los eventos generados por las acciones del usuario y cualquier otro código necesario. Al igual que todo el código en una aplicación Silverlight, estos eventos se ejecutan en el lado del cliente.

```
Partial Public Class MainPage
  Inherits UserControl

  Public Sub New()
    InitializeComponent()
  End Sub
End Class
```

Otros ficheros que forman parte de la aplicación son:

- *AppManifest.xml*. Es el manifiesto de la aplicación requerido para generar el fichero *.xap*.

- *AssemblyInfo.vb*. Este fichero contiene los metadatos nombre y versión que están incrustados en el ensamblado generado.

- *References*. Este nodo hace referencia a las bibliotecas necesarias para una aplicación Silverlight.

Además, para lanzar la ejecución de la aplicación Silverlight se necesita de una infraestructura que puede ser un fichero HTML o una aplicación o sitio web ASP.NET. En nuestro caso, optamos por un sitio web ASP.NET *ApSilverlight.Web* que incluye los elementos siguientes:

- *ClientBin*. Carpeta que contiene el fichero XAP que empaqueta la aplicación Silverlight y que se crea cuando se construye la misma.

- *ApSilverlightTestPage.aspx*. Un fichero *.aspx* que se utiliza para configurar y crear una instancia del complemento Silverlight que descarga y ejecuta la aplicación Silverlight.

- *ApSilverlightTestPage.html*. Un fichero *.html* que se utiliza para configurar y crear una instancia del complemento Silverlight que descarga y ejecuta la aplicación Silverlight.

- *Silverlight.js*. Un fichero auxiliar que contiene funciones JavaScript para iniciar una instancia del complemento Silverlight y funciones para determinar la versión instalada en el cliente del complemento.

- *web.config*. Fichero de configuración del sitio web.

Compilación de la aplicación Silverlight

Según lo estudiado hasta ahora, cuando se crea una solución de Silverlight, Visual Studio genera también una página web ASP.NET que, evidentemente, puede contener otros elementos como formularios, páginas HTML o servicios webs. Sin embargo, hay una diferencia entre este sitio web y los estudiados en ASP.NET, y es que éste ha sido creado con la capacidad de alojar el contenido de Silverlight. Para entender cómo funciona esto, vamos a ver qué ocurre durante el proceso de compilación de la aplicación Silverlight que acabamos de construir:

1. El proyecto Silverlight es compilado en un fichero *.dll*.

2. El ensamblado resultante es empaquetado en un fichero con extensión *.xap*. Este fichero, además del ensamblado, contiene cualquier otro ensamblado dependiente que la aplicación esté utilizando (excepto aquéllos que son parte del núcleo de ejecución de Silverlight) y el manifiesto de la aplicación que hace referencia a los ficheros de los que consta la misma.

3. Finalmente, Visual Studio copia el fichero *.xap* en la carpeta *ClientBin* del sitio ASP.NET para que pueda ser accedido por las páginas del sitio web. Técnicamente este paso no es necesario, es simplemente una forma de separar el contenido de la aplicación Silverlight del resto del sitio web.

El fichero XAP es realmente un fichero ZIP (esto se puede verificar añadiendo al fichero la extensión ZIP). ¿Qué aporta este fichero? Como es un fichero comprimido, que no es descomprimido hasta que no llegue al cliente, se reduce el tiempo para descargar la aplicación y además, se simplifica la instalación; esto es, para publicar la aplicación simplemente es necesario copiar en el servidor web el fichero XAP junto con el fichero *.aspx* o *.html* que lanza la ejecución de la misma.

Sin embargo, hay algo más que debemos tener en cuenta. El servidor web debe estar configurado para permitir peticiones de ficheros de tipo XAP. Este tipo de archivo ya está incluido en IIS 7; en otro caso, habrá que añadirlo en la sección de tipos MIME del servidor.

Página de entrada

Según hemos visto en los apartados anteriores, cuando en Visual Studio se crea un proyecto Silverlight con un sitio web ASP.NET, éste proporciona dos opciones para lanzar la ejecución de la aplicación Silverlight: utilizar un fichero *.aspx* o utilizar un fichero *.html*. No obstante, en ambos casos, el mecanismo utilizado es el mismo: utilizar un elemento **<Object>** que está envuelto en un elemento ** **. La posición del elemento ** ** en la página es la posición donde se mostrará el contenido de la aplicación Silverlight. Veamos, a modo de ejemplo, cómo es este código en nuestra aplicación (fichero *ApSilverlightTestPage*):

```
<body>
  <form id="form1" runat="server" style="height: 100%">
  <div id="silverlightControlHost">
    <object data="data:application/x-silverlight-2,"
            type="application/x-silverlight-2"
            width="100%" height="100%">
      <param name="source" value="ClientBin/ApSilverlight.xap" />
      <param name="onError" value="onSilverlightError" />
      <param name="background" value="white" />
      <param name="minRuntimeVersion" value="5.0.61118.0" />
      <param name="autoUpgrade" value="true" />
```

```
      <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=5.0.61118.0"
         style="text-decoration: none">
        <img src="http://go.microsoft.com/fwlink/?LinkId=161376"
             alt="Obtenga Microsoft Silverlight"
             style="border-style: none" />
      </a>
    </object>
    <iframe id="_sl_historyFrame" style="visibility: hidden;
              height: 0px; width: 0px; border: 0px">
    </iframe>
  </div>
  </form>
</body>
```

El elemento **<Object>** incluye cinco parámetros: la localización del fichero XAP, la función que se ejecutará si ocurre un error, color del fondo de la región donde se mostrará el contenido de la aplicación, versión de Silverlight y permiso de actualización automática. A continuación hay una sección de código HTML que se mostrará si la etiqueta **<Object>** no se ejecuta satisfactoriamente o el complemento no está disponible. En la página de prueba estándar, este código se compone de una imagen "Obtenga Microsoft Silverlight" dentro de un hipervínculo que conduce a la página de descarga de Silverlight.

Cuando un usuario solicita la ejecución de esta página, se suceden los siguientes pasos:

1. El servidor crea todos los objetos del lado del servidor e inicia el ciclo de vida de la página ASP.NET.

2. Después de que haya finalizado la ejecución de todos los controladores de eventos, el servidor construye la página HTML que enviará al cliente.

3. La página HTML construida se envía al cliente y los objetos construidos en el lado del servidor se liberan.

4. Cuando el explorador recibe la página, empieza a procesarla y cuando se encuentra con el elemento **<Object>**, inicia el complemento Silverlight, descarga el archivo XAP e inicia la aplicación.

5. La aplicación Silverlight se ejecuta en el navegador del cliente. No se ejecutará más código de la página web del lado del servidor, a menos que el usuario navegue a otra página o se actualice la página actual; cualquiera de estos dos casos cerrará la aplicación Silverlight actual y reiniciará todo el proceso. Si el usuario interactúa con un control ASP.NET en cualquier otra parte de la página, ese control puede enviar de vuelta la página, lo que pondría fin a la aplicación Silverlight, o hacer una llamada al servidor web utilizando ASP.NET

AJAX, en cuyo caso no interfiere en la misma. Como se verá más adelante, la aplicación Silverlight también tiene la capacidad de activar el código del servidor web llamando a un servicio web.

DISEÑAR UNA PÁGINA SILVERLIGHT

Una vez que hemos creado el proyecto Silverlight, para diseñar una página, simplemente tenemos que abrir el fichero *MainPage.xaml* y añadir el código XAML que define los controles que van a formar parte de la misma. Esto no difiere de los diseños WPF que hemos realizado en capítulos anteriores, excepto que ahora, en algunos casos, tendremos algunas restricciones en cuanto a los componentes que podemos utilizar y a las propiedades de esos componentes.

Para organizar la interfaz de usuario, Silverlight proporciona tres paneles de diseño: **Canvas**, **StackPanel** y **Grid**. El panel predeterminado es **Grid**, que, como ya sabemos, es el panel de diseño más flexible y eficaz. En cuanto a los componentes que podemos utilizar para diseñar la interfaz, Silverlight proporciona una biblioteca completa de controles y cuadros de diálogo (puede verlos en la caja de herramientas de Visual Studio durante la fase de diseño). Algunos de estos controles poseen una representación visual y otros actúan como contenedores de otros controles o están contenidos, como imágenes o elementos multimedia.

Veamos a continuación algunos ejemplos.

Controles Silverlight

En los capítulos anteriores hemos trabajado ya con muchos controles que son, también, compatibles con Silverlight. Por lo tanto, en este apartado nos vamos a limitar a hacer un resumen, pensando en Silverlight, de los elementos más utilizados, tales como controles de texto estático, imágenes, controles de contenido, controles de elementos, controles de texto y elementos de texto, controles de rango definido, controles para gestionar fechas y controles varios como **GridSplitter** o **Border**. También estudiaremos otros más especializados como son los utilizados para la elaboración de gráficos 2-D, para aplicar *zoom* a las imágenes y para la reproducción de vídeo.

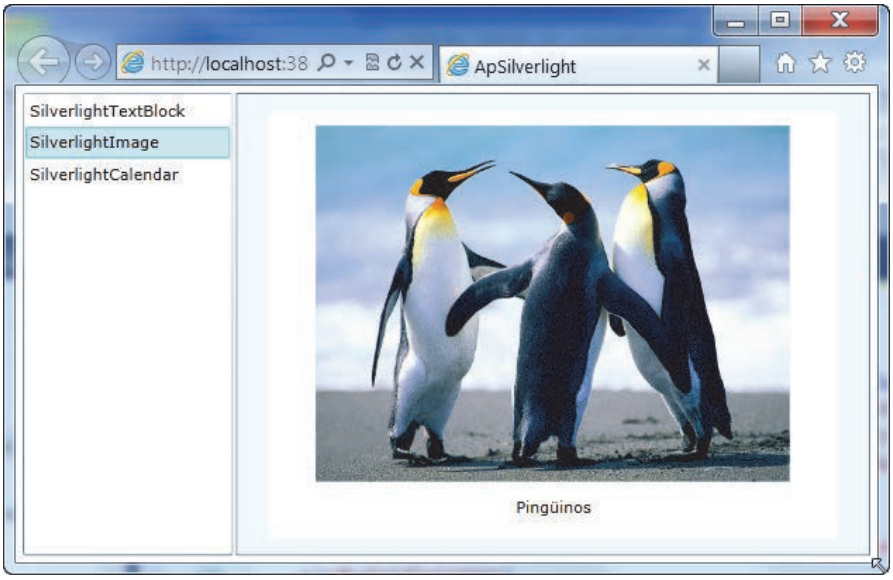
Redistribuir el espacio de los elementos de un Grid

El control **GridSplitter** redistribuye el espacio entre las filas o columnas de un **Grid**, sin cambiar las dimensiones de éste; esto es, cuando cambia el tamaño de dos columnas, incrementa la propiedad **ActualWidth** de una columna y al mismo

tiempo disminuye en la misma proporción la propiedad **ActualWidth** de la otra columna.

Para cambiar el tamaño de las filas, establezca su propiedad **HorizontalAlignment** al valor *Stretch* y **VerticalAlignment** al valor *Top*, *Bottom* o *Center* y para cambiar el tamaño de las columnas, establezca su propiedad **VerticalAlignment** al valor *Stretch* y **HorizontalAlignment** al valor *Left*, *Right* o *Center*.

Como ejemplo vamos a diseñar la página *MainPage.xaml* de la aplicación *ApSilverlight* como muestra la figura siguiente:



Esta página está formada por un **Grid** con tres columnas: la columna 0 contiene un menú formado por un **ListBox**, la columna 1 contiene un control **GridSplitter** y la columna 2, un control **Border** que hará de contenedor, por medio de su propiedad **Child**, de otras páginas utilizadas para mostrar los resultados del uso de diferentes controles que expondremos a continuación.

Border sólo puede contener un único objeto secundario, el asignado a su propiedad **Child**. Si desea colocar un borde alrededor de varios objetos, debe incluirlos en un contenedor; por ejemplo, en **StackPanel**.

Según lo expuesto, el código XAML correspondiente al diseño de esta página podría ser más o menos así:

```
<UserControl x:Class="ApSilverlight.MainPage"
  ...
```

```
  <Grid x:Name="LayoutRoot" Background="White" Margin="5">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"></ColumnDefinition>
      <ColumnDefinition Width="Auto"></ColumnDefinition>
      <ColumnDefinition Width="3*"></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <ListBox
      SelectionChanged="lstControlesSilverlight_SelectionChanged">
      <ListBoxItem Content="SilverlightTextBlock"></ListBoxItem>
      <ListBoxItem Content="SilverlightImage"></ListBoxItem>
      <ListBoxItem Content="SilverlightCalendar"></ListBoxItem>
      ...
    </ListBox>

    <sdk:GridSplitter Grid.Column="1"
                      HorizontalAlignment="Center" Width="2" />

    <Border Grid.Column="2"
            BorderBrush="SlateGray"  BorderThickness="1"
            Name="contenedorPag" Background="AliceBlue" />
  </Grid>
</UserControl>
```

Cada vez que el usuario haga clic sobre una de las opciones de la lista de la columna 0 (obsérvese que esta lista actúa como un menú de opciones), se mostrará en la columna 2 (la que está más a la derecha) la página correspondiente a la clase que tiene como nombre el texto del elemento seleccionado de la lista; por ejemplo, *SilverlightTextBlock*.

Esto quiere decir que para cada demostración que se haga a continuación sobre uno o más controles, añadiremos una nueva página a la aplicación (de tipo *Control de Usuario de Silverlight*) y el nombre de la clase, que define esa página, pasará a ser un nuevo elemento de la lista. De esta forma resultará bastante sencillo asignar un nuevo objeto de esa clase a la propiedad **Child** del **Border**. Todas las páginas serán guardadas en una carpeta *Paginas* de la aplicación.

Según lo expuesto, edite el controlador del evento **SelectionChanged** de la lista como se indica a continuación:

```
Private Sub lstControlesSilverlight_SelectionChanged(sender As Ob-
ject, e As SelectionChangedEventArgs)
  ' Acceder al elemento seleccionado
  Dim lista As ListBox = TryCast(sender, ListBox)
  Dim elementoSeleccionado As String =
    TryCast(lista.SelectedItem, ListBoxItem).Content.ToString()
  ' Crear la página correspondiente a la selección
  Dim tipo As Type = Me.GetType() ' tipo de Me
```

```
  ' Ensamblado al que pertenece "tipo"
  Dim assembly As Reflection.Assembly = tipo.Assembly
  ' Crear la página especificada
  Dim pagina As UserControl =
    DirectCast(assembly.CreateInstance(
      tipo.Namespace & "." & elementoSeleccionado), UserControl)
  ' Mostrar la página
  contenedorPag.Child = pagina
End Sub
```

El código anterior obtiene el texto correspondiente al elemento seleccionado de la lista. Ese texto es el nombre de la clase de la página que hay que crear para asignar el objeto resultante a la propiedad **Child** del **Border** y mostrar así el contenido de dicha página, para lo cual hemos utilizado la reflexión.

La reflexión es uno de los pilares de .NET. Esta característica permite obtener información acerca de los ensamblados cargados y de los tipos definidos en los mismos, como clases, interfaces y tipos valor. Gracias a la reflexión es posible implementar técnicas como la recolección de basura o la seriación. Para aplicarla, .NET proporciona las clases de los espacios de nombres **System.Reflection** y **System.Type**. También se puede utilizar la reflexión, según muestra el ejemplo anterior, para crear durante la ejecución objetos de clases específicas, así como para invocarlas y obtener acceso a las mismas.

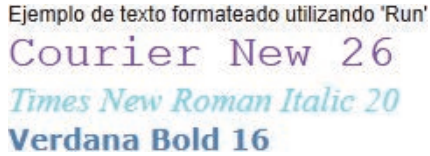
Texto estático

El control **TextBlock** es el elemento principal para mostrar texto en aplicaciones basadas en Silverlight. Es la mejor opción para mostrar texto con formato.

El ejemplo siguiente muestra cómo definir cadenas de texto con formato diferente en un objeto **TextBlock** mediante objetos **Run** separados con **LineBreak**.

```
<TextBlock FontFamily="Arial" FontSize="12"
           Text="Ejemplo de texto formateado utilizando 'Run'">
  <LineBreak/>
  <Run Foreground="BlueViolet" FontFamily="Courier New"
       FontSize="26" Text="Courier New 26"/>
  <LineBreak/>
  <Run Foreground="Aqua"  FontFamily="Times New Roman"
       FontSize="20" FontStyle="Italic"
       Text="Times New Roman Italic 20"/>
  <LineBreak/>
  <Run Foreground="SteelBlue" FontFamily="Verdana" FontSize="16"
       FontWeight="Bold" Text="Verdana Bold 16"/>
</TextBlock>
```

La ejecución del código anterior da lugar al siguiente resultado:



Silverlight soporta las fuentes Arial, Arial Black, Comic Sans MS, Courier New, Georgia, Lucida Grande/Lucida Sans Unicode, Times New Roman, Trebuchet MS y Verdana, garantizadas para ser representadas correctamente en cualquier navegador y sistema operativo compatible con Silverlight.

Imágenes

Mostrar una imagen en Silverlight es una tarea muy sencilla. Simplemente hay que añadir un elemento **Image** y establecer su propiedad **Source** especificando una dirección URL absoluta (por ejemplo, *http://sitio_web/miImagen.jpg*) o una dirección URL relativa al archivo XAP de la aplicación. Si el valor especificado no es válido, se genera el evento **ImageFailed**. Veamos un ejemplo en código XAML:

```
<StackPanel x:Name="LayoutRoot" Background="White">
  <Image Margin="10" Height="250"
    Source="/ApSilverlight;component/Imagenes/Penguins.jpg" />
  <TextBlock Text="Pingüinos" HorizontalAlignment="Center" />
</StackPanel>
```

Si no se especifican las propiedades **Height** y **Width**, la imagen se muestra a su tamaño original. Si se especifica alguna de ellas o ambas, la imagen se mostrará sin perder la proporcionalidad. Los formatos de imagen soportados por Silverlight son JPEG y PNG.

La ejecución del código anterior da lugar al siguiente resultado:



Pingüinos

La propiedad **Source** también puede establecerse a partir de un objeto **BitmapImage** construido previamente utilizando un URI:

```
Dim imagen As New Image()
imagen.Source = New BitmapImage(
  New Uri("/ApSilverlight;component/Imagenes/Penguins.jpg",
          UriKind.Relative))
LayoutRoot.Children.Add(imagen)
```

Controles de contenido

Según vimos en los capítulos anteriores, estos controles se derivan de la clase **ContentControl** y son: Label, Button, RepeatButton, ToggleButton, HyperlinkButton, ScrollViewer, Frame, CalendarButton, CheckBox, RadioButton, ListBoxItem, TabItem, ToolTip, DataGridCell o ChildWindow, entre otros. Son controles diseñados para mostrar un solo elemento de contenido a través de su propiedad **Content**. Precisamente, el límite de un elemento es lo que los diferencia de otros contenedores como los paneles de diseño, que pueden contener todos los elementos que queramos colocar en los mismos.

Veamos algunos ejemplos utilizando etiquetas y cajas de texto enlazadas a datos, ya que la funcionalidad ofrecida por **Label** en Silverlight es más amplia que la WPF, y merece la pena conocerla. También veremos otra forma de validar los datos introducidos por el usuario utilizando *atributos de anotación de datos* que especifiquen las reglas de validación. Según esto, empezaremos estudiando qué aportan estos atributos y continuaremos realizando el diseño de una página a base de etiquetas y cajas de texto que nos permita conocer la funcionalidad de la clase **Label** en Silverlight, así como la forma de validar los datos introducidos en las cajas de texto especificando atributos en las propiedades origen del enlace.

Atributos de anotación de datos

Cuando una aplicación Silverlight utiliza objetos de datos, con el fin de enlazar sus propiedades a controles de contenido, a cada clase de objeto o a sus miembros se le puede aplicar atributos que especifiquen las reglas de validación (atributos de validación), que indiquen cómo se muestran los datos (atributos de presentación) o que establezcan las relaciones entre las clases de entidad (atributos de modelado de datos). Estos atributos están definidos mediante clases derivadas de **ValidationAttribute** pertenecientes al espacio de nombres **System.ComponentModel.DataAnnotations**. El nombre de la clase es el nombre del atributo más el sufijo **Attribute**; por ejemplo, **CustomValidationAttribute**. Algunos de estos atributos son los siguientes:

- **CustomValidation**. Este atributo utiliza un método personalizado **Public** y **Shared** para la validación.

```
<CustomValidation(GetType(clase_del_método), "método")>
```

- **Range**. Define los valores mínimo y máximo para una entrada.

```
<Range(0, 999999999)>
```

- **RegularExpression**. Este atributo permite especificar una expresión regular para determinar la entrada válida. Por ejemplo, la siguiente expresión regular indica que se admite cualquiera de los rangos de caracteres, caracteres especificados y el espacio en blanco (\s); también indica que el número de caracteres tiene que estar entre 1 y 32.

```
<RegularExpression("[a-zA-Zá-úÁ-ÚñÑüÜ\s]{1,32}", ErrorMessage :=
         "No se permiten ni números ni caracteres especiales")>
```

- **Required**. Especifica que se debe proporcionar un valor.

```
<Required(ErrorMessage := "Se requiere el nombre")>
```

- **StringLength**. Designa el número de caracteres máximo y mínimo.

```
<StringLength(32, MinimumLength := 3, ErrorMessage :=
         "El nombre de tener entre 3 y 32 caracteres")>
```

- **DataType**. Especifica un tipo de datos determinado. Por ejemplo, el siguiente atributo indicaría que el valor del miembro al que se aplica el atributo tiene que ser una dirección de correo electrónico.

```
<DataType(DataType.EmailAddress)>
```

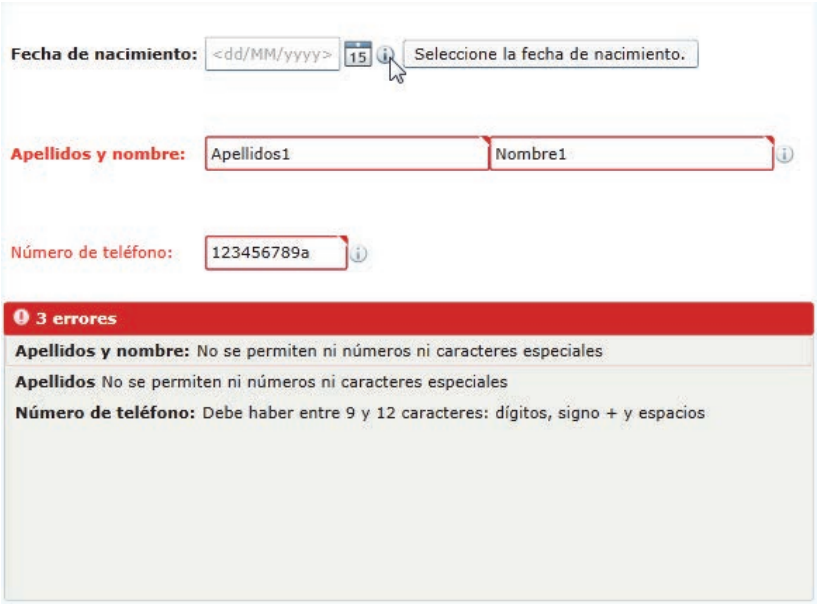
- **Display**. Se aplica a una propiedad para especificar valores que se usarán en los elementos de la interfaz de usuario para mejorar la presentación. Más adelante veremos que el valor de la propiedad **Name**, del ejemplo siguiente, lo utilizaremos como texto para un control **Label** y el de la propiedad **Descrip-**

tion, como el mensaje abreviado a mostrar por un control **DescriptionViewer**.

```
<Display(Name := "Apellidos y nombre:", Description :=
    "Introduzca primero los apellidos y después el nombre.")>
```

Para más detalles, recurra a la ayuda proporcionada por Microsoft MSDN.

Continuando con el ejemplo, la idea es mostrar una página como la siguiente:



Diseño de la interfaz

El diseño anterior puede obtenerlo a partir del código siguiente (se incluyen los enlaces a datos que comentaremos un poco más adelante):

```
<UserControl
    x:Class="ApSilverlight.SilverlightContentControl"
    ...

    xmlns:sdk="http://.../presentation/sdk"
    Width="600" Height="450">

  <Grid x:Name="LayoutRoot" Margin="15" Background="White">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="auto"/>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
```

```
<Grid.RowDefinitions>
  <RowDefinition Height="*"/>
  <RowDefinition Height="*"/>
  <RowDefinition Height="*"/>
  <RowDefinition Height="3*"/>
</Grid.RowDefinitions>

<!-- Fecha de nacimiento -->
<sdk:Label Content="Fecha de nacimiento:"
           IsRequired="True" Margin="5" />
<StackPanel Orientation="Horizontal" Grid.Column="1">
  <sdk:DatePicker Height="24" />
  <sdk:DescriptionViewer
        Description="Seleccione la fecha de nacimiento."/>
</StackPanel>

<!-- Apellidos y nombre -->
<sdk:Label Grid.Row="1" Margin="5"
           Target="{Binding ElementName=spNomAp}"
           PropertyPath="Nombre" />
<StackPanel Orientation="Horizontal"
            Grid.Column="1" Grid.Row="1">
  <StackPanel x:Name="spNomAp" Orientation="Horizontal" >
    <TextBox x:Name="tbNombre"
             Text="{Binding Path=Nombre, Mode=TwoWay,
             ValidatesOnExceptions=true,
             NotifyOnValidationError=true}"
             Height="24" Width="200" />
    <TextBox x:Name="tbApellidos"
             Text="{Binding Path=Apellidos, Mode=TwoWay,
             ValidatesOnExceptions=true,
             NotifyOnValidationError=true}"
             Height="24" Width="200" />
  </StackPanel>
  <sdk:DescriptionViewer Target="{Binding ElementName=spNomAp}"
                         PropertyPath="Nombre"/>
</StackPanel>

<!-- Número de teléfono -->
<sdk:Label Grid.Row="2" Margin="5"
           Target="{Binding ElementName=tbTelefono}" />
<StackPanel Orientation="Horizontal"
            Grid.Column="1" Grid.Row="2">
  <TextBox x:Name="tbTelefono" Height="24" Width="100"
           Text="{Binding Path=Telefono, Mode=TwoWay,
           ValidatesOnExceptions=true,
           NotifyOnValidationError=true}" />
  <sdk:DescriptionViewer
        Target="{Binding ElementName=tbTelefono}"/>
</StackPanel>
```

```
    <!-- Resumen de la validación -->
    <sdk:ValidationSummary Grid.ColumnSpan="2" Grid.Row="3"
                           DataContext="{Binding}" />
  </Grid>
</UserControl>
```

Recuerde que algunos de los controles del espacio de nombres **System.Windows.Controls** no se incluyen en el motor de ejecución de Silverlight, sino que se distribuyen en el SDK de Silverlight (por ejemplo **Label**, **DataPicker**, **DescriptionViewer** o **ValidationSummary**), por eso, para utilizarlos, hay que agregar una referencia al ensamblado System.Windows.Controls.dll e incluir el espacio de nombres XML correspondiente (**xmlns:sdk=…**) en XAML.

Contexto de datos

El origen de datos para los controles de esta página será un objeto de la clase *CPersona* que describiremos más adelante. Dicho origen lo asignaremos al contexto de datos de la página en el fichero de código subyacente de la misma, según se indica a continuación.

```
Partial Public Class SilverlightContentControl
    Inherits UserControl

  Public Sub New()
    InitializeComponent()
    Me.DataContext = New CPersona(
      "dd/MM/yy", "Apellidos", "Nombre", "123456789")
  End Sub

End Class
```

TextBox

Como ya sabemos, se trata de un control que se puede utilizar para introducir múltiples líneas de texto con un único formato.

Para habilitar múltiples líneas de texto se utiliza la propiedad **AcceptsReturn**; para habilitar las barras de desplazamiento horizontal o vertical se utilizan las propiedades **HorizontalScrollBarVisibility** o **VerticalScrollBarVisibility**; para que el control sea de sólo lectura se utiliza la propiedad **IsReadOnly**; y para modificar el borde se utilizan las propiedades **BorderBrush** y **BorderThickness**. La propiedad de contenido es **Text**.

Si necesita una caja de texto que acepte contraseñas u otros datos confidenciales, utilice el control **PasswordBox**.

DescriptionViewer

Un control **DescriptionViewer** muestra un glifo (un icono) de información y un mensaje abreviado cuando el puntero del mouse está sobre el glifo. Observe, a modo de ejemplo, este control en la figura anterior. Puede usarse para mostrar una descripción sola, estableciendo su propiedad **Description**, o una descripción asociada con otro control.

Para asociar este control con otro, se utiliza su propiedad **Target** (es una referencia a **UIElement**). Cuando esta propiedad está establecida, **DescriptionViewer** inspecciona, en el **Binding** del control referenciado, el origen del enlace (propiedad **Path**) para buscar los metadatos que pueden usarse para establecer el texto de este control. Véase más adelante **Label**, ya que ambos controles mantienen una forma de trabajo análoga.

ValidationSummary

Este control muestra una lista de errores de validación para un contenedor concreto. De forma predeterminada, mostrará tanto los errores de nivel de propiedad como de objeto, y cuando de esta lista de errores se seleccione uno, se establecerá el foco en el control en el que se originó ese error.

Label

Un control **Label** puede mostrar un texto, hacer de indicador de campo obligatorio (el texto aparece en negrita) y hacer de indicador del error de validación (el texto aparece en rojo). También se puede establecer el texto por medio de su propiedad **Content**. Normalmente se usa con un control **TextBox**. Si no necesita esta funcionalidad, puede usar el control **TextBlock**.

Para asociar un control **Label** con otro control se utiliza su propiedad **Target** (esta propiedad es una referencia a un **UIElement**). Cuando esta propiedad está establecida, **Label** inspecciona, en el **Binding** del control referenciado, el origen del enlace (propiedad **Path**) para buscar los metadatos que pueden usarse para establecer el texto del **Label**, especificar si el campo es obligatorio y especificar las reglas de validación. Si el control **Target** tiene más de un **Binding**, como ocurre en el ejemplo con el control *spNomAp*, **Label** especifica de cuál recuperar los metadatos estableciendo su propiedad **PropertyPath**.

Validación de los datos

Según vimos en el capítulo *Introducción a WPF*, los mecanismos de enlace de datos incluyen varias opciones para evaluar la validez de los datos de entrada cuando trabajamos con una vista editable.

Para el caso que nos ocupa (validación utilizando atributos de anotación notificando errores a través de una etiqueta **Label**) si la propiedad vinculada con el control referenciado por **Target** de **Label** tiene un atributo **Display** y la propiedad **Name** de éste está establecida, la propiedad **Content** de **Label** se establecerá con el valor de **Name**; si también tiene establecido un atributo **Required**, el contenido de **Label** aparecerá en negrita (este efecto puede conseguirse también de forma aislada poniendo la propiedad **IsRequired** de **Label** en **True**). Por ejemplo:

```
<Display(Name:="Apellidos y nombre:", Description:=
  "Introduzca primero los apellidos y después el nombre.")> _
... _
Public Property Nombre() As String
  ...
```

Y si la entrada del usuario no cumple las reglas establecidas, el texto de **Label** cambiará a rojo, volviendo al estado visual válido cuando se resuelva el error. Para que se dé este comportamiento ante un error, hay que establecer las propiedades **ValidatesOnExceptions** y **NotifyOnValidationError** de los controles referenciados por **Target** en **True**. El modo de enlace de los **Binding** de estos controles debe ser *TwoWay*. Por ejemplo:

```
<TextBox x:Name="tbNombre"
         Text="{Binding Path=Nombre, Mode=TwoWay,
         ValidatesOnExceptions=true,
         NotifyOnValidationError=true}"
```

Como veremos a continuación, para validar una propiedad vinculada a un control basándose en los atributos especificados para la misma, hay que probar las reglas definidas por cada atributo **ValidationAttribute** establecido. Para ello, en el código subyacente que implementa cada una de las propiedades, antes de asignar el valor *value* a la propiedad, hay que validarlo. Esto se hace invocando al método estático **ValidateProperty** de la clase **Validator** pasando como primer argumento el valor a validar y como segundo, un objeto **ValidationContext** que contiene información necesaria para la validación, como es el objeto cuya propiedad se quiere validar (**Me** en el ejemplo); obsérvese también que cuando se crea el objeto **ValidationContext** se inicia su propiedad **MemberName** con el nombre de la propiedad de **Me** cuyos atributos hay que probar.

```
' Atributos ValidationAttribute
' ...
Public Property Nombre() As String
  Get
    Return m_nombre
  End Get
  Set(ByVal value As String)
    Validator.ValidateProperty(value,
```

```
      New ValidationContext(Me, Nothing, Nothing) _
      With {.MemberName = "Nombre"})
    m_nombre = value
  End Set
End Property
```

Origen de los datos

Según lo expuesto, vamos a implementar la clase *CPersona* que dará lugar al objeto cuyas propiedades *Fecha_nacimiento*, *Nombre*, *Apellidos* y *Telefono* almacenarán los datos introducidos por el usuario, para lo cual, dichas propiedades estarán enlazadas en modo *TwoWay* con los controles correspondientes de la interfaz del usuario. Así mismo, cada una de estas propiedades definirá los atributos **ValidationAttribute** requeridos para que las etiquetas de la interfaz del usuario, que por medio de su propiedad **Target** hacen referencia a los controles vinculados con estas propiedades, puedan mostrar un texto indicando si el campo es obligatorio o hacer de indicador del error de validación.

Según lo expuesto, añada al proyecto una nueva clase *CPersona*, según se muestra a continuación, en la misma carpeta que la página *SilverlightContentControl* que estamos diseñando.

```
Imports System.ComponentModel.DataAnnotations

Public Class CPersona
  Private m_fecha_nacimiento As String
  Private m_nombre As String
  Private m_apellidos As String
  Private m_telefono As String

  Public Sub New(Optional f As String = Nothing,
  Optional n As String = Nothing, Optional a As String = Nothing,
  Optional t As String = Nothing)
    m_fecha_nacimiento = f
    m_nombre = n
    m_apellidos = a
    m_telefono = t
  End Sub

  Public Property Fecha_nacimiento() As String
    Get
      Return m_fecha_nacimiento
    End Get
    Set(value As String)
      m_fecha_nacimiento = value
    End Set
  End Property
```

```
  <Display(Name:="Apellidos y nombre:", Description:=
    "Introduzca primero los apellidos y después el nombre.")> _
  <Required(ErrorMessage:="Se requiere el nombre")> _
  <RegularExpression("[a-zA-Zá-úÁ-ÚñÑüÜ\s]{1,32}", ErrorMessage:=
    "No se permiten ni números ni caracteres especiales")> _
  Public Property Nombre() As String
    Get
      Return m_nombre
    End Get
    Set(ByVal value As String)
      Validator.ValidateProperty(value,
        New ValidationContext(Me, Nothing, Nothing) _
        With {.MemberName = "Nombre"})
      m_nombre = value
    End Set
  End Property

  <Required(ErrorMessage:="Se requieren los apellidos")> _
  <RegularExpression("[a-zA-Zá-úÁ-ÚñÑüÜ\s]{1,32}", ErrorMessage:=
    "No se permiten ni números ni caracteres especiales")> _
  Public Property Apellidos() As String
    Get
      Return m_apellidos
    End Get
    Set(value As String)
      Validator.ValidateProperty(value,
      New ValidationContext(Me, Nothing, Nothing) _
      With {.MemberName = "Apellidos"})
      m_apellidos = value
    End Set
  End Property

  <Display(Name:="Número de teléfono:", Description:=
    "Un valor compuesto por dígitos, signo + y espacios")> _
  <RegularExpression("[+0-9\s]{9,12}", ErrorMessage:=
    "Debe haber entre 9 y 12 caracteres: dígitos, signo + y espacios")> _
  Public Property Telefono() As String
    Get
      Return m_telefono
    End Get
    Set(value As String)
      Validator.ValidateProperty(value,
        New ValidationContext(Me, Nothing, Nothing) _
        With {.MemberName = "Telefono"})
      m_telefono = value
    End Set
  End Property
End Class
```

Obsérvense los atributos definidos para validar cada una de las propiedades y mostrar al usuario la información de error, si éste tuviera lugar. Obsérvese también la llamada al método **Validator.ValidateProperty** para determinar si el valor especificado para la propiedad es válido, para, si no lo es, iniciar una excepción de tipo **ValidationException**.

Controles de elementos

Estos controles, según vimos en los capítulos anteriores, fueron diseñados para contener colecciones de elementos y, generalmente, se derivan de **ItemsControl**. Si echa una ojeada al capítulo *Controles y cajas de diálogo*, podrá repasar el trabajo con los controles ListBox, ComboBox, TreeView o DataGrid, entre otros.

Otro control de interés derivado de **ItemsControl** es **TabControl**. Este control está compuesto de varios elementos **TabItem** de los cuales sólo uno está visible, el seleccionado por el usuario, lo que permite a una aplicación exponer una gran cantidad de datos en un espacio de pantalla común a todos los elementos. El elemento de pestaña que se muestra puede ser accedido a través de la propiedad **SelectedItem** o de la propiedad **SelectedIndex**.

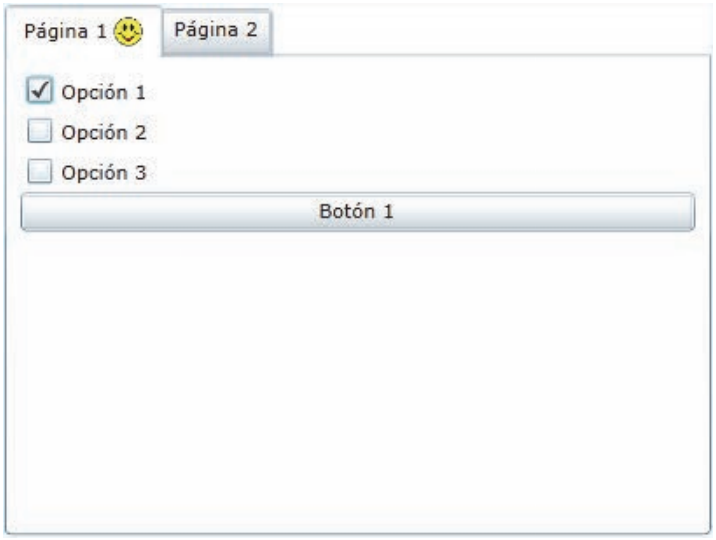
Al igual que otros controles Silverlight, **TabControl** se distribuye en el SDK de Silverlight. Para utilizar este control, hay que llenarlo con uno o más elementos **TabItem**. Cada uno de estos elementos representa una página con pestaña dentro del control. Por ejemplo, el siguiente código XAML define un **TabControl** con dos páginas, cada una de las cuales contienen otros controles:

```
<UserControl x:Class="ApSilverlight.SilverlightTabControl"
    ...
    xmlns:sdk="http://.../presentation/sdk"
    Width="400" Height="300" >

  <Grid x:Name="LayoutRoot" Background="White">
    <sdk:TabControl Name="tabControl1" >
      <sdk:TabItem Name="tabItem1">
        <sdk:TabItem.Header>
          <StackPanel Orientation="Horizontal">
            <TextBlock Margin="3" Text="Página 1"/>
            <Image
              Source="/ApSilverlight;component/Imagenes/smile.jpg"
              Height="16" />
          </StackPanel>
        </sdk:TabItem.Header>
        <StackPanel Margin="3">
          <CheckBox Name="cb1" Margin="3" Content="Opción 1" />
          <CheckBox Name="cb2" Margin="3" Content="Opción 2" />
          <CheckBox Name="cb3" Margin="3" Content="Opción 3" />
          <Button Name="bt1" Content="Botón 1" Click="bt1_Click" />
```

```
        </StackPanel>
      </sdk:TabItem>
      <sdk:TabItem Header="Página 2">
        <StackPanel Margin="3">
          ...
        </StackPanel>
      </sdk:TabItem>
    </sdk:TabControl>
  </Grid>
</UserControl>
```

Obsérvese que el primer **TabItem** contiene un título (propiedad **Header**) formado por un texto y una imagen, y un panel con tres casillas de verificación y un botón de pulsación. El contenido de la página se asigna de forma predeterminada a la propiedad **Content** del control **TabItem**.



Al igual que un **ListBox**, **TabControl** genera el evento **SelectionChanged** cuando la página visible cambia. Por ejemplo, el siguiente método se corresponde con el controlador de este evento y obtiene una referencia al propio control y otra a la página actualmente seleccionada:

```
Private Sub tabControl1_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  Dim tc As TabControl = TryCast(sender, TabControl)
  ' Página seleccionada
  Dim pagina As TabItem = TryCast(e.AddedItems(0), TabItem)
  ...
End Sub
```

La referencia *pagina* a la página actualmente seleccionada también podría haberse obtenido así:

```
Dim pagina As TabItem = TryCast(tc.SelectedItem, TabItem)
```

El acceso a los controles de cada una de las páginas o el control de los eventos de estos controles no varía de lo que ya hemos estudiado a lo largo de los capítulos anteriores. Por ejemplo, el siguiente método se corresponde con el controlador del evento **Click** del botón de la primera página y verifica si la primera casilla de verificación está seleccionada:

```
Private Sub bt1_Click(sender As Object, e As RoutedEventArgs)
  If Me.cb1.IsChecked = True Then
    System.Diagnostics.Debug.WriteLine("Opción 1 seleccionada")
  End If
End Sub
```

Controles de texto y elementos de texto

Silverlight también permite trabajar con controles de texto como TextBox, PasswordBox, AutoCompleteBox o RichTextBox, así como con los elementos derivados de Block y de Inline que estudiamos en el apartado *Documentos dinámicos* del capítulo *Controles y cajas de diálogo* (estos elementos son la base para el control RichTextBox y para el control TextBlock).

A modo de ejemplo, veamos cómo utilizar el control **AutoCompleteBox**. Este control, según muestra la figura siguiente, está compuesto por un **TextBox** y por un **ListBox** y se distribuye en el SDK de Silverlight.



El siguiente ejemplo XAML define un control de este tipo y pone estilo a la caja de texto y a los elementos de la lista:

```
<sdk:AutoCompleteBox x:Name="acbProvincias"
      Width="250"  Height="30"
      TextBoxStyle="{StaticResource EstiloTextBox}"
      ItemContainerStyle="{StaticResource EstiloListBoxItem}" />
```

El control **AutoCompleteBox** permite al usuario seleccionar un elemento de la lista que se despliega automáticamente si hay coincidencias con la entrada que ha realizado, para completar los datos de la caja de texto en lugar de escribir la entrada completa.

La lista desplegable se llena con las posibles coincidencias cuando se cumplan las condiciones establecidas por **MinimumPopulateDelay** (retraso en milisegundos antes de que la lista se rellene con las posibles coincidencias; el valor predeterminado es 0) y **MinimumPrefixLength** (número mínimo de caracteres que debe especificarse en la caja de texto antes de que se muestren las posibles coincidencias; el valor predeterminado es 1).

Una vez que se ha añadido un control **AutoCompleteBox** a la interfaz gráfica, hay que crear una matriz o lista que contenga la colección con las sugerencias (en ningún orden en particular) que se proporcionarán al control cuando el usuario empiece a escribir en la caja de texto, y asignar esta lista a la propiedad **ItemsSource** del control. Por ejemplo:

```
Partial Public Class SilverlightAutoCompleteBox
  Inherits UserControl
  Public Sub New()
    InitializeComponent()

    acbProvincias.ItemsSource = New [String]() {
      "Álava (Araba)", "Albacete", "Alicante",
      ...
      "Almería", "Zamora", "Zaragoza"}
  End Sub
End Class
```

Esto es suficiente para disponer de la funcionalidad predeterminada que proporciona este control.

La colección de elementos a filtrar puede ser de cualquier tipo de objetos. Por ejemplo, de la clase *CPersona*:

```
Private listaPersonas As New List(Of CPersona)()
```

En este caso, la clase *CPersona* debe sobre escribir el método **ToString** para especificar la representación de cadena para un objeto *CPersona*. Por ejemplo:

```
Public Overrides Function ToString() As String
  Return Nombre & " " & Apellidos
End Function
```

Si se desea que la primera coincidencia encontrada se muestre automáticamente en la caja de texto, sólo hay que asignar a la propiedad **IsTextCompletionEnabled** el valor **True**.

El comportamiento predeterminado de **AutoCompleteBox** es aplicar sobre los elementos de la lista un filtro que proporcione todos aquellos elementos que empiecen por los caracteres que haya escrito el usuario. Esta conducta puede ser cambiada si es necesario. Para ello, simplemente hay que cambiar el valor de la propiedad **FilterMode** por otro de la enumeración definida por **AutoCompleteFilterMode**. Algunos de estos valores son: *None*, *StartsWith* (valor predeterminado), *StartsWithCaseSensitive*, *Contains*, *ContainsCaseSensitive* o *Custom*.

Para aplicar un filtrado personalizado (*Custom*), hay que establecer la propiedad **TextFilter** si **ItemsSource** es una colección de **String** o **ItemFilter** si **ItemsSource** es una colección de otro tipo de objetos más complejo. Ambas propiedades toman como valor un delegado que hace referencia al método que se tiene que ejecutar para realizar el filtrado. Por ejemplo:

```
acbProvincias.ItemFilter = AddressOf Filtro;
```

Donde *Filtro* es un método que tiene dos parámetros: el primero contiene el texto que el usuario ha introducido y el segundo, el elemento que actualmente se está filtrando. El método devolverá el valor **True** cuando se cumpla el filtro y **False** en caso contrario. Por ejemplo:

```
Public Function Filtro(texto As String, objeto As Object) As Boolean
  Return TryCast(objeto, CPersona).Telefono.StartsWith("686")
End Function
```

Controles de rango definido

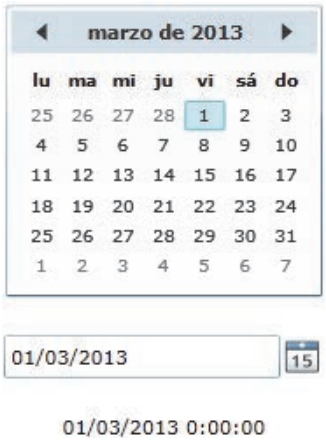
Silverlight incluye también controles que proporcionan un valor dentro de un rango previamente definido. Estos controles son ScrollBar, Slider y ProgressBar. Estos controles ya fueron expuestos en el capítulo *Controles y cajas de diálogo*.

Controles para gestionar fechas

Silverlight incluye dos controles para la gestión de fechas: Calendar y DatePicker. Ambos han sido diseñados para permitir al usuario seleccionar una fecha.

El siguiente ejemplo muestra cómo utilizar estos controles:

```
<StackPanel x:Name="LayoutRoot" Background="White">
  <sdk:Calendar x:Name="calendario" Margin="10"
     SelectedDatesChanged="calendario_SelectedDatesChanged"/>
  <sdk:DatePicker x:Name="recolectorFechas" Margin="10"
     HorizontalAlignment="Center" Width="180"
     SelectedDateChanged="recolectorFechas_SelectedDateChanged"/>
  <TextBlock Name="btxFecha" Margin="10" Text="Fecha seleccionada"
             HorizontalAlignment="Center" />
</StackPanel>
```



Para ambos controles, su propiedad **SelectedDate** proporciona la fecha seleccionada. Por ejemplo:

```
Private Sub calendario_SelectedDatesChanged(sender As Object,
e As SelectionChangedEventArgs)
  btxFecha.Text = calendario.SelectedDate.ToString()
End Sub
```

Degradados

Silverlight proporciona muchas opciones para agregar características visuales interesantes a una aplicación. Puede usar dibujos, formas, rutas y geometrías complejas. Las áreas definidas por geometrías pueden rellenarse con efectos, como imágenes, degradados de color o clips de vídeo, mediante el uso de pinceles. Por ejemplo, el siguiente código colorea el fondo del **Grid** aplicando un degradado:

```
<Grid x:Name="LayoutRoot">
  <Grid.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="White"/>
      <GradientStop Color="#FF00FFEF" Offset="0.5"/>
    </LinearGradientBrush>
  </Grid.Background>
```

```
</Grid>
```

La clase **LinearGradientBrush** pinta un área con un degradado lineal (a lo largo de una línea). Observe que los puntos finales de la línea vienen definidos por las propiedades **StartPoint** y **EndPoint**. El valor predeterminado para **StartPoint** es (0,0) y para **EndPoint**, (1,1); esto es, la línea que sigue el degradado va desde la esquina superior izquierda hasta la esquina inferior derecha. Para describir la ubicación y el color de cada punto de la transición en un degradado se utilizan objetos **GradientStop**; esta clase tiene una propiedad **Offset** que indica la ubicación del punto de degradado dentro del vector de degradado. Los colores en el degradado resultante se interpolan a lo largo del trazado.

En el ejemplo anterior, la línea que sigue el degradado va verticalmente de arriba a bajo. Si quisiéramos un degradado radial, utilizaríamos la clase **RadialGradientBrush**. Por ejemplo, el siguiente código pinta un área con un degradado radial fijando en una décima (0,1) la ubicación del punto de degradado (la degradación se aplica cada 0,1):

```
<Grid x:Name="LayoutRoot">
  <Grid.Background>
    <RadialGradientBrush>
      <GradientStop Color="White" Offset="1"/>
      <GradientStop Color="#FF00FFEF" Offset="0.1"/>
    </RadialGradientBrush>
  </Grid.Background>
</Grid>
```

El resultado sería el mostrado por la figura siguiente:



Ventanas y cajas de diálogo

Silverlight incluye las cajas de diálogo **OpenFileDialog**, que permite al usuario seleccionar uno o más ficheros del sistema de ficheros, **SaveFileDialog**, que permite al usuario guardar un fichero, las ventanas **ChildWindow**, que proporcionan una ventana que se puede mostrar sobre una ventana primaria bloqueando la interacción con ésta, y **Popup**, que coloca el contenido encima del contenido de otro elemento. Los diálogos para abrir y guardar ficheros ya los vimos en el capítulo *Controles y cajas de diálogo*; por lo tanto, vamos a centrarnos ahora en las ventanas **ChildWindow** y **Popup**.

Popup

Este control tiene mucho en común con **ToolTip**. A diferencia de **ToolTip**, el control **Popup** nunca es mostrado automáticamente; para mostrarlo hay que asignar explícitamente a su propiedad **IsOpen** el valor **True** y para ocultarlo, el valor **False**. El contenido a mostrar tiene que ser asignado a su propiedad **Child**.

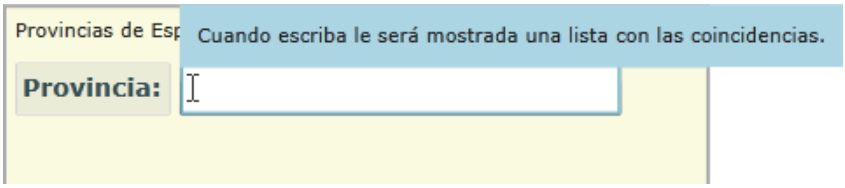
La colocación del control en el código XAML no es importante, debido a que su esquina superior izquierda siempre se alinea con la esquina superior izquierda de la región de contenido de Silverlight.

Como ejemplo, vamos a añadir un **Popup** a la página *SilverlightAutoCompleteBox* que creamos anteriormente, para que muestre un mensaje cuando el control **AutoCompleteBox** reciba el foco (por ejemplo, este mensaje: "Cuando escriba le será mostrada una lista con las coincidencias").

```
<Popup x:Name="popUpProvincias" MaxWidth="200">
  <Border Background="LightBlue"
      MouseLeftButtonDown="popUpProvincias_MouseLeftButtonDown">
    <TextBlock Margin="10">
      Cuando escriba le será mostrada una lista con
      las coincidencias.
    </TextBlock>
  </Border>
</Popup>
```

Este ejemplo utiliza un **Popup** para mostrar un borde que contiene un bloque de texto: el mensaje a mostrar. El control **Popup** se muestra cuando el control **AutoCompleteBox** recibe el foco (evento **GotFocus**):

```
Private Sub acbProvincias_GotFocus(sender As Object,
e As RoutedEventArgs)
  popUpProvincias.IsOpen = True
End Sub
```



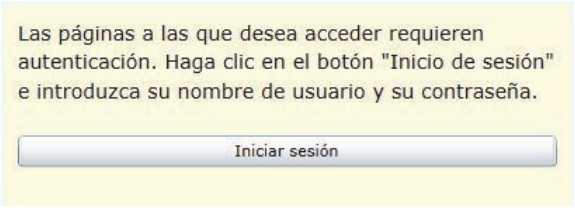
El control **Popup** se cierra al hacer clic sobre él con el botón primario del ratón:

```
Private Sub popUpProvincias_MouseLeftButtonDown(sender As Object,
e As MouseButtonEventArgs)
  popUpProvincias.IsOpen = False
End Sub
```

ChildWindow

En muchas ocasiones es necesario mostrar temporalmente una ventana para solicitar o mostrar contenido al usuario antes de continuar en la misma u otra página de la aplicación. En WPF esto lo hicimos con cajas de diálogo modales y no modales. Pues bien, análogamente, Silverlight proporciona una ventana modal que se puede mostrar sobre una ventana primaria; esta ventana será un objeto de la clase **ChildWindow** (ventana hija).

Veamos un ejemplo. Supongamos que al ejecutar una aplicación Silverlight ésta solicita al usuario que se autentique para poder continuar.



El diseño de la página anterior podría ser similar al siguiente:

```
<UserControl x:Class="ApSilverlight.SilverlightChildWindow"
    ...
    Width="400" Height="300">

  <StackPanel x:Name="LayoutRoot" Background="LightYellow">
    <TextBlock x:Name="tbInfo" Margin="10" FontSize="14"
               TextWrapping="Wrap">
      Las páginas a las que desea acceder requieren autenticación.
      Haga clic en el botón "Inicio de sesión" e introduzca
      su nombre de usuario y su contraseña.
    </TextBlock>
```

```
    <Button Name="btInicioSesion" Margin="10"
            Content="Iniciar sesión"
            Click="btInicioSesion_Click"></Button>
  </StackPanel>
</UserControl>
```

Cuando el usuario haga clic en el botón "Iniciar sesión" deseamos que se muestre una ventana de diálogo como la siguiente:



Esta ventana será un objeto de la clase *Login* derivada de **ChildWindow**. Su diseño es similar al de un control de usuario. Para facilitar esta tarea, Visual Studio proporciona la plantilla *Ventana secundaria de Silverlight* que da lugar a una ventana con dos botones en el fondo de la misma. Añada, por lo tanto, esta ventana al proyecto (análogamente a como ha hecho en los ejercicios anteriores) y complete su diseño añadiendo el resto de controles que muestra la figura anterior. El código XAML resultante podría ser similar al siguiente:

```
<controls:ChildWindow x:Class="ApSilverlight.Login"
    ...
    xmlns:controls="clr-namespace:System.Windows.Controls..."
    Title="Inicio de sesión" UseLayoutRounding="False">
  <Grid x:Name="LayoutRoot" Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock VerticalAlignment="Center" Text="Usuario:"/>
    <TextBox x:Name="txtUsuario" Grid.Column="1" Margin="0,5"/>
    <TextBlock Grid.Row="1" VerticalAlignment="Center"
               Text="Contraseña:"/>
    <PasswordBox x:Name="txtContraseña" Grid.Row="1"
                 Grid.Column="1" Margin="0,5"/>
```

```
    <Button x:Name="CancelButton" Content="Cancelar"
            Grid.Row="2" Grid.Column="1"
            Width="75" Height="23" Margin="0,12,0,0"
            HorizontalAlignment="Right"
            Click="CancelButton_Click" />
    <Button x:Name="OKButton" Content="Aceptar"
            Grid.Row="2" Grid.Column="1"
            Width="75" Height="23" Margin="0,12,79,0"
            HorizontalAlignment="Right"
            Click="OKButton_Click" />
  </Grid>
</controls:ChildWindow>
```

Los controladores de eventos para los dos botones, simplemente, según se muestra a continuación, establecen el valor de la propiedad **DialogResult** de **ChildWindow**, según estudiamos en el capítulo *Controles y cajas de diálogo*. Así, cuando el usuario haga clic sobre el botón *Aceptar* o *Cancelar* el diálogo se cerrará retornando el control a la ventana primaria. El valor **True** o **False** asignado a **DialogResult** permitirá saber qué botón se pulsó.

```
Partial Public Class Login
  Inherits ChildWindow

  Public Sub New()
    InitializeComponent()
  End Sub
  Private Sub OKButton_Click(sender As Object, e As RoutedEventArgs)
    Me.DialogResult = True
  End Sub

  Private Sub CancelButton_Click(sender As Object, e As RoutedEventArgs)
    Me.DialogResult = False
  End Sub
End Class
```

Esta ventana será mostrada desde el controlador del botón "Iniciar sesión" de la página que diseñamos anteriormente. Para ello, es necesario crear un objeto de la clase *Login* e invocar a su método **Show** (este método no devuelve nada). Según esto, añada el código siguiente a la página *SilverlightChildWindow*.

```
Partial Public Class SilverlightChildWindow
  Inherits UserControl
  Private ventanaHija As New Login()

  Public Sub New()
    InitializeComponent()
  End Sub
```

```
  Private Sub btInicioSesion_Click(sender As Object,
  e As RoutedEventArgs)
    ventanaHija.Show()
  End Sub
End Class
```

Es importante darse cuenta de que, aunque la ventana hija bloquea la interacción con la página padre, **Show** no bloquea la ejecución del código de la misma, si lo hubiera (por ejemplo, un vídeo), que seguiría ejecutándose en segundo plano. Más aún, si hubiera otro código después de la llamada a **Show**, ese código se ejecutaría inmediatamente, sin esperar a que se cierre la ventana hija. Esto presenta un problema para obtener los datos introducidos una vez cerrada la ventana hija después de que el usuario haya pulsado el botón *Aceptar*, como es nuestro caso. Entonces, para ejecutar esta tarea, tendremos que hacerlo desde el controlador del evento **Closed** de la ventana hija, según se muestra a continuación:

```
Partial Public Class SilverlightChildWindow
  Inherits UserControl

  Private ventanaHija As New Login()

  Public Sub New()
    InitializeComponent()
    AddHandler ventanaHija.Closed, AddressOf ventanaHija_Closed
  End Sub

  Private Sub btInicioSesion_Click(sender As Object,
  e As RoutedEventArgs)
    ventanaHija.Show()
  End Sub

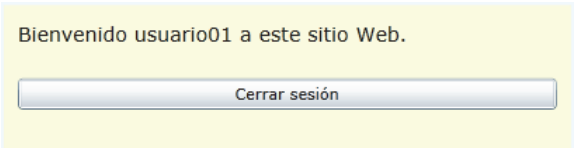
  Private Sub ventanaHija_Closed(sender As Object, e As EventArgs)
    If ventanaHija.DialogResult = True Then
      ' Aquí verificaríamos si es válida
      ' 'ventanaHija.txtContraseña.Password'
      tbInfo.Text = "Bienvenido " &
        Convert.ToString(ventanaHija.txtContraseña.Password) &
        " a este sitio Web."
      btInicioSesion.Content = "Cerrar sesión"
    End If
  End Sub
End Class
```

Obsérvese que la propiedad **DialogResult** de **ChildWindow** sólo valdrá **True** si el usuario hizo clic en el botón *Aceptar*. Si esta condición se da, se verificaría si la contraseña es correcta y se procedería según el resultado. En nuestro ejemplo, nos hemos limitado a dar la bienvenida al usuario.

Como una acción más, también hemos previsto que el botón "Iniciar sesión", después de pulsar el botón *Aceptar* de la ventana hija, cambie su título a "Cerrar sesión" y que en este estado no se pueda volver a mostrar la ventana hija.

```
Private Sub btInicioSesion_Click(sender As Object, e As RoutedEventArgs)
  If btInicioSesion.Content.ToString() = "Cerrar sesión" Then Return
    ventanaHija.Show()
End Sub
```

El resultado después de ejecutarse el controlador del evento **Closed** será éste:



Una última modificación. Extrayendo los datos introducidos directamente de las propiedades de los controles, se crean dependencias innecesarias entre la página principal y la ventana hija. Por ejemplo, piense qué ocurriría si decide cambiar el tipo de control para capturar el usuario o la contraseña: seguramente tendría que modificar el método *ventanaHija_Closed*. Para evitar estas dependencias, puede añadir a la clase *Login* las propiedades siguientes:

```
Public ReadOnly Property Usuario() As String
  Get
    Return Me.txtUsuario.Text
  End Get
End Property

Public ReadOnly Property Contraseña() As String
  Get
    Return Me.txtContraseña.Password
  End Get
End Property
```

Modifique ahora el método *ventanaHija_Closed* de la ventana principal para que haga uso de estas propiedades:

```
Private Sub ventanaHija_Closed(sender As Object, e As EventArgs)
  If ventanaHija.DialogResult = True Then
    ' Aquí verificaríamos si 'ventanaHija.Contraseña' es válida.
    tbInfo.Text = "Bienvenido " & ventanaHija.Usuario &
                  " a este sitio Web."
    btInicioSesion.Content = "Cerrar sesión"
  End If
End Sub
```

GRÁFICOS, ANIMACIÓN Y MULTIMEDIA

Silverlight ofrece la posibilidad de incorporar gráficos, animación y contenidos multimedia en una aplicación web. A continuación veremos brevemente ejemplos de aplicaciones Silverlight con gráficos, animación y contenido multimedia.

Gráficos

Este apartado resume cómo dibujar y pintar con Silverlight. En Silverlight, la forma más simple de dibujar gráficos 2-D es utilizar las formas predeterminadas que aporta su biblioteca: líneas, elipses, rectángulos, polígonos, líneas poligonales y rutas (líneas y curvas conectadas). Todas estas formas son objetos de clases (**Line**, **Ellipse**, **Rectangle**, **Polygon**, **Polyline**, **Path**) que derivan de **Shape**, del espacio de nombres **System.Windows.Shapes**, clase que a su vez se deriva de FrameworkElement, que a su vez se deriva de UIElement, lo que pone de manifiesto que las formas, al tratarse de elementos de la interfaz de usuario, se pueden utilizar dentro de una variedad de objetos contenedores como Button, Grid o Canvas. Obviamente, el contenedor **Canvas** es el más apropiado porque permite colocar las formas en coordenadas específicas, lo cual es importante cuando estamos haciendo un dibujo compuesto por múltiples formas.

Por ejemplo, el siguiente botón incluye un gráfico construido a partir de las clases derivadas de **Shape**.



El código XAML que da lugar al botón anterior es el siguiente:

```
<Button Width="100" Height="24" Margin="140,200,0,0" >
  <Button.RenderTransform>
    <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
  </Button.RenderTransform>
  <StackPanel Orientation="Horizontal">
    <Canvas Width="18" Height="18" VerticalAlignment="Center">
      <Ellipse Canvas.Left="1" Canvas.Top="1" Width="16"
               Height="16" Fill="Yellow" Stroke="Black" />
      <Ellipse Canvas.Left="4.5" Canvas.Top="5" Width="2.5"
               Height="3" Fill="Black" />
      <Ellipse Canvas.Left="11" Canvas.Top="5" Width="2.5"
               Height="3" Fill="Black" />
      <Path Data="M 5,10 A 3,3 0 0 0 13,10" Stroke="Black" />
    </Canvas>
    <TextBlock VerticalAlignment="Center" Text="Clic!" Margin="2"/>
  </StackPanel>
```

```
</Button>
```

Observamos que el contenido del botón es un **StackPanel** que contiene un **Canvas** y un **TextBlock**. El contenedor **Canvas** incluye tres elipses y una ruta para dibujar el gráfico que representa una cara con fondo amarillo, y el **TextBlock** incluye el texto que aparece a la derecha del gráfico. Todo el contenido se escala aplicando una transformación de escala (**ScaleTransform**).

Los objetos **Shape** comparten algunas propiedades como **Stroke**, que describe cómo se pinta el contorno de la forma, **StrokeThickness**, que describe el grosor del contorno de la forma, o **Fill**, que describe cómo se pinta el interior de la forma. También comparten otras propiedades de datos para especificar coordenadas y vértices medidos en píxeles independientes del dispositivo.

El hecho de que las formas sean objetos derivados de **UIElement** aporta cosas importantes. Una ya la hemos visto, que pueden actuar como contenido de otros elementos. Otra es que no es necesario manejar manualmente el proceso de repintado invalidando el área correspondiente, por ejemplo cuando una página se redimensiona, porque es algo inherente a las formas. Y otra es que las formas soportan los mismos eventos que otros elementos; esto es, no es necesario realizar ningún trabajo extra para tratar con los eventos del ratón o del teclado, por ejemplo. Para más detalles véase el capítulo de *Introducción a WPF*.

Por ejemplo, añadimos el controlador del evento **Click** del botón:

```
Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
  tb.Text = "Acaba de hacer clic en el botón"
End Sub
```

Cuando el usuario haga clic sobre este botón se mostrará en el control *tb* (un **TextBlock**) el mensaje especificado. Pero si hemos colocado sobre el botón un **StackPanel**, que contiene un **Canvas** y un **TextBlock**, ¿cómo es que todo ha funcionado correctamente independientemente de sobre qué objeto se produjo el clic? Según explicamos en el capítulo de *Introducción a WPF*, los eventos enrutados, al propagarse entre el elemento que los desencadena y la raíz del árbol de elementos, permiten asociar ese controlador a cualquier elemento en esa ruta.

Transformaciones

Una transformación define cómo asignar, o transformar, puntos de un espacio de coordenadas a otro y se describe mediante una estructura **Matrix** que no es más que una matriz de tres filas por tres columnas de valores **Double**; los valores predeterminados para esta estructura son los siguientes:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Los números importantes son los seis números de las columnas 1 y 2. La parte superior izquierda 2×2 de la matriz representa la parte lineal de la transformación (escalados y rotaciones), y las dos primeras entradas de la tercera fila representan la traslación. Por ejemplo:

$$\begin{pmatrix} 100 & 40 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ xCentro & yCentro & 1 \end{pmatrix} = \begin{pmatrix} 100 + xCentro & -40 + yCentro & 1 \end{pmatrix}$$

En este ejemplo, el resultado es que el punto (100, 40) se asigna al punto (100+*xCentro*, –40+*yCentro*): una rotación (reflejar el punto) más una traslación. Tenga en cuenta que la tercera columna de la matriz 3×3 contiene los números 0, 0, 1 y que un punto del plano es una matriz 1×3 con una tercera coordenada ficticia (la técnica más habitual es hacer que todas las terceras coordenadas sean iguales a 1). Esto siempre será así para el caso de la matriz 3×3 de una transformación afín: una transformación lineal (escalados y rotaciones) seguida de una traslación.

Vemos entonces que manipulando los valores de la matriz, puede girar, escalar y trasladar un objeto. Observe que los valores iniciales de **Matrix** no realizan ninguna transformación.

Silverlight, al igual que WPF, abstrae esta matriz proporcionando cuatro transformaciones: una para traslación, otra para escalado, otra para rotación y otra para sesgado. También tiene un tipo de transformación especial que permite definir e implementar una matriz de transformación. Veamos a continuación las clases correspondientes a esas transformaciones 2-D y algunos ejemplos que ilustren las mismas (los ejemplos completos puede verlos en la carpeta *Cap09\ApSilverlight* del CD del libro):

- **TranslateTransform**. Mueve (traslada) un elemento usando las propiedades X y Y. Por ejemplo:

```
<TranslateTransform X="-15" Y="-15" />
```

  El siguiente ejemplo mueve el hexágono cada vez que el usuario hace clic sobre el mismo:

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

```
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="200" />
  </Grid.RowDefinitions>
  <TextBlock Margin="10" HorizontalAlignment="Center">
    Clic sobre el hexágono para iniciar la animación.
  </TextBlock>
  <Polygon Grid.Row="1" x:Name="Hexagono"
           Points="100,50 200,0 300,50 300,150 200,200 100,150
                   100,50"
           Stroke="Blue" Fill="LightBlue"
           MouseLeftButtonDown="Mouse_Clicked">
    <Polygon.RenderTransform>
      <TranslateTransform x:Name="transforHexagono" />
    </Polygon.RenderTransform>
  </Polygon>
</Grid>
```

Escriba el controlador que responde al evento **Mouse_Clicked** del objeto **Polygon** en el fichero de código subyacente:

```
Private Sub Mouse_Clicked(sender As Object,
e As MouseButtonEventArgs)
  ' Traslación
  transforHexagono.X = transforHexagono.X + inc
  transforHexagono.Y = transforHexagono.Y + inc
  If inc > 0 Then
    cont += 1
    If cont = 2 Then inc = -15
  Else
    If inc < 0 Then
      cont -= 1
      If cont = -2 Then inc = +15
    End If
  End If
End Sub
```

Declare las variables *cont=2* e *inc=-15* como atributos de la clase.

- **ScaleTransform**. Escala un elemento usando las propiedades **ScaleX** y **ScaleY**. Se usa para cambiar el tamaño de un objeto según el eje horizontal, el eje vertical o ambos ejes. Por ejemplo:

```
<ScaleTransform ScaleX="1.2" ScaleY="1.2"
                CenterX="200" CenterY="100" />
```

Si aplica esta transformación al hexágono anterior, su tamaño se multiplicará por 1.2 en ambos ejes. Las propiedades **CenterX** y **CenterY** especifican el

punto central para la operación de escalado. Por defecto este punto es el (0,0), que corresponde a la esquina superior izquierda del rectángulo que envuelve la figura; esto tiene el efecto de mover el elemento. Tenga presente que cuando se aplica una transformación, se cambia el espacio de coordenadas en el que reside el objeto.

- **RotateTransform**. Gira un elemento según el valor de ángulo especificado por la propiedad **Angle** en el sentido de las agujas del reloj alrededor del punto **CenterX**, **CenterY** que por defecto es (0,0).

```
<RotateTransform Angle="30" />
```

  Las propiedades **CenterX** y **CenterY** especifican el punto central para la operación de rotación. Por defecto este punto es el (0,0), que corresponde a la esquina superior izquierda del rectángulo que envuelve la figura.

- **SkewTransform**. Sesga un elemento según los valores de las propiedades **AnleX** y **AngleY** especificados (tuerce el elemento hacia un lado poniéndolo inclinado u oblicuo). Este efecto visual resulta útil para crear la sensación de profundidad en una superficie 2-D.

```
<SkewTransform AngleX="30" AngleY="30"
               CenterX="200" CenterY="100"/>
```

  Las propiedades **CenterX** y **CenterY** especifican el punto central para la operación de transformación. Por defecto este punto es el (0,0), que corresponde a la esquina superior izquierda del rectángulo que envuelve la figura.

Las transformaciones se pueden combinar mediante el elemento **TransformGroup**. Este elemento simplemente permite especificar múltiples transformaciones aplicando el efecto combinado de cada una al objeto:

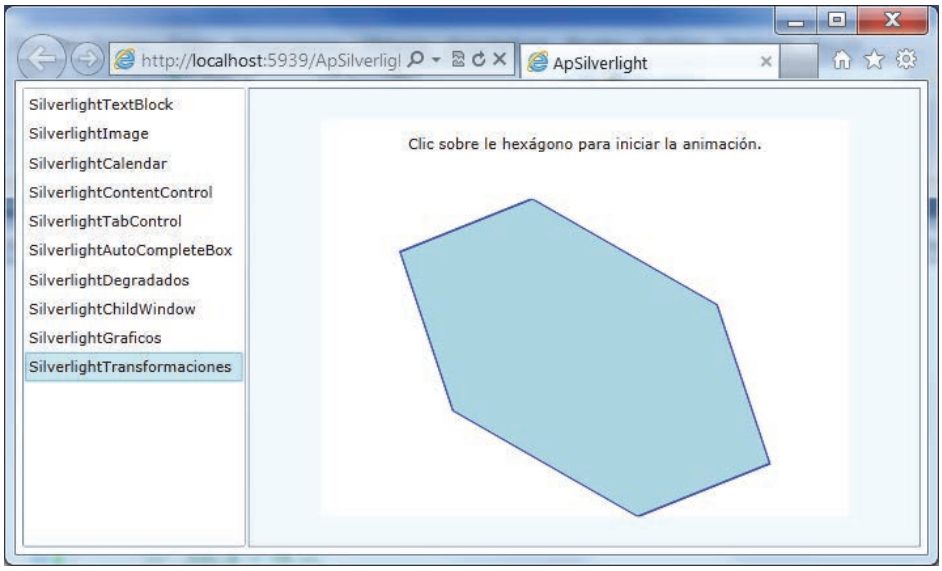
```
<TransformGroup>
  <TranslateTransform X="-15" Y="-15" />
  <ScaleTransform ScaleX="1.2" ScaleY="1.2"/>
  <RotateTransform Angle="30" />
  <SkewTransform AngleX="30" AngleY="30"/>
</TransformGroup>
```

Las transformaciones también se pueden combinar mediante una matriz de transformación, **MatrixTransform**, según explicamos anteriormente. Por ejemplo:

```
<Polygon.RenderTransform>
  <MatrixTransform Matrix="1.2 0.2 0.4 1.2 -80 -80"/>
</Polygon.RenderTransform>
```

Para aplicar una matriz de transformación desde el código subyacente proceda de forma análoga a como muestra el siguiente ejemplo:

```
Dim matriz As New Matrix(1.2, 0.2, -0.8, 1.2, 80, -80)
Dim mt As New MatrixTransform()
mt.Matrix = matriz
Hexagono.RenderTransform = mt
```



Animaciones

Este apartado resume cómo crear y trabajar con animaciones en Silverlight. Una animación es una ilusión que se crea mediante el cambio rápido entre una serie de imágenes, cada una de las cuales es ligeramente diferente de la anterior. Silverlight anima los objetos aplicando cambios a sus propiedades individuales. Por ejemplo, podemos hacer que un objeto **UIElement** se desvanezca hasta desaparecer de la vista modificando el valor de su propiedad **Opacity**.

Podemos animar diferentes tipos de propiedades; por ejemplo: **Double**, **Point** o **Color**. Por ejemplo, para animar propiedades que almacenan un valor **Double**, como **Width** u **Opacity**, se utiliza una animación que genera valores de este tipo, como **DoubleAnimation** o **DoubleAnimationUsingKeyFrames**; para animar propiedades que almacenan un valor **Color**, como, por ejemplo, el fondo o el trazo de un elemento, se utiliza una animación que genera valores de este tipo, como **ColorAnimation** o **ColorAnimationUsingKeyFrames**; para animar propiedades que almacenan un valor **Point**, como segmentos de líneas o curvas que se definen

con puntos, se utiliza una animación que genera valores de este tipo, como **PointAnimation** o **PointAnimationUsingKeyFrames**; y así sucesivamente.

Las animaciones *tipo***Animation** se denominan *animaciones From/To/By* y animan el valor de una propiedad entre dos valores de destino usando la interpolación lineal durante un tiempo especificado, y las animaciones *tipo***AnimationUsingKeyFrames** se denominan *animaciones de fotogramas clave* y animan el valor de la propiedad a lo largo de uno o más objetos que definen los fotogramas clave de la animación. Este tipo de animación es más eficaz que las animaciones *From/To/By* porque se puede especificar cualquier número de valores de destino e incluso controlar su método de interpolación. Las animaciones *From/To/By* pueden tener un máximo de dos valores de destino.

Las animaciones *From/To/By* crean una animación a partir de un valor inicial especificado por la propiedad **From** de la animación (o a partir de su valor actual si no se ha establecido) hasta un valor final especificado por la propiedad **To** o hasta un valor especificado por la suma de las propiedades **From** y **By**. Si establece el valor de las propiedades **To** y **By**, la propiedad **By** se ignora.

Como ejemplo, vamos a animar la propiedad **Fill** del hexágono que utilizamos en los ejemplos del apartado anterior, para que el color de fondo del hexágono progrese de azul claro a azul, y viceversa, en 4 segundos de forma ininterrumpida. Para ello, tenemos que realizar los siguientes pasos:

1. Crear un objeto **ColorAnimation** que anime el valor de una propiedad **Fill** entre azul claro a azul con una duración de 4 segundos.

```
<ColorAnimation
  From="LightBlue" To="Blue" Duration="0:0:4"
  AutoReverse="True"
  RepeatBehavior="Forever"
/>
```

   Observe que el valor de color inicial (**From**) de la animación es *LightBlue*, el final (**To**) *Blue*, que la animación tarda en completarse (**Duration**) 4 segundos, que la escala de tiempo retrocede después de completar una iteración de avance (**AutoReverse**) y que se repite (**RepeatBehavior**) indefinidamente.

2. Crear un guión gráfico (objeto **Storyboard**) para controlar la animación con una escala de tiempo. En nuestro ejemplo se trata de generar valores para la propiedad **Fill** entre dos valores de destino usando la interpolación lineal durante 4 segundos.

```
<Storyboard>
  <ColorAnimation
```

```
    Storyboard.TargetName="Hexagono"
    Storyboard.TargetProperty=
       "(Polygon.Fill).(SolidColorBrush.Color)"
    ...
  />
</Storyboard>
```

Observe que el objeto que se anima es *Hexagono* (**TargetName**) y que la propiedad (**TargetProperty**) que se anima de este objeto, indirectamente, es **Fill**. Esta propiedad establece el objeto **Brush** (exactamente un objeto de una clase que se deriva de **Brush** como, por ejemplo, **SolidColorBrush**) que especifica cómo se pinta el interior de la forma. Un objeto **SolidColorBrush** encapsula un color que puede ser accedido a través de la propiedad **Color**.

```
<Polygon.Fill>
  <SolidColorBrush Color="LightBlue" />
</Polygon.Fill>
```

Entonces, al animar un **Fill**, se debe utilizar la identificación indirecta. Esto es, si se va a animar el **Color** de un objeto **SolidColorBrush** que es el **Fill** del objeto **Polygon**, la sintaxis será:

```
Storyboard.TargetProperty="(Polygon.Fill).(SolidColorBrush.Color)"
```

3. Definir el objeto **Storyboard** como un recurso, por ejemplo *sbColorFondoHexagono*, para que pueda ser fácilmente referenciado desde el código con el fin de iniciar, detener, poner en pausa y reanudar el guión gráfico.

```
<Grid.Resources>
  <Storyboard x:Name="sbColorFondoHexagono">
    <ColorAnimation
        Storyboard.TargetName="Hexagono"
        Storyboard.TargetProperty=
           "(Polygon.Fill).(SolidColorBrush.Color)"
        From="LightBlue" To="Blue" Duration="0:0:4"
        AutoReverse="True"
        RepeatBehavior="Forever"
      />
  </Storyboard>
</Grid.Resources>
```

4. Asociar el guión gráfico a un evento. Hasta ahora, simplemente hemos especificado el objeto de destino de la animación y cómo se comporta la animación. El paso siguiente es especificar cuándo comienza la animación. Para ello puede utilizar un evento, por ejemplo, **MouseLeftButtonDown**:

```
<Polygon Grid.Row="1" x:Name="Hexagono"
```

```
            Points="100,50 200,0 300,50 300,150 200,200 100,150
                    100,50"
            Stroke="Blue" Fill="LightBlue"
            MouseLeftButtonDown="Mouse_Clicked">
</Polygon>
```

Escribir el controlador del evento en el fichero de código subyacente para que inicie la animación definida por el guión gráfico *sbColorFondoHexagono*:

```
Private Sub Mouse_Clicked(sender As Object,
e As MouseButtonEventArgs)
  sbColorFondoHexagono.Begin()
End Sub
```

Storyboard tiene varios métodos que permiten controlar la animación, entre los que se incluyen **Begin**, **Stop**, **Pause** y **Resume**. En este ejemplo, se utiliza el método **Begin** para iniciar la animación cuando el usuario haga clic en el rectángulo y genere el evento **MouseLeftButtonDown**.

Storyboard, así como todos los demás objetos de animación (**DoubleAnimation**, **DoubleAnimationUsingKeyFrames**, **ColorAnimation**, etc.), hereda de la clase **Timeline**. Esta clase tiene propiedades que en algunos casos le serán útiles. Por ejemplo, la propiedad **BeginTime**, como su nombre indica, permite especificar el momento en que se inicia la actividad del objeto de animación:

```
<Storyboard BeginTime="0:0:1" x:Name="sb">
  ...
  <DoubleAnimation BeginTime="0:0:2" ... />
  ...
</Storyboard>
```

En el ejemplo anterior se ha especificado un tiempo de un segundo para la propiedad **BeginTime** del objeto **Storyboard** y un tiempo de dos segundos para la misma propiedad en el objeto **DoubleAnimation**. Cuando se inicie el objeto **Storyboard** mediante **Begin**, el comienzo del guión gráfico se retrasará un segundo, pero la actividad del objeto **DoubleAnimation** se iniciará tres segundos después (un retardo de un segundo para el objeto **Storyboard** más un retardo de dos segundos para el objeto **DoubleAnimation**).

Y después de lo estudiado hasta ahora quizás le surja esta pregunta, ¿podemos animar una transformación? La respuesta es sí. ¿Cómo? Pues animando la propiedad que nos interese de la transformación. Por ejemplo, volviendo a los ejemplos anteriores, vamos a hacer rotar el hexágono animando la propiedad **Angle** de la transformación **RotateTransform**.

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

```
  <Grid.Resources>
    <Storyboard x:Name="sbRotacionHexagono">
      <DoubleAnimation
          Storyboard.TargetName="transforHexagono"
          Storyboard.TargetProperty="Angle"
          To="360" Duration="0:0:8"
          RepeatBehavior="Forever"
        />
    </Storyboard>
  </Grid.Resources>

  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="200" />
  </Grid.RowDefinitions>

  <TextBlock Margin="10" HorizontalAlignment="Center">
    Clic sobre el hexágono para iniciar la animación.
  </TextBlock>
  <Polygon Grid.Row="1" x:Name="Hexagono"
           Points="100,50 200,0 300,50 300,150 200,200 100,150
                   100,50"
           Stroke="Blue" Fill="LightBlue"
           MouseLeftButtonDown="Mouse_Clicked">
    <Polygon.RenderTransform>
      <RotateTransform x:Name="transforHexagono"
                       CenterX="200" CenterY="100" />
    </Polygon.RenderTransform>
  </Polygon>
</Grid>
```

Ahora, el objeto destino de la animación es *transforHexagono*, de tipo **RotateTransform**, y la propiedad que se anima de este objeto es **Angle**. Obsérvese que los valores de esta propiedad serán calculados entre 0 (no se especificó **From**) y 360 (**To**) grados usando la interpolación lineal durante 8 segundos.

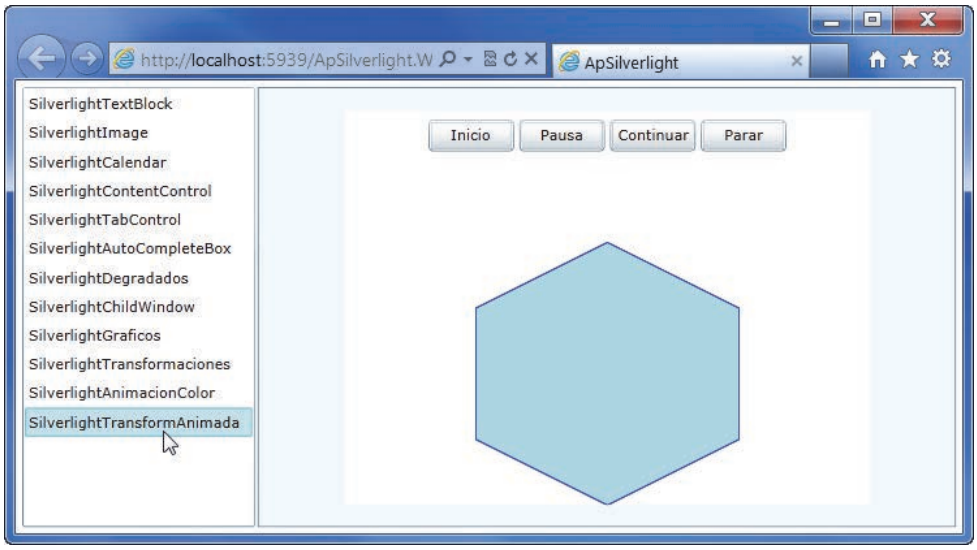
Finalmente, haciendo uso de los métodos **Begin**, **Stop**, **Pause** y **Resume** de **Storyboard** vamos a añadir a la interfaz gráfica del ejemplo anterior cuatro botones, *Inicio*, *Pausa*, *Continuar* y *Parar*, que permitan al usuario controlar la animación del hexágono. El código XAML necesario para añadir estos botones puede ser el siguiente:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.Resources>
    <Storyboard x:Name="sbRotacionHexagono">
      ...
    </Storyboard>
  </Grid.Resources>
  <Grid.RowDefinitions>
```

```
    ...
  </Grid.RowDefinitions>

  <StackPanel Orientation="Horizontal" Margin="5"
              VerticalAlignment="Top" HorizontalAlignment="Center">
    <Button Click="Inicio_Click"
      Width="65" Height="24" Margin="2" Content="Inicio" />
    <Button Click="Pausa_Click"
      Width="65" Height="24" Margin="2" Content="Pausa" />
    <Button Click="Continuar_Click"
      Width="65" Height="24" Margin="2" Content="Continuar" />
    <Button Click="Parar_Click"
      Width="65" Height="24" Margin="2" Content="Parar" />
  </StackPanel>
  <Polygon Grid.Row="1" x:Name="Hexagono"
    ...
  </Polygon>
</Grid>
```

El resultado será el que muestra la figura siguiente:



Sólo queda escribir los controladores para el evento **Click** de cada uno de los botones:

```
Private Sub Inicio_Click(sender As Object, e As RoutedEventArgs)
  If sbRotacionHexagono.GetCurrentState() = ClockState.Stopped Then
    sbRotacionHexagono.Begin()
  End If
End Sub
```

```
Private Sub Pausa_Click(sender As Object, e As RoutedEventArgs)
  sbRotacionHexagono.Pause()
End Sub

Private Sub Continuar_Click(sender As Object, e As RoutedEventArgs)
  sbRotacionHexagono.Resume()
End Sub

Private Sub Parar_Click(sender As Object, e As RoutedEventArgs)
  sbPararRotHexagono.Stop()
End Sub
```

Observe que el método **Begin** sólo se ejecutará si la animación fue parada (se invocó al método **Stop**). De esta forma se evita lanzar el guión repetidas veces, lo cual perjudica al rendimiento de la aplicación.

Para detener la animación simplemente hay que invocar al método **Stop**. Esto hace que el objeto animado vuelva a su posición original de una forma brusca, en un solo paso. Quizás sea interesante hacer una parada controlada para que sea más suave. Para ello, podemos iniciar un segundo guión que sustituya al primero. Este nuevo guión puede obviar la propiedad **From**, lo que permitirá al polígono girar sin problemas desde su posición actual a la posición original en un tiempo especificado. Por ejemplo:

```
<Grid.Resources>
  <Storyboard x:Name="sbRotacionHexagono">
    ...
  </Storyboard>
  <Storyboard x:Name="sbPararRotHexagono"
              Completed="sbPararRotHexagono_Completed">
    <DoubleAnimation
        Storyboard.TargetName="transforHexagono"
        Storyboard.TargetProperty="Angle"
        Duration="0:0:0.6"
      />
  </Storyboard>
</Grid.Resources>
```

Según este nuevo diseño, los controladores de los botones *Inicio* y *Parar* modificarán su respuesta para establecer el guión gráfico, de los dos existentes, que corresponda en cada caso. Piense que el usuario puede parar la animación y volverla a iniciar de nuevo.

```
Private Sub Inicio_Click(sender As Object, e As RoutedEventArgs)
  If sbRotacionHexagono.GetCurrentState() = ClockState.Stopped Then
    Storyboard.SetTarget(sbRotacionHexagono,
                         Hexagono.RenderTransform)
    sbRotacionHexagono.Begin()
```

```
  End If
End Sub

Private Sub Parar_Click(sender As Object, e As RoutedEventArgs)
  Storyboard.SetTarget(sbPararRotHexagono,
                       Hexagono.RenderTransform)
  sbPararRotHexagono.Begin()
End Sub
```

Observe también que, ahora, cuando se para la animación (**Stop**) a la que da lugar el primer guión, se inicia el segundo guión gráfico, lo que conducirá al objeto a su posición original de una forma suave. Hasta aquí todo perfecto, pero si ahora intentamos reanudar la animación, ésta no se inicia. ¿Por qué? Observe las sentencias **if**, ninguno de los dos guiones ha sido parado. La solución será parar (**Stop**) ambos guiones cuando el segundo guión se complete, proceso que ejecutaremos en respuesta al evento **Completed** del segundo guión:

```
Private Sub sbPararRotHexagono_Completed(sender As Object,
e As EventArgs)
  sbPararRotHexagono.Stop()
  sbRotacionHexagono.Stop()
End Sub
```

Audio y vídeo

Este apartado resume cómo se trabaja con audio y vídeo en Silverlight. Agregar contenido multimedia a una página es tan simple como agregar un objeto **MediaElement** y asignar a su propiedad **Source** el identificador uniforme de recursos (URI) correspondiente al contenido multimedia que se desea reproducir. La reproducción se iniciará cuando se cargue la página. Por ejemplo:

```
<MediaElement x:Name="meReproductor"
   Source="/ApSilverlight;component/Media/Pretenders.mp3">
</MediaElement>
```

El identificador uniforme de recursos utilizado tiene el mismo formato que utilizamos con **Image**.

El objeto **MediaElement** puede reproducir archivos de vídeo de Windows Media (WMV), audio de Windows Media (WMA), MP3 y MP4, entre otros.

La clase **MediaElement**, además de heredar las propiedades de **UIElement**, proporciona varias propiedades específicas del contenido multimedia. Veamos las más comunes:

- **AutoPlay**. Esta propiedad especifica si la reproducción de **MediaElement** se debe iniciar automáticamente. El valor predeterminado es **True**.

- **Volume**. Indica el volumen de sonido del objeto **MediaElement**. Es un valor de 0 a 1, siendo 1 el valor más alto. El valor predeterminado es 0,5.

- **IsMuted**. Permite desactivar (valor **True**) o activar (valor **False**) el sonido de **MediaElement**. El valor predeterminado es **False**.

- **Stretch**. Especifica cómo se ajusta el vídeo para rellenar el rectángulo de destino del contenido del objeto **MediaElement**. Los valores posibles son:
  - *None*. El contenido conserva su tamaño original. Recomendado para mejorar el rendimiento.
  - *Fill*. El tamaño del contenido se ajusta a las dimensiones del destino. La relación de aspecto original no se conserva.
  - *Uniform*. Es el valor predeterminado. El tamaño del contenido se ajusta a las dimensiones del destino y conserva su relación de aspecto original.
  - *UniformToFill*. Igual que *Uniform* y además, si la relación de aspecto del rectángulo de destino difiere del origen, el contenido de origen se recorta de modo que se ajuste a las dimensiones de destino.

También proporciona varios eventos, entre los que destacamos los siguientes:

- **MediaOpened**. Este evento se produce cuando la secuencia multimedia se ha validado, abierto y se han leído los encabezados del fichero a reproducir.

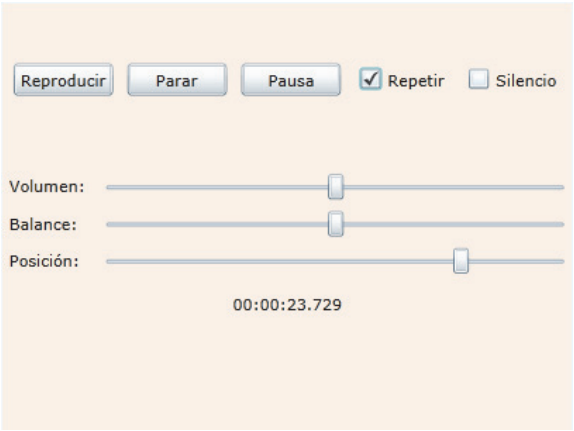
- **MediaEnded**. Este evento se produce cuando la reproducción de la secuencia multimedia ha finalizado.

- **MediaFailed**. Este evento se produce cuando hay un error asociado con el origen a reproducir, propiedad **Source** del elemento multimedia, bien porque el fichero origen de la secuencia multimedia no se encontró, porque el formato no es válido o por cualquier otra causa desconocida.

Estas propiedades y los métodos proporcionados por **MediaElement** nos permitirán tener un control específico sobre el reproductor según veremos en los ejemplos que iremos exponiendo a continuación. Un detalle: cuando utilizamos **MediaElement** para reproducir audio, el objeto permanece invisible, cosa que no ocurre, obviamente, cuando lo utilizamos para reproducir vídeo.

Por ejemplo, vamos a diseñar la interfaz que muestra la figura siguiente para realizar esas operaciones a las que nos hemos referido y otras.

Para este diseño emplearemos una rejilla con tres filas. En la primera fila añadiremos los botones, en la segunda, los controles deslizantes y en la tercera, un control de texto para mostrar el tiempo de reproducción transcurrido así como el estado del reproductor.



Añadimos en primer lugar el elemento multimedia **MediaElement**. La conducta inicial del objeto **MediaElement** está determinada por la propiedad **AutoPlay**. Por lo tanto, si deseamos tener un control específico sobre el reproductor, empezaremos por poner esta propiedad a **False**:

```
<MediaElement x:Name="meReproductor" AutoPlay="False"
   Source="/ApSilverlight;component/Media/Pretenders.mp3">
</MediaElement>
```

Después, estableceremos la propiedad **Source** con el fichero a reproducir (asegúrese de que el fichero a reproducir forma parte del proyecto y de que su propiedad *Acción de compilación* vale *Resource*). Al establecer **Source** se restablecerá el valor de la propiedad **Position** del elemento multimedia en 00:00:00. Esta propiedad, de tipo **TimeSpan**, indica el tiempo de reproducción que actualmente ha transcurrido.

Como la propiedad **AutoPlay** vale **False**, el fichero de audio (o de vídeo) será cargado, pero no será iniciada su reproducción, sino que tendremos que interactuar con el dispositivo para poner en marcha la misma, detenerla o finalizarla. Estas operaciones las podremos realizar utilizando los métodos **Play**, **Pause** y **Stop**.

Entonces, como siguiente paso, añadamos los botones *Reproducir*, *Parar*, *Pausa*, *Repetir* y *Silencio*:

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
            VerticalAlignment="Center" Margin="5">
```

```
  <Button Name="btReproducir" Content="Reproducir"
          Width="70" Margin="5" Click="btReproducir_Click"/>
  <Button Name="btParar" Content="Parar" Width="70" Margin="5"
          Click="btParar_Click"/>
  <Button Name="btPausa" Content="Pausa" Width="70" Margin="5"
          Click="btPausa_Click"/>
  <CheckBox Margin="8" Content="Repetir" x:Name="chbRepetir"/>
  <CheckBox Margin="8" Content="Silencio" x:Name="chbSilencio"
            Click="chbSilencio_Click"/>
</StackPanel>
```

Los controladores de los botones *Reproducir*, *Parar* y *Pausa* simplemente tendrán que invocar, respectivamente, a los métodos **Play**, **Stop** y **Pause**:

```
Private Sub btReproducir_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.Play()
End Sub

Private Sub btParar_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.Stop()
End Sub

Private Sub btPausa_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.Pause()
End Sub
```

El botón de opción *Repetir* tiene como finalidad repetir la reproducción una vez que ésta haya finalizado. Según lo expuesto anteriormente, cuando esto suceda se generará el evento **MediaEnded** del elemento multimedia. Entonces, será el controlador de este evento el que reinicie la reproducción si la opción *Repetir* está señalada. Por lo tanto, añada este controlador para que, si la opción *Repetir* está señalada, ponga a cero la propiedad **Position** e inicie de nuevo la reproducción:

```
Private Sub meReproductor_MediaEnded(sender As Object,
e As RoutedEventArgs)
  If chbRepetir.IsChecked = True Then
    meReproductor.Position = TimeSpan.Zero
    meReproductor.Play()
  End If
End Sub
```

El botón de *Silencio* cuando esté señalado debe desactivar el audio, lo que se consigue poniendo la propiedad **IsMuted** del reproductor a valor **True**. Por lo tanto, añada el controlador del evento **Click** de este botón y asigne a esta propiedad el valor de la propiedad **IsChecked** del botón *Silencio*:

```
Private Sub chbSilencio_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.IsMuted = CBool(chbSilencio.IsChecked)
```

```
End Sub
```

Ya tenemos el reproductor controlado, pero es bueno que el usuario tenga información del tiempo de reproducción transcurrido, del estado en el que se encuentra el reproductor o de si se produjo algún error. Esta información la mostraremos por medio de un control de texto que colocaremos en la tercera fila de la rejilla, según dijimos anteriormente:

```
<TextBlock Grid.Row="2" x:Name="tbkEstado" Text="00:00:00"
           Margin="10" HorizontalAlignment="Center"/>
```

Cuando se produce un error se genera el evento **MediaFailed** del elemento multimedia. Añada entonces el controlador de este evento para que, si se produjera un error, pueda mostrar a través de este control el mensaje correspondiente:

```
Private Sub meReproductor_MediaFailed(sender As Object,
e As ExceptionRoutedEventArgs)
  tbkEstado.Text = e.ErrorException.Message
End Sub
```

Este control mostrará también el estado (*Playing*, *Paused* o *Stopped*) del elemento multimedia proporcionado por su propiedad **CurrentState**. Cuando el valor de esta propiedad cambia, el objeto **MediaElement** genera su evento **CurrentStateChanged**. Por lo tanto, añada el controlador de este evento y complételo como se indica a continuación:

```
Private Sub meReproductor_CurrentStateChanged(sender As Object,
e As RoutedEventArgs)
  tbkEstado.Text = meReproductor.CurrentState.ToString()
End Sub
```

Queda mostrar el tiempo de reproducción transcurrido proporcionado por la propiedad **Position** del elemento multimedia. Actualizar periódicamente el control de texto *tbkEstado* con este valor requerirá añadir un temporizador (los temporizadores fueron explicados en el apartado *Temporizadores y modelo de subprocesos* en el capítulo *Controles y cajas de diálogo*). Pues bien, añadamos un temporizador de la clase **DispatcherTimer** en el código subyacente de nuestra página y añadamos también el controlador de su evento **Tick** que se generará cada **Interval** segundos:

```
Partial Public Class SilverlightAudio
  Inherits UserControl
  Private temporizador As DispatcherTimer

  Public Sub New()
    InitializeComponent()
```

```
    temporizador = New DispatcherTimer()
    temporizador.Interval = TimeSpan.FromSeconds(0.1)
    AddHandler temporizador.Tick, AddressOf Temporizador_Tick
  End Sub

  Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
  End Sub
  ...
End Class
```

A continuación, complete el controlador *Temporizador_Tick* para que asigne el valor de la propiedad **Position** del elemento multimedia al control *tbkEstado*:

```
Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
  tbkEstado.Text =
    meReproductor.Position.ToString().TrimEnd(New Char() {"0"c})
End Sub
```

El temporizador será puesto en marcha cuando se inicie la reproducción del contenido multimedia. Por lo tanto, añada al controlador del evento **Click** del botón *Reproducir* la línea de código que realiza esta operación:

```
Private Sub btReproducir_Click(sender As Object, e As RoutedEventArgs)
  temporizador.Start()
  meReproductor.Play()
End Sub
```

Así mismo, el temporizador será detenido cuando el usuario haga clic en los botones *Parar* o *Pausa*, o cuando finalice la reproducción y el botón *Repetir* no esté señalado.

```
Private Sub btParar_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.Stop()
  temporizador.Stop()
End Sub

Private Sub btPausa_Click(sender As Object, e As RoutedEventArgs)
  meReproductor.Pause()
  temporizador.Stop()
End Sub

Private Sub meReproductor_MediaEnded(sender As Object, e As
RoutedEventArgs)
  If chbRepetir.IsChecked = True Then
    meReproductor.Position = TimeSpan.Zero
    meReproductor.Play()
  Else
    temporizador.Stop()
  End If
```

```
End Sub
```

A continuación vamos a añadir los deslizadores que permitan al usuario controlar el volumen, el balance y la posición de la reproducción. Para que el diseño le resulte sencillo, puede colocar las etiquetas y los **Slider**, que muestra la figura anterior, en un **Grid** de tres filas por dos columnas.

```
<TextBlock Margin="5" Text="Volumen: "/>
<Slider Grid.Column="1" x:Name="sliderVolumen"
        Minimum="0" Maximum="1" Value="0.5"
        ValueChanged="sliderVolumen_ValueChanged" />
<TextBlock Grid.Row="1" Margin="5" Text="Balance: "/>
<Slider Grid.Row="1" Grid.Column="1" x:Name="sliderBalance"
        Minimum="-1" Maximum="1"
        ValueChanged="sliderBalance_ValueChanged" />
<TextBlock Grid.Row="2" Margin="5" Text="Posición: "/>
<Slider Grid.Column="1" Grid.Row="2"  x:Name="sliderPosicion"
        Minimum="0" />
```

El **Slider** para controlar el volumen proporciona un valor entre 0 y 1, el correspondiente a la propiedad **Volumen** del elemento multimedia. Cuando el usuario interactúe con este control para seleccionar un valor del rango establecido, se generará el evento **ValueChanged** y el valor seleccionado será proporcionado por su propiedad **Value**. Este valor será el nuevo volumen, por lo que tendrá que ser asignado a la propiedad **Volume**. Un razonamiento análogo haremos para controlar el balance, con la diferencia de que la proporción de volumen en los altavoces oscila entre -1 (100% volumen a la izquierda) y 1 (100% volumen a la derecha); el valor predeterminado es 0. Según esto, añada los controladores del evento **ValueChanged** de estos controles y edítelos así:

```
Private Sub sliderVolumen_ValueChanged(sender As Object,
e As RoutedPropertyChangedEventArgs(Of Double))
  If meReproductor IsNot Nothing Then
    meReproductor.Volume = sliderVolumen.Value
  End If
End Sub

Private Sub sliderBalance_ValueChanged(sender As Object,
e As RoutedPropertyChangedEventArgs(Of Double))
  If meReproductor IsNot Nothing Then
    meReproductor.Balance = sliderBalance.Value
  End If
End Sub
```

Nos queda controlar la posición de la reproducción. El rango de valores del **Slider** que permitirá al usuario realizar este control estará entre 0 y la duración del fichero multimedia abierto actualmente, dato que es proporcionado por la propie-

dad **NaturalDuration** del elemento multimedia; este valor no estará disponible mientras no se haya generado el evento **MediaOpened**. Por lo tanto, añada el controlador de este evento y edítelo como se indica a continuación para asignar este valor a la propiedad **Maximum** del **Slider** (el valor **Minimum** fue asignado en el código XAML):

```
Private Sub meReproductor_MediaOpened(sender As Object,
e As RoutedEventArgs)
  sliderPosicion.Maximum =
    meReproductor.NaturalDuration.TimeSpan.TotalSeconds
End Sub
```

La propiedad **NaturalDuration** devuelve un objeto **Duration** y este objeto tiene una propiedad **TimeSpan** que referencia al objeto **TimeSpan** que almacena el tiempo concreto. Entonces, utilizando los miembros de **TimeSpan**, como su propiedad **TotalSeconds**, podremos obtener el valor deseado.

Por otra parte, el **Slider** para controlar la posición de la reproducción tendrá que ser actualizado simultáneamente con el control *tbkEstado*, operación que realiza el controlador del temporizador:

```
Private Sub Temporizador_Tick(sender As Object, e As EventArgs)
  tbkEstado.Text =
    meReproductor.Position.ToString().TrimEnd(New Char() {"0"c})
  sliderPosicion.Value = meReproductor.Position.TotalSeconds
End Sub
```

Ahora, si ejecuta la aplicación, observará que este **Slider** está sincronizado con la posición de la reproducción, pero aún no permite modificar esta posición cuando arrastramos su posición actual a una nueva.

Esto es, el usuario hará clic sobre este **Slider** si desea avanzar o retroceder la posición actual de reproducción. La respuesta de la aplicación a esta acción será poner en pausa la reproducción, operación que podemos realizar desde el controlador del evento **MouseLeftButtonDown** del ratón (cuando el usuario haga clic con el botón primario del ratón sobre el control):

```
Private Sub sliderPosicion_MouseLeftButtonDown(sender As Object,
e As MouseButtonEventArgs)
  btPausa_Click(btPausa, Nothing)
End Sub
```

Con el botón primario del ratón pulsado, el usuario avanzará o retrocederá la posición actual de reproducción, que será efectiva cuando suelte el botón del ratón, momento en el que se genera el evento **MouseLeftButtonUp**. Entonces, ten-

drá que ser el controlador de este evento el que reanude la reproducción desde esta nueva posición, según puede ver en el código mostrado a continuación:

```
Private Sub sliderPosicion_MouseLeftButtonUp(sender As Object,
e As MouseButtonEventArgs)
  meReproductor.Position = TimeSpan.FromSeconds(sliderPosicion.Value)
  meReproductor.Play()
  temporizador.Start()
End Sub
```

Una observación. Cuando añadimos los controladores para los eventos **MouseLeftButtonDown** y **MouseLeftButtonUp** observamos que el primer evento no funcionaba. Esto sucede porque el deslizador maneja estos eventos internamente. Por lo tanto, si queremos manejar estos eventos, al menos para el primero, será necesario utilizar el método **AddHandler** para implementarlos (véase la explicación sobre este método y los eventos en general en el apartado *Eventos enrutados* del capítulo *Introducción a WPF*):

```
Public Sub New()
  InitializeComponent()
  temporizador = New DispatcherTimer()
  temporizador.Interval = TimeSpan.FromSeconds(0.1)
  AddHandler temporizador.Tick, AddressOf Temporizador_Tick
  sliderPosicion.AddHandler(
    MouseLeftButtonDownEvent,
    New MouseButtonEventHandler(
          AddressOf sliderPosicion_MouseLeftButtonDown),
    True)
    sliderPosicion.AddHandler(
      MouseLeftButtonUpEvent,
      New MouseButtonEventHandler(
            AddressOf sliderPosicion_MouseLeftButtonUp),
    True)
  End Sub
```

Cuando el valor del tercer parámetro de **AddHandler** es **True**, se registra el controlador de tal forma que se invoque aunque el evento enrutado esté marcado como administrado en sus datos de evento.

Todo lo explicado sobre el uso de la clase **MediaElement** para reproducir audio, se aplica exactamente igual cuando se trate de reproducir un fichero de vídeo.

La diferencia clave con los ficheros de audio es que en el caso del vídeo las propiedades visuales y las relacionadas con el diseño son importantes. Por ejemplo, el tamaño original del vídeo es proporcionado por las propiedades **NaturalVideoHeight** y **NaturalVideoWidth** de **MediaElement**. También, como vimos

anteriormente, el modo de presentación de un vídeo puede adaptarse a diferentes tamaños mediante la propiedad **Stretch**.

Como ejemplo, vamos a diseñar la interfaz gráfica que muestra la figura siguiente. Para este diseño emplearemos una rejilla con tres columnas y dos filas. En la primera fila, ocupando las tres columnas añadiremos el reproductor, y en la segunda, los botones, uno en cada columna.




```
<MediaElement x:Name="meReproductor"
              Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3"
              Source="/ApSilverlight;component/Media/Butterfly.wmv"
              AutoPlay="False"
              Width="300" Height="300"
              MediaEnded="meReproductor_MediaEnded" />

<Button Grid.Column="0" Grid.Row="1" Content="Reproducir"
        Margin="5" Click="btReproducir_Click"/>
<Button Grid.Column="1" Grid.Row="1" Content="Pausa" Margin="5"
        Click="btPausa_Click" />
<Button Grid.Column="2" Grid.Row="1" Content="Parar" Margin="5"
        Click="btParar_Click" />
```

El código subyacente incluirá los controladores para el evento **Click** de los botones y para el evento **MediaEnded** del reproductor:

```
Partial Public Class SilverlightVideo
  Inherits UserControl
  Public Sub New()
    InitializeComponent()
  End Sub

  Private Sub btReproducir_Click(sender As Object
  e As RoutedEventArgs)
```

```
    meReproductor.Play()
  End Sub

  Private Sub btPausa_Click(sender As Object, e As RoutedEventArgs)
    meReproductor.Pause()
  End Sub

  Private Sub btParar_Click(sender As Object,
  e As RoutedEventArgs)
    meReproductor.Stop()
  End Sub

  Private Sub meReproductor_MediaEnded(sender As Object,
  e As RoutedEventArgs)
    meReproductor.Stop()
  End Sub
End Class
```

Como ejercicio, puede completar el ejemplo anterior para que realice otras operaciones análogas a las que hicimos en el ejemplo de audio.

Finalizamos este apartado informando de que, a partir de la versión 4, Silverlight proporciona soporte para entrada de vídeo y audio, a través de una webcam y el micrófono del usuario. Esto abre la puerta a nuevos tipos de aplicaciones ricas de Internet que soporten videoconferencia, por ejemplo. Para ello, la biblioteca de Silverlight proporciona las clases **CaptureSource**, que proporciona métodos que funcionan con capturas de audio o vídeo específicas del dispositivo de captura asociado, y **VideoCaptureDevice**, que describe el formato de vídeo deseado y compatible de un dispositivo de captura de vídeo, como una cámara web. Ambas clases pertenecen al espacio de nombres **System.Windows.Media**. No obstante, esta aportación, insuficiente para una aplicación profesional, pensamos que es el inicio de todo lo que queda aún por venir.

Este apoyo de audio y vídeo ha permitido la expansión a una nueva característica de Silverlight: *Deep Zoom*. La idea es permitir usar animaciones de tipo muelle. Esto es, piense en una imagen que en su resolución nativa es demasiado grande para mostrarse en la pantalla totalmente. Entonces, con esta técnica se puede mostrar la imagen en un tamaño reducido, de modo que el usuario obtiene una vista panorámica de toda la imagen, para después poder ampliar una zona específica, incluso desplazar la imagen por la vista.

NAVEGACIÓN

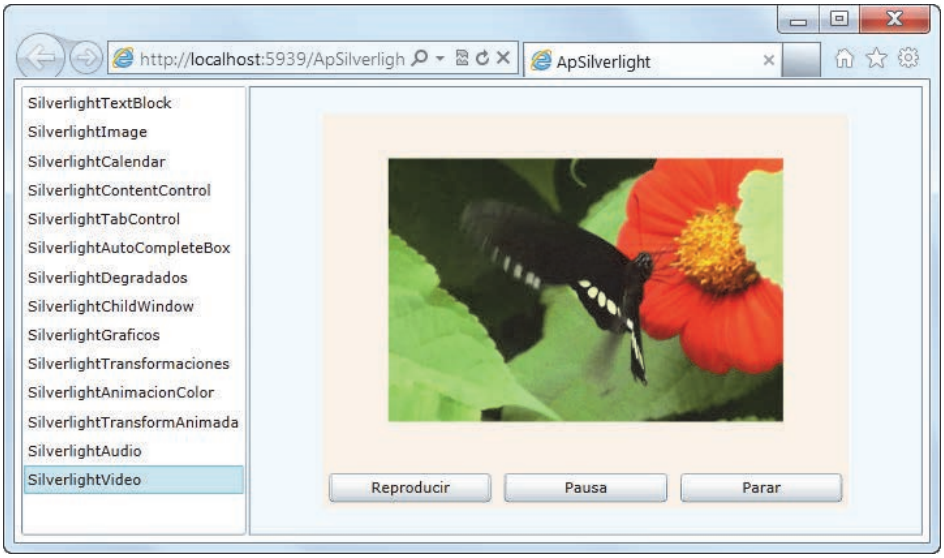
Es importante también saber cómo navegar entre las páginas que componen una aplicación Silverlight. La navegación WPF ya la estudiamos en el capítulo *Nave-*

gación de tipo web. Así que en este apartado, nos limitaremos a aplicar ese conocimiento adaptándolo al sistema de navegación de Silverlight.

En una aplicación Silverlight se pueden utilizar dos estrategias básicas para llevar a cabo los cambios de página: la primera opción es construir un sistema de navegación personalizado mediante la manipulación directa de la interfaz de usuario, y la segunda es utilizar el sistema de navegación de Silverlight.

Navegación personalizada

La opción de construir un sistema de navegación personalizado ya la hemos aplicado en la aplicación que hemos venido desarrollando a partir del apartado *Controles Silverlight*, cuya interfaz mostraba el aspecto siguiente:



Si recuerda, la página principal (*MainPage.xaml*) de esta aplicación estaba formada por un **Grid** con tres columnas: la columna 0 contenía un menú formado por un **ListBox** que nos permitía navegar entre las páginas que se alojaban en un contenedor común (un **Border**), la columna 1 contenía un control **GridSplitter** y la columna 2, el control **Border** que hacía de contenedor, por medio de su propiedad **Child**, para las distintas páginas que componían nuestra aplicación.

Evidentemente, dependiendo del control que utilicemos como contenedor, necesitaremos escribir un código u otro. Por ejemplo, si en lugar de utilizar como contenedor un **Border** utilizamos un **Grid**,

```
<Grid Grid.Column="2" Name="contenedorPag" Background="AliceBlue"/>
```

el código subyacente para cambiar de página sería ahora el siguiente, ya que este control puede contener múltiples controles (no uno sólo) en su colección **Children**. Entonces, necesitamos borrar los elementos de la colección (la página que se está mostrando actualmente) y añadir a la colección la nueva página:

```
contenedorPag.Children.Clear();
contenedorPag.Children.Add(pagina);
```

Navegación de Silverlight

La navegación de Silverlight gira en torno a estos dos controles: **Frame** y **Page**. La idea básica es que las páginas representen el contenido y el marco sirva como contenedor de las páginas facilitando la navegación entre las mismas. Esto es, en un instante determinado, el marco muestra el contenido de una sola página y mostrará el contenido de otra cuando naveguemos a la misma. Esta forma de proceder es la que hemos utilizado con nuestro sistema de navegación manual pero, de esa forma, no teníamos alcance a otras características, como el historial de navegación, que nos proporciona el sistema de navegación de Silverlight.

Para ir describiendo la navegación de Silverlight vamos a crear una nueva aplicación. Para ello, abra Visual Studio o Visual Web Developer y seleccione *Archivo* > *Nuevo* > *Proyecto* > *Visual Basic* > *Silverlight* > *Aplicación de Silverlight*. Denomine al proyecto, por ejemplo, *ApNavegacionSilverlight* y continúe según explicamos al principio de este capítulo.

En estos momentos tenemos una página principal, *MainPage.xaml*, que será creada y mostrada cuando se inicie la aplicación y se cree el objeto *App*. Por lo tanto, éste es el lugar donde implementaremos el sistema de navegación en torno a un control **Frame** de Silverlight. Después, opcionalmente, podemos mostrar diferentes unidades de contenido utilizando objetos **Page**.

Las páginas de contenido las guardaremos en una carpeta *Paginas* que añadiremos ahora al proyecto. Así mismo, para mostrar durante el diseño un código más claro y fácil de analizar, vamos a separar los estilos de los distintos elementos que compondrán la interfaz gráfica en un fichero *Estilos.xaml* que almacenaremos en otra carpeta *Recursos* que añadiremos ahora también al proyecto. Un ejemplo de cómo utilizaremos estos estilos es el siguiente:

```
<Grid x:Name="LayoutRoot" Style="{StaticResource LayoutRootGridStyle}">

</Grid>
```

Para añadir este fichero de recursos a la carpeta *Recursos*, haga clic con el botón secundario del ratón sobre el nombre de la carpeta y seleccione *Agregar* >

Nuevo elemento > Diccionario de recursos Silverlight. Después, seleccione este fichero y en la ventana de propiedades especifique como *Acción de compilación*, *Resource*.

Seguidamente, para poner estos recursos a disposición de toda la aplicación, abra el fichero *App.xaml* y edítelo como se indica a continuación:

```
<Application
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="ApNavegacionSilverlight.App">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Recursos/Estilos.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Para más detalles sobre los diccionarios de recursos, véase el apartado *Recursos* del capítulo *Introducción a WPF*.

Frame

Como ya sabemos, este control se deriva de **ContentControl** y contiene un único elemento de contenido expuesto a través de su propiedad **Content**. Sin embargo, la diferencia de este control con respecto a otros controles de contenido es que podemos cambiar su contenido invocando a su método **Navigate**. Este método cambia el valor de la propiedad **Content** y, además, desencadena el servicio de navegación responsable de registrar la historia de navegación.

Por ejemplo, vamos a añadir a *MainPage.xaml* un **Border** que contenga un **Frame**. En realidad, no es necesario utilizar un **Border**, a no ser que tengamos que aplicar alguna propiedad, como **CornerRadius**, pero facilita el diseño.

```
<UserControl x:Class="ApNavegacionSilverlight.MainPage"
    xmlns="http://.../xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sdk="http://.../xaml/presentation/sdk">

  <Grid x:Name="LayoutRoot"
        Style="{StaticResource LayoutRootGridStyle}">
    <Border x:Name="ContentBorder"
            Style="{StaticResource ContentBorderStyle}">
      <sdk:Frame x:Name="ContentFrame"
                 Style="{StaticResource ContentFrameStyle}">
```

```
      </sdk:Frame>
    </Border>
  </Grid>
</UserControl>
```

Obsérvese que para poder utilizar la clase **Frame** se ha asignado un espacio de nombres XAML *.../presentation/sdk* al prefijo *sdk*.

Actualmente el contenido de **Frame** está vacío. Vamos a hacer que su contenido sea una página *Inicio.xaml*. Para ello, haga clic con el botón secundario del ratón sobre el nombre de la carpeta *Paginas* y seleccione *Agregar > Nuevo elemento > Página de Silverlight*. Después, añada al **Grid** un **TextBlock** con un texto que especifique que se trata de la página de inicio.

```
<navigation:Page x:Class="ApNavegacionSilverlight.Inicio"
  xmlns="http://.../xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;
                    assembly=System.Windows.Controls.Navigation"
  Title="Página Inicio">

  <Grid x:Name="LayoutRoot">
    <TextBlock Text="Esta es la página de inicio (Inicio.xaml)"/>
  </Grid>
</navigation:Page>
```

Obsérvese que para poder utilizar la clase **Page** se ha asignado el espacio de nombres del CLR System.Windows.Controls.**Navigation** del ensamblado *Navigation*, al prefijo espacio de nombres *navigation*.

Finalmente, para navegar a esta página tenemos varias opciones:

- Desde el código subyacente se podría invocar al método **Navigate** del objeto **Frame** pasando como argumento el **Uri** de la página. Observe que el URI siempre empieza por /, la cual representa la raíz de la aplicación:

  ```
  ContentFrame.Navigate(New Uri("/Paginas/Inicio.xaml",
                                UriKind.Relative))
  ```

  En este ejemplo, cuando se invoca al método **Navigate**, Silverlight crea un objeto de la clase *Inicio*, lo asigna al contenido de **Frame** y registra la acción en el historial de navegación.

- O bien, desde el código subyacente se podría asignar a la propiedad **Source** del objeto **Frame** el **Uri** de la página:

```
ContentFrame.Source = New Uri("/Paginas/Inicio.xaml",
                              UriKind.Relative)
```

- También, desde el código XAML se podría asignar a la propiedad **Source** del objeto **Frame** el URI de la página:

```
<sdk:Frame x:Name="ContentFrame"
           Style="{StaticResource ContentFrameStyle}"
           Source="/Paginas/Inicio.xaml" >
```

Como lo normal es que la aplicación pueda navegar a varias páginas, una mejor opción puede ser utilizar un control **HyperlinkButton** por cada una de ellas, según muestra el código siguiente:

```
<Grid x:Name="LayoutRoot" ...>
  <Border x:Name="ContentBorder" ...>
    <sdk:Frame x:Name="ContentFrame" ...>
    </sdk:Frame>
  </Border>
  <Grid x:Name="NavigationGrid"
        Style="{StaticResource NavigationGridStyle}">
    <Border x:Name="LinksBorder"
            Style="{StaticResource LinksBorderStyle}">
      <StackPanel x:Name="LinksStackPanel"
                  Style="{StaticResource LinksStackPanelStyle}">
        <HyperlinkButton x:Name="Link1" Content="Inicio"
                         Style="{StaticResource LinkStyle}"
                         NavigateUri="/Paginas/Inicio.xaml"
                         TargetName="ContentFrame"/>
      <StackPanel/>
    </Border>
  </Grid>
</Grid>
```

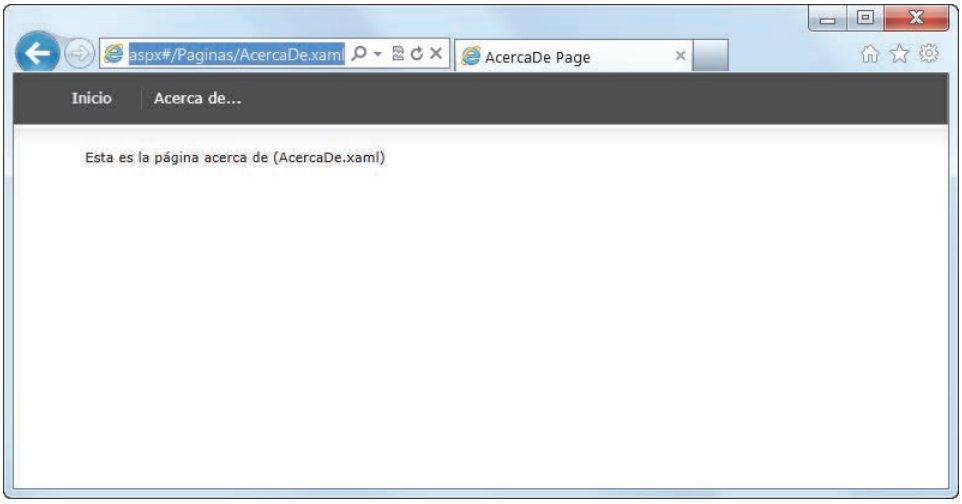
Si hemos puesto el **HyperlinkButton** dentro de un **StackPanel** y éste dentro de un **Border** y éste, a su vez, dentro de otro **Grid**, es simplemente por dos cuestiones: para poder añadir más **HyperlinkButton** (un **Border** sólo admite un elemento) y por facilitar el diseño.

La propiedad **Name** de **HyperlinkButton** especifica el identificador para dirigirse al mismo desde el código, **Content** especifica el título que mostrará el botón, **Style** es el estilo aplicado al botón, **NavigateUri** especifica el URI de destino cuando se hace clic en el botón y **TargetName** hace referencia al marco de destino en el que se va a abrir la página.

Como ejemplo, añada una página más denominada *AcercaDe.xaml* y verifique cómo trabaja el sistema de navegación de Silverlight. Esto implica añadir otro **HyperlinkButton** que permita al usuario navegar a esta página:

```
<StackPanel x:Name="LinksStackPanel"
            Style="{StaticResource LinksStackPanelStyle}">
  <HyperlinkButton x:Name="Link1" Content="Inicio"
                   Style="{StaticResource LinkStyle}"
                   NavigateUri="/Paginas/Inicio.xaml"
                   TargetName="ContentFrame"/>
  <Rectangle Style="{StaticResource DividerStyle}"/>
  <HyperlinkButton x:Name="Link2" Content="Acerca de..."
                   Style="{StaticResource LinkStyle}"
                   NavigateUri="/Paginas/AcercaDe.xaml"
                   TargetName="ContentFrame"/>
</StackPanel>
```

Observe que hemos añadido como separador un pequeño **Rectangle**.



Ejecute ahora la aplicación y observe que cuando se cambia el contenido de un control **Frame**, utilizando el sistema de navegación de Silverlight, el nombre del recurso XAML se anexa al actual URI, después del marcador de fragmento (#). Este hiperenlace (*pagina.aspx#pagina.xaml*) que lleva a una página de un sitio web, en vez de llevar a la página principal de ese sitio, se denomina enlace profundo (*deep link*). Esto puede ser un problema si un usuario lo utiliza para saltarse la página principal y no deseamos que sea así, pero también aporta la ventaja de que todas las páginas tienen un URI distinto, lo que diferencia a una página de otra en el historial de navegación. No obstante, podemos manejar el evento **Navigating** de **Frame** y verificar de qué página se trata, pudiendo cancelar la navega-

ción para las páginas que deseemos. Por ejemplo, el siguiente controlador evitaría el intento de acceder directamente a una página que consideramos prohibida:

```
Private Sub ContentFrame_Navigating(sender As Object,
e As System.Windows.Navigation.NavigatingCancelEventArgs)
  If e.Uri.ToString().ToLower().Contains("pagProhibida.xaml") Then
    e.Cancel = True
  End If
End Sub
```

Evidentemente, estamos asumiendo que nunca vamos a navegar a esa página, quizás porque la estemos utilizando con otros fines (por ejemplo, porque contiene código utilizado para otros propósitos), ya que el sistema de navegación no distingue entre si ha sido el usuario el que ha introducido directamente la URI o si hizo clic en un botón de enlace.

Ahora puede utilizar los botones de navegación situados en la esquina superior izquierda del explorador para retroceder a la página anterior o avanzar a la siguiente, si es posible. Pero aquí Silverlight presenta un problema y es que cuando se intenta retroceder a la página anterior a la de inicio (en nuestro ejemplo a *ApNavegacionSilverlightTestPage.aspx*) se produce un error, ya que esta página no incluye el fragmento utilizado por el historial de navegación, por lo que no puede recuperarla. La solución pasa por controlar el evento **NavigationFailed**:

```
Private Sub ContentFrame_NavigationFailed(sender As Object,
e As System.Windows.Navigation.NavigationFailedEventArgs)
  ' System.ArgumentException
  If TypeOf e.Exception Is ArgumentException Then
    e.Handled = True
    System.Diagnostics.Debug.WriteLine(e.Exception.Message)
  End If
End Sub
```

Para especificar si un marco se integra con la navegación del explorador web, se utiliza la propiedad **JournalOwnership**. El valor predeterminado para esta propiedad es *Automatic*, lo que significa que el marco se integrará con el diario del explorador si es un marco de nivel superior. Si no desea esto, asigne a esa propiedad el valor *OwnsJournal*, lo que significa que el marco mantiene su propio diario de navegación.

Administrador de identificadores de recursos

Según hemos podido ver en el apartado anterior, algunos URI utilizan un carácter "#" seguido de un identificador de recurso (llamado identificador de fragmento) para referirse a un determinado recurso, en nuestro caso a una página:

```
http://SitioWeb/ApNavegacionSilverlightTestPage.aspx#/Paginas/AcercaDe.xaml
```

En este ejemplo, vemos que se pone el nombre de la página en el URI. Pero, quizás, en alguna situación no desee difundir esta información, o simplemente desea utilizar un URI más sencillo de memorizar. Aquí es donde desarrolla su función el *administrador de identificadores de recursos* de Silverlight. Se trata de un objeto que administra la conversión de un identificador URI en otro URI que será utilizado en un **Frame**.

Normalmente, este objeto será de la clase **UriMapper**, derivada de **UriMapperBase** del espacio de nombres **System.Windows.Navigation**, ya que proporciona la implementación más común. Un objeto de esta clase almacena una colección de objetos **UriMapping**, cada uno de los cuales convierte un URI solicitado en otro URI.

Frame tiene una propiedad **UriMapper** que hace referencia a este objeto administrador de URI que se implementa así:

```
<sdk:Frame x:Name="ContentFrame"
           Style="{StaticResource ContentFrameStyle}"
           Source="/Paginas/Inicio.xaml"
           Navigating="ContentFrame_Navigating"
           NavigationFailed="ContentFrame_NavigationFailed">
  <sdk:Frame.UriMapper>
    <sdk:UriMapper>
      <sdk:UriMapping Uri="" MappedUri="/Paginas/Inicio.xaml"/>
      <sdk:UriMapping Uri="/{nombrePagina}"
                      MappedUri="/Paginas/{nombrePagina}.xaml"/>
    </sdk:UriMapper>
  </sdk:Frame.UriMapper>
</sdk:Frame>
```

En el código anterior se pueden observar los objetos **UriMapping** que definen los modelos para convertir un identificador URI solicitado en un nuevo URI, para lo cual utilizan sus propiedades **Uri** y **MappedUri**. **Uri** especifica el modelo de un URI para que coincida con el URI especificado por **MappedUri** que realmente indica el recurso al que se desea navegar.

Según lo expuesto, el URI:

```
http://SitioWeb/PaginaWeb.aspx
```

será resuelto según el primer **UriMapping** del código anterior así:

```
http://SitioWeb/PaginaWeb.aspx#/Paginas/Inicio.xaml
```

La correspondencia ha sido de nada ("") a */Paginas/Inicio.xaml*. El URI:

```
http://SitioWeb/PaginaWeb.aspx#/Inicio
```

será resuelto según el segundo **UriMapping** del código anterior así:

```
http://SitioWeb/PaginaWeb.aspx#/Paginas/Inicio.xaml
```

donde se ha sustituido *{nombrePagina}* por *Inicio*. El URI:

```
http://SitioWeb/PaginaWeb.aspx#/AcercaDe
```

será resuelto también según el segundo **UriMapping** del código anterior así:

```
http://SitioWeb/PaginaWeb.aspx#/Paginas/AcercaDe.xaml
```

donde se ha sustituido *{nombrePagina}* por *AcercaDe*.

Obsérvese que el URI propuesto se corresponde con la parte del URI que pertenece a la navegación dentro del marco que contiene las asignaciones; esto es, con la parte que está después del delimitador de fragmentos (#). Esto hace posible que la aplicación Silverlight siga funcionando correctamente aunque se mueva a una página web distinta o se ejecute como aplicación fuera del explorador.

La solicitud de navegar a un URI determinado se resuelve con el primer modelo que coincida con la solicitud. Por lo tanto, hay que agregar los objetos de asignación de URI (**UriMapping**) ordenados de más específicos a más generales.

Los segmentos, expresiones encerradas ente {}, actúan como variables, lo que nos permite especificar también URI que toman argumentos. Por ejemplo, se puede especificar un objeto **UriMapping** también así:

```
<sdk:UriMapping Uri="Asignaturas/{tipo}"
  MappedUri="/Pags/DetalleAsignatura.xaml?tipoasignatura={tipo}">
</sdk:UriMapping>
```

El modelo *Asignaturas/{tipo}* coincidirá con las solicitudes que incluyan *Asignaturas* seguido por un segmento. Los identificadores URI coincidentes pueden incluir *Asignaturas/obligatorias*, *Asignaturas/optativas* u otra solicitud similar. El valor especificado en segundo lugar (*obligatorias* u *optativas*) se incluye en la cadena de consulta del URI asignado en el lugar *{tipo}*.

Continuando con el desarrollo de nuestra aplicación, ¿cómo son ahora los URI propuestos por los **HyperlinkButton**? Veamos:

```
<HyperlinkButton x:Name="Link1" Content="Inicio"
                 Style="{StaticResource LinkStyle}"
                 NavigateUri="/Inicio"
                 TargetName="ContentFrame"/>
<Rectangle Style="{StaticResource DividerStyle}"/>
<HyperlinkButton x:Name="Link2" Content="Acerca de..."
                 Style="{StaticResource LinkStyle}"
                 NavigateUri="/AcercaDe"
                 TargetName="ContentFrame"/>
```

La propiedad **Source** de **Frame** también tendrá asignado el valor */Inicio* o, simplemente, nada.

Cuando utilizamos un objeto **UriMapper** para establecer la página inicial,

```
<sdk:UriMapping Uri="" MappedUri="/Paginas/Inicio.xaml"/>
```

no es necesario controlar el evento **NavigationFailed** para gestionar la excepción **ArgumentException** porque ésta no se lanzará, ya que con esta técnica el URI:

```
http://SitioWeb/PaginaWeb.aspx
```

será resuelto según el primer **UriMapping** del **UriMapper** así:

```
http://SitioWeb/PaginaWeb.aspx#/Paginas/Inicio.xaml
```

Navegación externa

Para habilitar la navegación a otras páginas webs, puede usar también el control **HyperlinkButton**, establecer la propiedad **NavigateUri** con el recurso externo y establecer la propiedad **TargetName** para abrir una nueva ventana del explorador. Por ejemplo, vamos a añadir a la página *AcercaDe* un enlace que permita navegar a la página web del autor:

```
<navigation:Page x:Class="ApNavegacionSilverlight.AcercaDe"
                 ...>
  <Grid x:Name="LayoutRoot">
    <StackPanel>
      <TextBlock Text="Esta es la página acerca de."/>
      <HyperlinkButton NavigateUri="http://www.fjceballos.es"
                       Content="Mi Web" TargetName="_new" />
    </StackPanel>
  </Grid>
</navigation:Page>
```

También, si lo desea, puede deshabilitar la navegación externa. Para ello sólo tiene que establecer la propiedad **enableNavigation** en *none* al iniciar el comple-

mento Silverlight. El valor predeterminado para esta propiedad es *all*, lo que permite navegar a cualquier URI. Por ejemplo, para deshabilitar la navegación externa en la aplicación que estamos desarrollando, abra el fichero *.aspx* y añada la siguiente línea de código:

```
<body>
  <form id="form1" runat="server" style="height: 100%">
    <div id="silverlightControlHost">
      <object
        ...
        <param name="enableNavigation" value="none" />
        ...
    </div>
  </form>
</body>
```

Extender el sistema de navegación

En Silverlight es posible extender el sistema de navegación estableciendo la propiedad **ContentLoader** en una implementación **INavigationContentLoader** personalizada. Por ejemplo, esto sería necesario si quisiéramos añadir a nuestra aplicación, además de la navegación, la autenticación y autorización, lo que permitiría especificar a qué páginas pueden navegar los usuarios, pero este tema se sale de los objetivos de este capítulo. No obstante, sepa que Visual Studio incluye la plantilla *Aplicación de negocios Silverlight* que incluye esta funcionalidad.

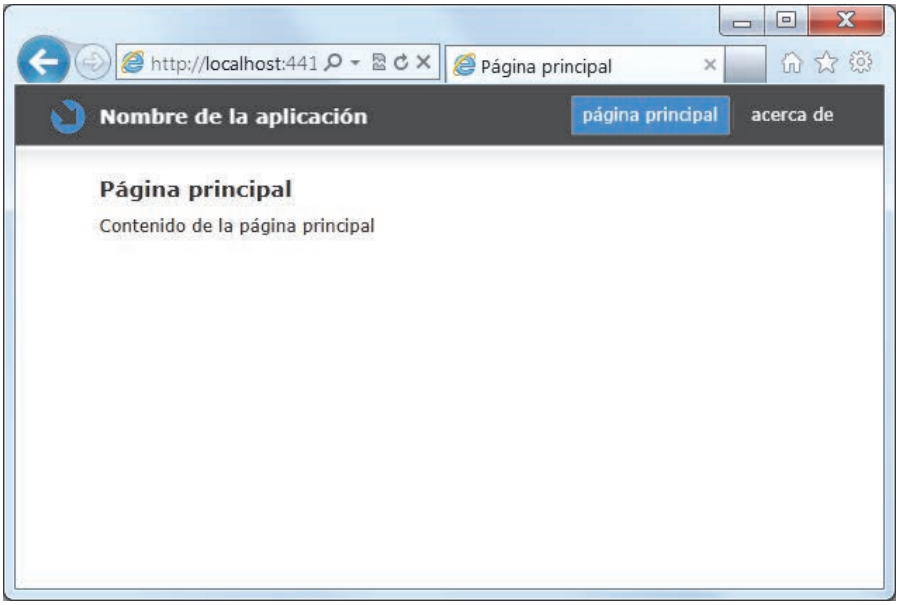
Compatibilidad de ejecución fuera del explorador

Una aplicación Silverlight se puede configurar también para que los usuarios la puedan ejecutar fuera del explorador. La configuración consiste sencillamente en proporcionar información adicional sobre una aplicación, que será almacenada en el manifiesto de la misma, necesaria para poder habilitar la compatibilidad con la ejecución fuera del explorador en una aplicación existente sin tener que volver a compilarla.

Para realizar esta configuración, haga clic con el botón secundario del ratón sobre el nombre del proyecto Silverlight y seleccione *Silverlight > Habilitar la ejecución de aplicación fuera del explorador > Configuración de la ejecución fuera del explorador*. Se muestra el diálogo *Configuración de la ejecución fuera del explorador* que tendrá que completar con los datos solicitados, al menos, en lo que se refiere a los campos obligatorios: *título de la ventana*, *nombre del acceso directo* y *descripción de la aplicación*.

Plantilla aplicación de navegación de Silverlight

Visual Studio incluye la plantilla *Aplicación de navegación de Silverlight* que genera automáticamente el esqueleto para una aplicación Silverlight análoga a la desarrollada anteriormente. Como ejemplo, utilizando esta plantilla, genere un nuevo proyecto *ApNavegacionSilverlight2*. Si ahora ejecuta esta aplicación, observará el siguiente resultado:



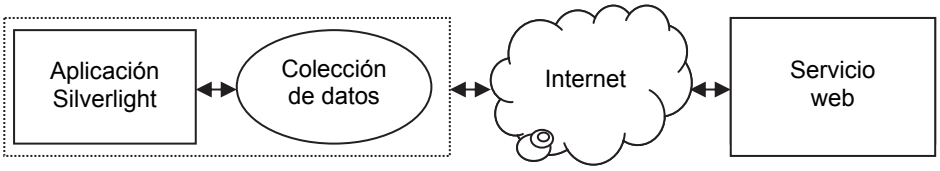
Observando la página principal comprobamos que contiene componentes de la interfaz de usuario permanentes y un marco para el contenido navegable. En Internet puede localizar otras plantillas para generar este tipo de aplicaciones.

ACCESO A DATOS

Las aplicaciones Silverlight pueden obtener acceso a los datos utilizando diferentes tecnologías, como colecciones de datos, datos XML, servicios de datos de WCF, entre otras, o una combinación de éstas.

Normalmente, las aplicaciones de Silverlight funcionan en un espacio de seguridad de confianza parcial, que a veces se denomina espacio aislado; esto es, un entorno de ejecución de aplicaciones con privilegios limitados. Debido a esto, no pueden obtener acceso al sistema de ficheros ni a otros recursos del equipo del cliente de la misma manera que lo hacen las aplicaciones tradicionales de .NET, aunque estas acciones, a iniciativa del usuario, aún podrían llevarse a cabo. Así

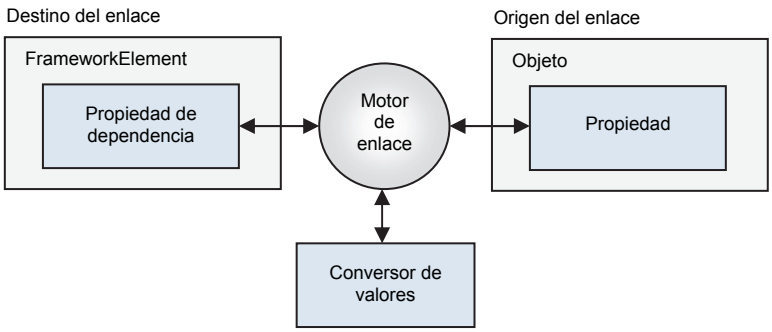
que, lo lógico es acceder a los datos a través de servicios webs. Esto es, una aplicación Silverlight, tras una llamada a los métodos que expone la interfaz de un servicio web, obtendría el conjunto de datos requerido y lo pondría a disposición de las páginas de la aplicación. Esta es la manera cómo Silverlight maneja el acceso a los datos, interactuando con colecciones de datos que normalmente recibe de servicios webs.



La interfaz gráfica del usuario de la aplicación y la lógica de negocio conectarán con los datos de la colección por medio de enlaces a datos. Tanto los enlaces a datos como las colecciones fueron expuestos en los capítulos precedentes, con lo que aquí nos limitaremos a aplicar los conceptos aprendidos adaptándolos según las variaciones que introduzca Silverlight.

Acceso a los datos de una colección

El enlace de datos, según hemos visto en los capítulos anteriores, proporciona un método simple y ágil para que las aplicaciones en general y las aplicaciones Silverlight en particular presenten datos e interactúen con ellos. Por lo estudiado hasta ahora sabemos que un enlace entre un elemento de la interfaz de usuario y un objeto de datos permite que los datos fluyan entre ambos, y que cuando estos datos cambian en uno u otro extremo del enlace, dependiendo del modo en el que se haya establecido el mismo, los elementos que están enlazados reflejarán los cambios de forma automática. Todo esto está fundamentado básicamente en un objeto **Binding** que hace de puente entre un origen y un destino permitiendo que fluyan los datos entre ambos, según ilustra la figura siguiente.



Para que los datos fluyan entre los dos extremos del enlace, el motor de enlace obtiene información del objeto **Binding** que define el enlace, sobre la propiedad del elemento de la interfaz de usuario destino de los datos, sobre la propiedad del objeto origen de los datos (que puede ser cualquier objeto CLR o, incluso, el propio elemento de destino u otros elementos de la interfaz de usuario), sobre la dirección del flujo de datos (propiedad **Mode** del objeto **Binding**) y, opcionalmente, sobre el conversor de valores que se aplica a los datos mientras se pasan.

El origen se establece en el código, generalmente, mediante el establecimiento de la propiedad **DataContext** (contexto de datos) de un elemento **FrameworkElement** o de la propiedad **ItemSource** de un elemento **ItemsControl**.

El contexto de datos se hereda, lo que resulta de gran utilidad si desea tener varios enlaces y que todos ellos usen el mismo origen; ahora bien, un elemento secundario puede invalidar este comportamiento estableciendo la propiedad **Source** en su objeto de enlace o estableciendo su contexto **DataContext**, que se aplicará a todos sus elementos secundarios.

En cuanto a la dirección del flujo de datos, recordar que cada enlace tiene una propiedad **Mode** que determina cómo y cuándo fluyen los datos. Silverlight permite tres tipos de enlaces: *OneTime*, *OneWay* y *TwoWay*.

Un enlace *OneTime* actualiza el destino con los datos del origen cuando se crea el enlace. Un enlace *OneWay* actualiza el destino con los datos del origen cuando se crea el enlace y en cualquier momento en que cambian los datos; es el modo predeterminado. Y un enlace *TwoWay* actualiza tanto el destino como el origen cuando cualquiera de ellos cambia. Ahora bien, como ya estudiamos, para que tenga lugar la actualización automática del destino, el objeto origen debe implementar la interfaz **INotifyPropertyChanged**.

Un objeto origen del enlace puede ser un objeto único, cuyas propiedades contienen datos, o bien una colección de objetos. Por ejemplo, el origen podría ser un objeto de la clase *Telefono* que defina los atributos identificador de la persona, nombre, dirección, número de teléfono, observaciones y foto de una persona.

```
Public Class Telefono
  Implements INotifyPropertyChanged

  Private m_idPersona As Integer
  Public Property IdPersona() As Integer
    Get
      Return m_idPersona
    End Get
    Set(value As Integer)
      m_idPersona = value
```

```
    OnPropertyChanged(New PropertyChangedEventArgs("IdPersona"))
  End Set
End Property

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

Private m_direccion As String
Public Property Direccion() As String
  Get
    Return m_direccion
  End Get
  Set(value As String)
    m_direccion = value
    OnPropertyChanged(New PropertyChangedEventArgs("Direccion"))
  End Set
End Property

Private m_telefono As String
Public Property NumTelefono() As String
  Get
    Return m_telefono
  End Get
  Set(value As String)
    m_telefono = value
    OnPropertyChanged(New PropertyChangedEventArgs("Telefono"))
  End Set
End Property

Private m_observaciones As String
Public Property Observaciones() As String
  Get
    Return m_observaciones
  End Get
  Set(value As String)
    m_observaciones = value
    OnPropertyChanged(New PropertyChangedEventArgs("Observaciones"))
  End Set
End Property

Private m_foto As Byte()
Public Property Foto() As Byte()
```

```
    Get
      Return m_foto
    End Get
    Set(value As Byte())
      m_foto = value
      OnPropertyChanged(New PropertyChangedEventArgs("Foto"))
    End Set
  End Property

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged
  Public Sub OnPropertyChanged(e As PropertyChangedEventArgs)
    RaiseEvent PropertyChanged(Me, e) ' generar evento
  End Sub

  Public Overrides Function ToString() As String
    Return Nombre
  End Function

  Public Sub New(id As Integer)
    IdPersona = id
    Nombre = "¿Nombre?"
    Direccion = "¿Dirección?"
    NumTelefono = "000000000"
    Observaciones = "Ninguna"
    Foto = Nothing
  End Sub

  Public Sub New(id As Integer, nom As String, dir As String,
  tel As String, obs As String)
    IdPersona = id
    Nombre = nom
    Direccion = dir
    NumTelefono = tel
    Observaciones = obs
    Foto = Nothing
  End Sub
End Class
```

O bien podría ser una colección *Telefonos* de objetos *Telefono* definida así:

```
Public Class Telefonos
  Inherits List(Of Telefono)

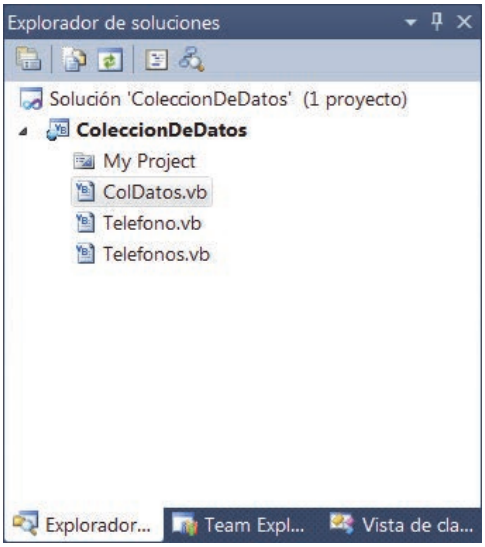
End Class
```

Basándonos en estas definiciones, vamos a simular una colección *Telefonos* que supuestamente se construiría a partir de una base de datos para realizar una aplicación Silverlight que muestre una interfaz gráfica que interactúe con esta co-

lección. La aplicación mostrará en una página, bien una lista de personas, o bien persona a persona, los datos de las personas almacenadas en la colección, permitirá añadir nuevas personas, borrarlas o modificarlas, y añadirá posibilidades de clasificación y filtrado de la información mostrada.

Crear la base de datos

Lo lógico es que los datos estuvieran en una base de datos, para lo cual tendríamos que diseñar y construir dicha base. Una vez construida, utilizando *Entity Framework*, asignaríamos el modelo de datos relacional a un modelo de objetos expresado en el lenguaje de programación Visual Basic, para después realizar las consultas sobre este modelo de objetos. En este ejemplo, vamos a construir un modelo de objetos más sencillo que suplante a ese otro modelo de objetos. Para ello, vamos a crear un nuevo proyecto de tipo *Biblioteca de clases Silverlight* que denominaremos *ColeccionDeDatos* según muestra la figura siguiente:



La clase *ColDatos* tiene un constructor que invocando al método privado *ObtenerTelefonos* construye una colección de objetos *Telefono* con datos ficticios.

```
Public Class ColDatos
  Implements INotifyPropertyChanged
  Public Sub New()
    ObtenerTelefonos()
  End Sub

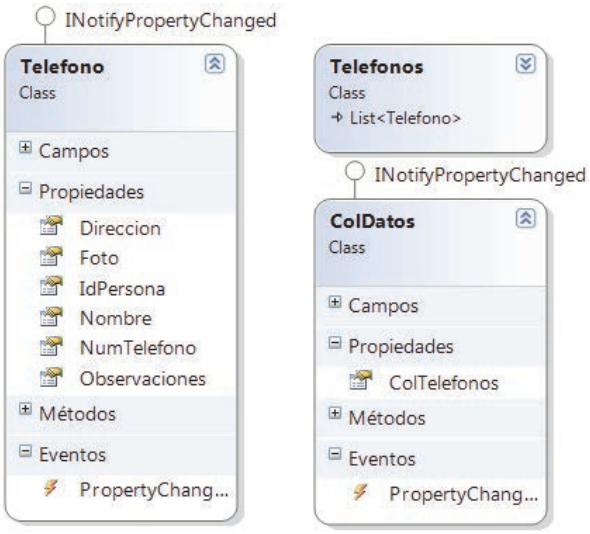
  Private m_colTelefonos As Telefonos
  Public Property ColTelefonos() As Telefonos
    Get
```

```
      Return m_colTelefonos
    End Get
    Set(value As Telefonos)
      m_colTelefonos = value
      OnPropertyChanged(New PropertyChangedEventArgs("ColTelefonos"))
    End Set
  End Property

  Public Event PropertyChanged As PropertyChangedEventHandler _
    Implements INotifyPropertyChanged.PropertyChanged
  Public Sub OnPropertyChanged(e As PropertyChangedEventArgs)
    RaiseEvent PropertyChanged(Me, e) ' generar evento
  End Sub

  ' Crear la base de datos
  Private Function ObtenerTelefonos() As Telefonos
    m_colTelefonos = New Telefonos()
    Dim prefijo As Integer = 636
    For i As Integer = 1 To 29
      Dim nom As String = "Persona " & String.Format("{0:D3}", i)
      Dim dir As String = "Dirección " & String.Format("{0:D3}", i)
      Dim tel As String = prefijo & "333" & String.Format("{0:D3}", i)
      Dim obs As String = "Observación" & String.Format("{0:D3}", i)
      If i Mod 10 = 0 Then
        prefijo += 10
      End If
      m_colTelefonos.Add(New Telefono(i, nom, dir, tel, obs))
    Next
    Return m_colTelefonos
  End Function
End Class
```

Esta biblioteca de clases pone a disposición de cualquier aplicación que requiera utilizarla el siguiente modelo de objetos:



Crear una aplicación Silverlight

El siguiente paso es crear la estructura básica de la aplicación Silverlight que accederá al modelo de objetos, representativo de la base de datos, que acabamos de generar. Para ello, según explicamos anteriormente, abra Visual Studio y cree un nuevo proyecto de tipo *Visual Basic > Silverlight > Aplicación de Navegación de Silverlight*. Denomine al proyecto, por ejemplo, *SilverlightAccesoColDatos* y continúe según explicamos anteriormente en este capítulo.

Para poder utilizar el modelo de objetos anterior en este proyecto, añada al mismo una referencia a la biblioteca de clases *ColeccionDeDatos*. Esto le resultará mucho más sencillo y, además, podrá realizar cambios sobre el modelo de objetos si agrega a este proyecto, el proyecto *ColeccionDeDatos*.

Para que esta aplicación Silverlight pueda acceder a los datos proporcionados por la biblioteca *ColeccionDeDatos*, simplemente tiene que crear un objeto *ColDatos*; la propiedad *ColTelefonos* de este objeto le dará acceso a la colección. Según esto, añada a la clase *App* de la aplicación Silverlight una propiedad *ColTelefonos* que permita el acceso a la colección de datos:

```
Partial Public Class App
  Inherits Application
  Private m_ColTelefonos As Telefonos
  Public Property ColTelefonos() As Telefonos
    Get
      Return m_ColTelefonos
    End Get
```

```
    Private Set(value As Telefonos)
      m_ColTelefonos = value
    End Set
  End Property

  Public Sub New()
    ColTelefonos = New ColDatos().ColTelefonos
    InitializeComponent()
  End Sub
  ' ...
End Class
```

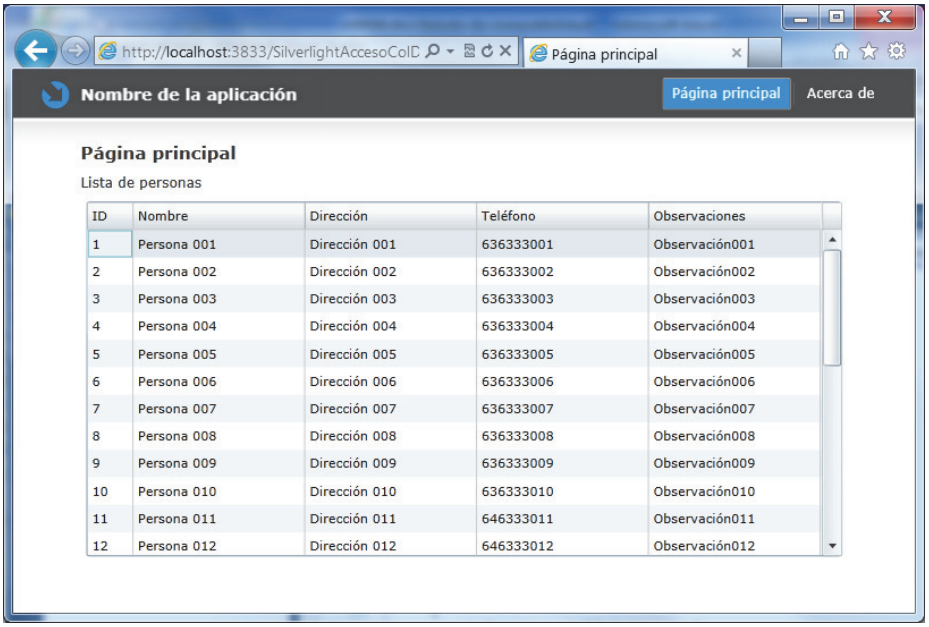
Ahora, cualquier página de la aplicación que quiera acceder a la colección de datos simplemente tendrá que utilizar el siguiente código:

```
TryCast(Application.Current, App).ColTelefonos
```

El siguiente paso es diseñar la interfaz gráfica de la aplicación *SilverlightAccesoColDatos*. Los elementos de esta interfaz se proveerán de los datos proporcionados por esa colección.

Vincular controles con el origen de datos

La aplicación, inicialmente, mostrará una página con una interfaz que permita mostrar una lista con todas las personas almacenadas en la colección.



ID	Nombre	Dirección	Teléfono	Observaciones
1	Persona 001	Dirección 001	636333001	Observación001
2	Persona 002	Dirección 002	636333002	Observación002
3	Persona 003	Dirección 003	636333003	Observación003
4	Persona 004	Dirección 004	636333004	Observación004
5	Persona 005	Dirección 005	636333005	Observación005
6	Persona 006	Dirección 006	636333006	Observación006
7	Persona 007	Dirección 007	636333007	Observación007
8	Persona 008	Dirección 008	636333008	Observación008
9	Persona 009	Dirección 009	636333009	Observación009
10	Persona 010	Dirección 010	636333010	Observación010
11	Persona 011	Dirección 011	646333011	Observación011
12	Persona 012	Dirección 012	646333012	Observación012

El proyecto contiene ya una página denominada *Home*. Realice el diseño de esta página para que muestre un **DataGrid** con una columna por cada atributo del objeto *Telefono*.

Para enlazar el **DataGrid** a los datos, establezca su propiedad **ItemsSource** con el objeto colección. De esta forma, cada fila de la cuadrícula de datos quedará enlazada automáticamente a su respectivo objeto en el origen de datos. A continuación, tendremos que enlazar cada una de las columnas de la cuadrícula de datos con su propiedad respectiva en el objeto de datos:

```
<Grid x:Name="LayoutRoot">
  <StackPanel x:Name="ContentStackPanel">
    <TextBlock x:Name="HeaderText" Text="Página principal"
               Style="{StaticResource HeaderTextStyle}"/>
    <TextBlock x:Name="ContentText" Text="Lista de personas"
               Style="{StaticResource ContentTextStyle}"/>

    <sdk:DataGrid Name="dgPersonas" Margin="5" MaxHeight="300"
        AutoGenerateColumns="False"
        ItemsSource="{Binding}">
      <sdk:DataGrid.Columns>
        <sdk:DataGridTextColumn Header="ID"
        Binding="{Binding Path=IdPersona}" MinWidth="40"/>
        <sdk:DataGridTextColumn Header="Nombre"
        Binding="{Binding Path=Nombre}" MinWidth="150"/>
        <sdk:DataGridTextColumn Header="Dirección"
        Binding="{Binding Path=Direccion}" MinWidth="150"/>
        <sdk:DataGridTextColumn Header="Teléfono"
        Binding="{Binding Path=NumTelefono}" MinWidth="150"/>
        <sdk:DataGridTextColumn Header="Observaciones"
        Binding="{Binding Path=Observaciones}" MinWidth="150"/>
      </sdk:DataGrid.Columns>
    </sdk:DataGrid>

  </StackPanel>
</Grid>
```

ItemsSource="{Binding}" especifica que se utilizará como origen de datos el contexto actual heredado del elemento primario donde se haya establecido; en este caso, lo estableceremos en el **Grid**. Para ello, abra el fichero de código subyacente de esta página y añada las sentencias que se muestran a continuación:

```
Imports System.Windows.Data

Partial Public Class Home
  Inherits Page
  Private fuenteDeDatos As New CollectionViewSource()
```

```
  Public Sub New()
    InitializeComponent()
    fuenteDeDatos.Source =
      TryCast(Application.Current, App).ColTelefonos
    LayoutRoot.DataContext = fuenteDeDatos
  End Sub

  ' ...
End Class
```

Observe el origen de datos que hemos asignado al contexto de datos del **Grid**: un objeto **CollectionViewSource** vinculado con la colección. El objeto **CollectionViewSource** es un *proxy* para acceder a un objeto que implemente la interfaz **ICollectionView** que representa la vista de la colección. Trabajar con una vista de colección facilita agregar y quitar elementos en lugar de trabajar directamente con la colección de origen, y desacopla el origen de datos de la interfaz gráfica, por lo que será mucho más fácil cambiar posteriormente las fuentes.

Recuerde, para crear una vista de colección para una colección que implementa la interfaz **IEnumerable**, lo recomendado es crear un objeto **CollectionViewSource**, agregar la colección a su propiedad **Source** y obtener la vista de colección de su propiedad **View**.

A diferencia de WPF, Silverlight no incluye la clase **CollectionView** utilizada en los capítulos anteriores, pero incluye otra clase que hace básicamente la misma función: **PagedCollectionView** del espacio de nombres **System.Windows.Data**. Esta clase, análogamente a **CollectionView**, representa una vista para agrupar, ordenar, filtrar y navegar por una colección de datos que implemente la interfaz **IEnumerable**.

Por ejemplo, vamos a modificar el código anterior con el fin de utilizar esta clase. En este caso, para construir la vista de una colección, tenemos que crear un objeto **PagedCollectionView** pasando como argumento la colección:

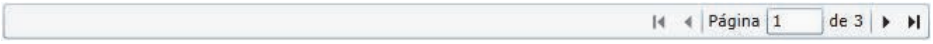
```
Partial Public Class Home
  Inherits Page
  Private fuenteDeDatos As PagedCollectionView = Nothing

  Public Sub New()
    InitializeComponent()
    fuenteDeDatos = New PagedCollectionView(
      TryCast(Application.Current, App).ColTelefonos)
    LayoutRoot.DataContext = fuenteDeDatos
  End Sub
  ' ...
End Class
```

El código anterior obtiene la vista de colección *Telefonos* y la asigna al contexto de datos del **Grid**, para que pueda ser utilizada por todos sus elementos. Pruebe a ejecutar ahora la aplicación y observe los resultados.

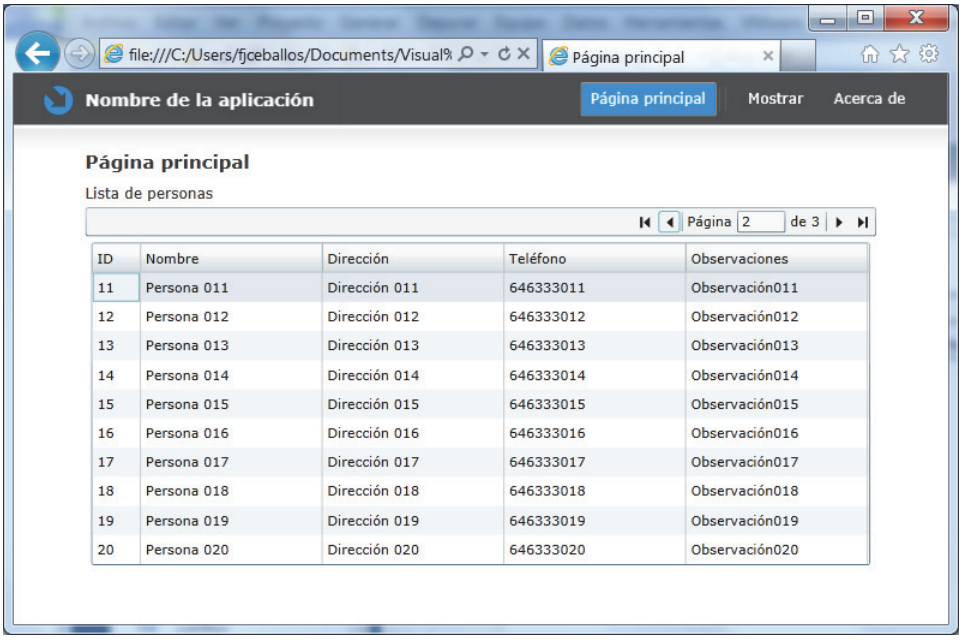
Paginación controlada

El control **DataPager** ofrece una interfaz de usuario adecuada para controlar la paginación con una vista **PagedCollectionView**. Su aspecto es el siguiente:



Los datos paginados se muestran por lo general en un control que representa una colección, como **DataGrid** o **ListBox**. Para ello, se asigna la colección de datos, generalmente una vista de tipo **PagedCollectionView**, a la propiedad **ItemsSource** del control que se desea paginar y a la propiedad **Source** del **DataPager**. El número de elementos que mostrará este control en una página viene fijado por la propiedad **PageSize** del mismo.

Como ejemplo, añada a la página *Home* anterior un control **DataPager** que muestre páginas de diez elementos, según muestra la figura siguiente:



ID	Nombre	Dirección	Teléfono	Observaciones
11	Persona 011	Dirección 011	646333011	Observación011
12	Persona 012	Dirección 012	646333012	Observación012
13	Persona 013	Dirección 013	646333013	Observación013
14	Persona 014	Dirección 014	646333014	Observación014
15	Persona 015	Dirección 015	646333015	Observación015
16	Persona 016	Dirección 016	646333016	Observación016
17	Persona 017	Dirección 017	646333017	Observación017
18	Persona 018	Dirección 018	646333018	Observación018
19	Persona 019	Dirección 019	646333019	Observación019
20	Persona 020	Dirección 020	646333020	Observación020

Para realizar este diseño, abra el fichero *Home.xaml* y añada el control **DataPager**:

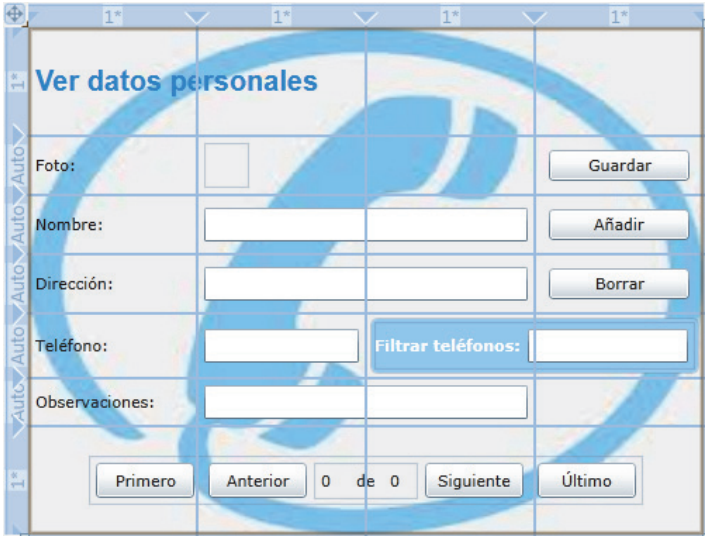
```
<Grid x:Name="LayoutRoot">
  <StackPanel x:Name="ContentStackPanel">
    <TextBlock x:Name="HeaderText" .../>
    <TextBlock x:Name="ContentText" .../>
    <sdk:DataPager Name="dpPaginador" PageSize="10" />
    <sdk:DataGrid Name="dgPersonas" Margin="5"
    ...
</Grid>
```

Y a continuación, en el fichero de código subyacente asigne a la propiedad **Source** del **DataPager** la misma colección de datos que asignó al **DataGrid**:

```
Public Sub New()
  InitializeComponent()
  fuenteDeDatos = New PagedCollectionView(
    TryCast(Application.Current, App).ColTelefonos)
  dpPaginador.Source = fuenteDeDatos
  LayoutRoot.DataContext = fuenteDeDatos
End Sub
```

Paginación personalizada

Ahora, añadiremos otra página con una interfaz que permita, por una parte, mostrar uno a uno los objetos que forman parte de la colección *Telefonos* y, por otra, otras operaciones como añadir o borrar un registro (un objeto), o filtrar los registros. Para ello, añada a la carpeta *Views* del proyecto otra página denominada *PagMostrarPersona* que, una vez diseñada, tendrá el aspecto siguiente:



El siguiente paso es realizar un diseño de *PagMostrarPersona* para que muestre los datos de la persona a la cual navegue el usuario por medio de los botones de avanzar y retroceder que colocaremos en esta página.

No abundaremos en más explicaciones sobre este diseño, puesto que es similar al realizado anteriormente en el apartado *Acceso a una base de datos desde una XBAP* del capítulo *Navegación de tipo web*.

Para que esta página pueda ser mostrada desde la barra de navegación de la aplicación, añada a la página *MainPage.xaml* el control **HyperlinkButton** correspondiente, según explicamos anteriormente en el apartado *Navegación*:

```
<Border x:Name="LinksBorder"
        Style="{StaticResource LinksBorderStyle}">
  <StackPanel x:Name="LinksStackPanel"
              Style="{StaticResource LinksStackPanelStyle}">
    ...

    <Rectangle x:Name="Divider2"
               Style="{StaticResource DividerStyle}"/>
    <HyperlinkButton x:Name="Link3"
                     Style="{StaticResource LinkStyle}"
                     NavigateUri="/PagMostrarPersona"
                     TargetName="ContentFrame" Content="Mostrar"/>
    ...
  </StackPanel>
</Border>
```

Ya tenemos generado el modelo de objetos, representación de la base de datos, esto es, la capa de acceso a datos (DAL), y diseñada la interfaz gráfica de la página *PagMostrarPersona*. Ahora, partiendo de la colección de datos proporcionada por este modelo, crearemos un objeto **PagedCollectionView** para utilizarlo como origen de datos en esta página. Para ello, añada a la clase que da lugar a esta página el código indicado a continuación:

```
Partial Public Class PagMostrarPersona
  Inherits Page
  Private fuenteDeDatos As PagedCollectionView = Nothing

  Public Sub New()
    InitializeComponent()
    fuenteDeDatos = New PagedCollectionView(
      TryCast(Application.Current, App).ColTelefonos)
    LayoutRoot.DataContext = fuenteDeDatos
  End Sub
  ' ...
End Class
```

Un breve vistazo a la propiedad *ColTelefonos* de la clase *App* nos recuerda que dicha propiedad proporciona la colección de objetos *Telefono* obtenida de nuestra supuesta base de datos. Observe que la vista *fuenteDeDatos* que hemos obtenido de esta colección la hemos asignado al contexto de datos del **Grid** para que pueda ser utilizada por todos los elementos de la interfaz gráfica.

Para que los controles de la página *pagMostrarPersona* muestren los datos referenciados por las propiedades *Nombre*, *Direccion*, *NumTelefono*, *Observaciones* y *Foto* de cada uno de los objetos *Telefono* de la colección subyacente de *fuenteDeDatos*, hemos enlazado cada una de esas propiedades con la propiedad adecuada de sus respectivos controles en la página, y el modo de enlace lo hemos establecido en las dos direcciones:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1"
       Style="{StaticResource EstiloImagen}"
       Source="{Binding Path=Foto, Mode=TwoWay}"/>
<TextBox Name="ctNombre" Grid.Column="1" Grid.Row="2"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=Nombre, Mode=TwoWay}" />
<TextBox Name="ctDireccion" Grid.Column="1" Grid.Row="3"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=Direccion, Mode=TwoWay}"/>
<TextBox Name="ctTelefono" Grid.Column="1" Grid.Row="4"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=NumTelefono, Mode=TwoWay}"/>
<TextBox Name="ctObservaciones" Grid.Column="1" Grid.Row="5"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=Observaciones, Mode=TwoWay}"/>
```

Si ejecuta ahora la aplicación, observará cómo la ventana muestra ya los datos del primer objeto de la colección.

El siguiente paso es programar los controladores del evento **Click** de los botones "Primero", "Anterior", "Siguiente" y "Último" para que muestren el objeto de la colección acorde a lo especificado por su título. Estas operaciones son proporcionadas por la vista *fuenteDeDatos* de la colección de objetos. Entonces, lo que vamos a hacer a continuación es añadir esos controladores:

```
<StackPanel Grid.ColumnSpan="4" Grid.Row="6"
            Orientation="Horizontal"
            Style="{StaticResource EstiloStackPanel}">
  <Button Name="btPrimero" Content="Primero"
          Style="{StaticResource EstiloButton}"
          Click="btPrimero_Click" />
  <Button Name="btAnterior" Content="Anterior"
          Style="{StaticResource EstiloButton}"
          Click="btAnterior_Click" />
```

```
  <StackPanel VerticalAlignment="Center" Orientation="Horizontal">
    <TextBlock Name="btxPosicionReg" Text="0"
               Margin="5" Width="15"/>
    <TextBlock Text="de" Margin="5" />
    <TextBlock Name="btxTotalReg" Text="0" Margin="5" Width="15"/>
  </StackPanel>
  <Button Name="btSiguiente" Content="Siguiente"
          Style="{StaticResource EstiloButton}"
          Click="btSiguiente_Click" />
  <Button Name="btUltimo" Content="Último"
          Style="{StaticResource EstiloButton}"
          Click="btUltimo_Click" />
</StackPanel>
```

Después, haciendo uso de los métodos **MoveCurrentToFirst**, **MoveCurrentToPrevious**, **MoveCurrentToNext** y **MoveCurrentToLast** de la clase **PagedCollectionView**, que permiten posicionarse en el primero, anterior, siguiente o último registro, respectivamente, edite cada uno de los controladores como se indica a continuación:

```
Private Sub btPrimero_Click(sender As Object, e As RoutedEventArgs)
  fuenteDeDatos.MoveCurrentToFirst()
End Sub

Private Sub btAnterior_Click(sender As Object, e As RoutedEventArgs)
  fuenteDeDatos.MoveCurrentToPrevious()
  If fuenteDeDatos.IsCurrentBeforeFirst Then
    fuenteDeDatos.MoveCurrentToFirst()
  End If
End Sub

Private Sub btSiguiente_Click(sender As Object, e As RoutedEventArgs)
  fuenteDeDatos.MoveCurrentToNext()
  If fuenteDeDatos.IsCurrentAfterLast Then
    fuenteDeDatos.MoveCurrentToLast()
  End If
End Sub

Private Sub btUltimo_Click(sender As Object, e As RoutedEventArgs)
  fuenteDeDatos.MoveCurrentToLast()
End Sub
```

Si ahora ejecuta la aplicación, observará cómo la ventana permite ya mostrar, uno a uno, los datos de todos los objetos de la colección.

El siguiente paso es hacer que los controles de texto que están entre los botones “Anterior” y “Siguiente” indiquen la posición del registro actual y el número total de registros. Esta operación hay que realizarla cuando se navega a esta pági-

na y cada vez que cambia el elemento actual de la vista, que está referenciado por la propiedad **CurrentItem**. Para ello, debemos saber que cada vez que cambia el elemento actual de la vista se genera el evento **CurrentChanged**. Según esto, vamos a añadir el controlador *fuenteDeDatos_CurrentChanged* para manejar este evento. Lo haremos cuando se cargue la página *PagMostrarPersona*:

```
Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
  ' Manejar el evento CurrentChanged de la vista
  AddHandler fuenteDeDatos.CurrentChanged,
    AddressOf fuenteDeDatos_CurrentChanged
  ' Actualizar posición actual y total elementos
  ActualizarPosicionReg()
End Sub
```

El método **OnNavigatedTo** se ejecuta cuando el usuario navega a esta página. Su función, en esta página, es establecer el controlador del evento **CurrentChanged** y establecer los valores iniciales, posición del registro mostrado actualmente y número total de registros de la colección, mostrados por los controles *btxPosicionReg* y *btxTotalReg*, respectivamente.

A continuación, edite el método *fuenteDeDatos_CurrentChanged* para que llame al método *ActualizarPosicionReg* y actualice los valores mostrados por los controles *btxPosicionReg* y *btxTotalReg*:

```
Public Sub fuenteDeDatos_CurrentChanged(sender As Object,
e As EventArgs)
  ActualizarPosicionReg()
End Sub
```

El valor mostrado por el control *btxPosicionReg* se corresponde con el valor de la propiedad **CurrentPosition** de la vista (la primera posición es la 0) y el valor mostrado por el control *btxTotalReg*, con el valor de la propiedad **Count**. Según esto, el método *ActualizarPosicionReg* lo escribiremos así:

```
Private Sub ActualizarPosicionReg()
  If fuenteDeDatos.CurrentItem Is Nothing Then Return
  ' Actualizar posición actual y total registros
  Dim posicionActual As String =
    Convert.ToString(fuenteDeDatos.CurrentPosition + 1)
  btxPosicionReg.Text = posicionActual
  btxTotalReg.Text = fuenteDeDatos.Count.ToString()
End Sub
```

Filtrar los registros de la colección

Observando la interfaz gráfica de la página *PagMostrarPersona*, observamos los siguientes controles: un **Border** azul, un **Label** ("Filtrar teléfonos") y un **TextBox**, *ctFiltrar*, en el que el usuario escribirá el patrón, los dígitos, que deben contener los teléfonos (campo *NumTelefono*) que deseamos filtrar.

```
<Border Grid.Column="2" Grid.ColumnSpan="2" Grid.Row="4"
        Style="{StaticResource EstiloBorder}" >
  <StackPanel Orientation="Horizontal" >
    <sdk:Label Content="Filtrar teléfonos:"
               FontWeight="Bold" Foreground="White" />
    <TextBox x:Name="ctFiltrar"
             Style="{StaticResource EstiloTextBox}"
             TextChanged="ctFiltrar_TextChanged" MinWidth="114" />
  </StackPanel>
</Border>
```

Partiendo de este diseño, ¿qué código y dónde tenemos que escribirlo para filtrar los elementos a visualizar? Sabemos que una vista tiene la capacidad de filtrar los elementos provenientes de la colección subyacente por medio de su propiedad **Filter**. ¿Dónde establecemos este filtro? Pues en el controlador del evento **TextChanged** del control *ctFiltrar*; así podremos capturar el texto introducido cada vez que cambie el contenido de la caja de texto.

Agregue este controlador si aún no lo ha añadido, muestre el fichero de código subyacente de *PagMostrarPersona* y edítelo como se indica a continuación:

```
Private Sub ctFiltrar_TextChanged(sender As Object,
e As TextChangedEventArgs)
  Dim ct As TextBox = TryCast(sender, TextBox)
  If ct Is Nothing Then Return
  If fuenteDeDatos.CanFilter Then
    fuenteDeDatos.Filter = Function(elemento As Object) _
      DirectCast(elemento, Telefono).NumTelefono.Contains(ct.Text)
  End If
End Sub
```

Para especificar los criterios que deben cumplir los objetos que pertenecerán a la colección filtrada, hay que vincular la propiedad **Filter** de la vista con el método que se utilice para especificar tales criterios. Por eso, esta propiedad es un predicado (un delegado) cuya firma es *Boolean Predicate(T obj)*, lo que significa que le tendremos que asignar un método anónimo que determine qué elementos de la colección cumplen el criterio de que la subcadena *ct.Text* esté contenida en *NumTelefono*; todos los elementos que cumplan este criterio pertenecerán a la colección de objetos filtrada.

Cuando se aplica el filtro, el contenido de la vista cambia y esto hace que se produzca el evento **CollectionChanged**. Pues bien, utilizaremos el controlador de este evento para actualizar el control *btxTotalReg* que muestra el número total de elementos que en este instante tiene la vista, porque, de no hacerlo así, este control no se actualizaría hasta que el elemento actualmente mostrado cambie, que es cuando se genera el evento **CurrentChanged**. Según esto, análogamente a como procedimos anteriormente, añadimos el controlador de este evento:

```
Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
  ' Manejar el evento CurrentChanged de la vista
  AddHandler fuenteDeDatos.CurrentChanged,
    AddressOf fuenteDeDatos_CurrentChanged
  ' Manejar el evento CollectionChanged de la vista
  AddHandler fuenteDeDatos.CollectionChanged,
    AddressOf fuenteDeDatos_CollectionChanged
  ' Actualizar posición actual y total elementos
  ActualizarPosicionReg()
End Sub
```

Finalmente, edite el método *fuenteDeDatos_CollectionChanged* así:

```
Public Sub fuenteDeDatos_CollectionChanged(sender As Object,
e As NotifyCollectionChangedEventArgs)
  ActualizarPosicionReg()
End Sub
```

Trabajar con imágenes

Cuando ejecute la aplicación, observará que puede modificar todos los campos, excepto la foto, y que las modificaciones persisten mientras la aplicación se está ejecutando (en un capítulo posterior, veremos cómo guardar los cambios en la base de datos de la cual se obtuvieron).

Probablemente, en esa supuesta base de datos haya registros con un valor **Nothing** para el campo foto. Lo que haremos en estos casos será: en el instante en el que se muestre el registro asignamos, al mismo tiempo que se actualiza la posición del registro, una imagen ficticia en el control destinado a mostrarla. Esa imagen la vamos a guardar como un recurso de la aplicación en la carpeta *Assets* (activos/recursos) del proyecto.

Una primera aproximación podría ser añadir al método *ActualizarPosicionReg* el siguiente código:

```
If TryCast(fuenteDeDatos.CurrentItem, Telefono).Foto IsNot Nothing Then
  Return
End If
```

```
Dim fotoFicticia As Bitmap = My.Resources.Resources.unafoto
Dim ms As New MemoryStream()
fotoFicticia.Save(ms, ImageFormat.Jpeg)
TryCast(vista.CurrentItem, Telefono).Foto = ms.ToArray()
```

Pero sucede que Silverlight no contempla la clase **Bitmap**, pero sí permite construir un objeto **BitmapImage** desde un recurso JPG o PNG (la clase **BitmapImage** se deriva de **BitmapSource** y ésta, a su vez, de **ImageSource**). Según esto, una segunda aproximación podría ser este otro código:

```
If TryCast(fuenteDeDatos.CurrentItem, Telefono).Foto IsNot Nothing Then
  Return
End If
Dim fotoFicticia As New BitmapImage(
  New Uri("/SilverlightAccesoColDatos;component/Assets/foto.jpg",
          UriKind.Relative))
imFoto.Source = fotoFicticia
```

Si ejecuta ahora la aplicación y muestra esta página, vemos que la foto ya aparece en cada uno de los registros, pero si la ejecuta en modo depuración y observa el panel de resultados, verá que cada vez que establecemos la imagen se envía el mensaje de error "No se puede convertir el tipo **BitmapImage** en el tipo **Byte()**". Esto ocurre cuando el motor de enlace, para el control **Image**, invocando al método **ConvertBack** del conversor predeterminado, trata de modificar la propiedad *Foto* de tipo **Byte()** del objeto *Telefono* con el que está enlazado, con los bytes correspondientes a la imagen establecida.

Lo anterior quiere decir que hay que proporcionar al enlace un conversor en ambos sentidos. Los conversores fueron estudiados en el capítulo *Introducción a WPF*. Recordando, de forma resumida, un conversor es una implementación de la interfaz **IValueConverter**. Esta interfaz aporta dos métodos: **Convert** y **ConvertBack**. El primero se invoca cuando la conversión ocurre en la dirección origen-destino y el segundo, cuando la conversión ocurre en la dirección contraria. Según esto, vamos a añadir a la carpeta *Assets* una clase *ConversorBytes_ImageSource* que implemente la interfaz **IValueConverter** y después, editamos los métodos citados, como se indica a continuación:

```
Public Class ConversorBytes_ImageSource
  Implements IValueConverter

  ' Byte() a ImageSource (origen --> destino)
  Private Function Convert(value As Object, targetType As Type,
  parameter As Object, culture As System.Globalization.CultureInfo) _
  As Object Implements IValueConverter.Convert
    If targetType.FullName = "System.Windows.Media.ImageSource" Then
      If value Is Nothing Then Return Nothing
```

```
      ' ...
    End If
    Return Nothing
  End Function

  ' ImageSource a Byte() (destino --> origen)
  Private Function ConvertBack(value As Object, targetType As Type,
  parameter As Object, culture As System.Globalization.CultureInfo) _
  As Object Implements IValueConverter.ConvertBack
    If targetType.FullName = "System.Byte()" Then
      If value Is Nothing Then Return Nothing

      ' ...
    End If
    Return Nothing
  End Function
End Class
```

El método **Convert** tiene que devolver la matriz de tipo **Byte()** referenciada por la propiedad *Foto* del objeto *Telefono* accedido convertida en la imagen que se asignará a la propiedad **Source** del control *imFoto*, y el método **ConvertBack**, la imagen referenciada por la propiedad **Source** del control *imFoto* accedido convertida en la matriz de tipo **Byte()** que será asignada a la propiedad *Foto* del objeto *Telefono* correspondiente.

Para poder utilizar este conversor en el enlace del control **Image**, tendremos que crear un objeto *ConversorBytes_ImageSource*, para lo cual definiremos un recurso que haga referencia al mismo, en la sección de recursos del **Grid**. Evidentemente, para utilizar esta clase necesitamos también especificar el espacio de nombres al que pertenece la misma. Todas estas operaciones puede verlas en el código siguiente:

```
<navigation:Page
   ...
   xmlns:local="clr-namespace:SilverlightAccesoColDatos"
   ...
   Title="Página mostrar persona">

  <Grid x:Name="LayoutRoot">
    <Grid.Resources>
      <local:ConversorBytes_ImageSource
                x:Key="conversorBytes_ImageSource"/>
    </Grid.Resources>
    ...
</navigation:Page>
```

Para utilizar este conversor, *conversorBytes_ImageSource*, en el enlace de la propiedad **Source** del control *imFoto* con la propiedad *Foto* de un objeto *Telefono*, simplemente tenemos que asignar el recurso a su propiedad **Converter**:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1"
        Style="{StaticResource EstiloImagen}"
        Source="{Binding Path=Foto, Mode=TwoWay,
        Converter={StaticResource conversorBytes_ImageSource}}"
        MouseLeftButtonDown="imFoto_MouseLeftButtonDown" />
```

Según lo expuesto, de este código se desprende que la propiedad **Source** de *imFoto* tomará el valor de la propiedad *Foto* del objeto *Telefono* enlazado convertido a un **BitmapSource** por medio del recurso *conversorBytes_ImageSource* y viceversa, la propiedad *Foto* del objeto *Telefono* tomará el valor de la propiedad **Source** de *imFoto* convertido a un objeto de tipo **Byte()**, dependiendo de que la propiedad que cambie sea **Source** o *Foto*.

Una vez añadida esta clase, intentamos completar el método **ConvertBack** para que realice la conversión requerida, pero la funcionalidad que proporciona la clase **BitmapImage** no permite obtener la matriz de bytes (*array*). Como alternativa, Silverlight proporciona una clase denominada **WriteableBitmap**, derivada también de **BitmapSource**, que proporciona una imagen que se puede modificar.

- System.Windows.Media.ImageSource
  - System.Windows.Media.Imaging.BitmapSource
    - System.Windows.Media.Imaging.BitmapImage
    - System.Windows.Media.Imaging.WriteableBitmap

Se puede construir un objeto **WriteableBitmap** a partir de un **BitmapSource** o a partir de las dimensiones del mapa de bits que se desea construir y, también, una vez construido el objeto, se puede obtener la matriz de enteros que representa la textura 2D del mapa de bits a partir de su propiedad **Pixels**. Esto soluciona casi el problema, pero necesitamos una matriz de tipo **Byte()**. Utilizando algunas de las clases básicas (MemoryStream, StreamWriter, BinaryWriter, BitConverter, etc.) podríamos implementar dos métodos que convirtieran del tipo **Integer()** a **Byte()** y viceversa, pero ya hay una clase **WriteableBitmap** extendida, que se puede descargar de la dirección *http://writeablebitmapex.codeplex.com/*, que proporciona una funcionalidad añadida que da solución al problema planteado. Optamos entonces por descargar esta biblioteca y copiarla en la carpeta *Assets* del proyecto. Después añadimos al proyecto una referencia a la misma.

Por ejemplo, esta clase proporciona los métodos **ToByteArray** y **FromByteArray**. **ToByteArray** permite obtener la matriz de bytes de la imagen:

```
Dim wbm As WriteableBitmap = TryCast(value, WriteableBitmap)
```

```
Dim bytesImagen As Byte() = wbm.ToByteArray()
```

y **FromByteArray** permite obtener la imagen desde la matriz de bytes, pero requiere las dimensiones del mapa de bits:

```
Dim bytesImagen As Byte() = DirectCast(value, Byte())
' ...
Return New WriteableBitmap(width, height).FromByteArray(bytesImagen)
```

Según lo expuesto, la propiedad *Foto* del objeto *Telefono* deberá almacenar no sólo el mapa de bits, sino también sus dimensiones. Según esto, completamos los métodos **Convert** y **ConvertBack** así:

```
Private Function Convert(value As Object, targetType As Type,
parameter As Object, culture As System.Globalization.CultureInfo) _
As Object Implements IValueConverter.Convert
  If targetType.FullName = "System.Windows.Media.ImageSource" Then
    If value Is Nothing Then Return Nothing

    ' Obtener las dimensiones y la imagen
    Dim bytesWHImagen As Byte() = DirectCast(value, Byte())
    Dim w As Byte() = BitConverter.GetBytes(0)
    Dim h As Byte() = BitConverter.GetBytes(0)
    Array.Copy(bytesWHImagen, 0, w, 0, 4)
    Array.Copy(bytesWHImagen, 4, h, 0, 4)
    Dim width As Integer = BitConverter.ToInt32(w, 0)
    Dim height As Integer = BitConverter.ToInt32(h, 0)

    Dim bytesImagen As Byte() = New Byte(bytesWHImagen.Length - 9) {}
    Array.Copy(bytesWHImagen, 8,
               bytesImagen, 0, bytesWHImagen.Length - 8)

    Return New WriteableBitmap(
                width, height).FromByteArray(bytesImagen)
  End If
  Return Nothing
End Function

' ImageSource a Byte() (destino --> origen)
Private Function ConvertBack(value As Object, targetType As Type,
parameter As Object, culture As System.Globalization.CultureInfo) _
As Object Implements IValueConverter.ConvertBack
  If targetType.FullName = "System.Byte[]" Then
    If value Is Nothing Then Return Nothing

    Dim wbm As WriteableBitmap = TryCast(value, WriteableBitmap)
    ' Añadir las dimensiones antes de la imagen
    Dim bytesImagen As Byte() = wbm.ToByteArray()
    Dim bytesWHImagen As Byte() = New Byte(bytesImagen.Length + 7) {}
```

```
    Dim w As Byte() = BitConverter.GetBytes(wbm.PixelWidth)
    Dim h As Byte() = BitConverter.GetBytes(wbm.PixelHeight)
    Array.Copy(w, 0, bytesWHImagen, 0, 4)
    Array.Copy(h, 0, bytesWHImagen, 4, 4)
    bytesImagen.CopyTo(bytesWHImagen, 8)

    If wbm IsNot Nothing Then Return bytesWHImagen
  End If
  Return Nothing
End Function
```

Observe que, ahora, el método **ConvertBack** requiere que *value* referencie a un objeto **WriteableBitmap** (para invocar a **ToByteArray**) y que un objeto **BitmapImage** no puede convertirse en un objeto **WriteableBitmap**. Esto último implica modificar el método *ActualizarPosicionReg* para que proporcione la imagen a partir de un objeto **WriteableBitmap**:

```
Private Sub ActualizarPosicionReg()
  If fuenteDeDatos.CurrentItem Is Nothing Then Return
  ' Actualizar posición actual y total registros
  Dim posicionActual As String =
    Convert.ToString(fuenteDeDatos.CurrentPosition + 1)
  btxPosicionReg.Text = posicionActual
  btxTotalReg.Text = fuenteDeDatos.Count.ToString()

  ' Poner una foto ficticia si este campo es Nothing
  If TryCast(fuenteDeDatos.CurrentItem,
    Telefono).Foto IsNot Nothing Then Return
  Dim wbm As WriteableBitmap =
    New WriteableBitmap(0, 0).FromResource("Assets/foto.jpg")
  imFoto.Source = wbm
End Sub
```

El método **FromResource** de **WriteableBitmap** permite cargar la imagen desde un recurso de la aplicación remplazando las dimensiones actuales, **PixelWidth** y **PixelHeight**, del objeto.

Cargar una nueva imagen

Para modificar la imagen, tendremos que implementar un código que permita obtener la imagen de nuestro sistema de ficheros y asignarla al control *imFoto* del registro actual.

Al igual que para el resto de los campos, para cambiar la foto del registro que se está mostrando, haremos clic sobre ese control. Esta acción hará que se genere, entre otros, el evento **MouseLeftButtonDown**. Entonces, vamos a añadir el controlador para este evento:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1"
        Style="{StaticResource EstiloImagen}"
        Source="{Binding Path=Foto, Mode=TwoWay}"
        MouseLeftButtonDown="imFoto_MouseLeftButtonDown" />
```

La respuesta a este evento será mostrar la caja de diálogo *Abrir* para que el usuario pueda seleccionar la imagen, para lo cual, editamos el método *imFoto_MouseDown* según se muestra a continuación:

```
Private Sub imFoto_MouseLeftButtonDown(sender As Object, e As Sys-
tem.Windows.Input.MouseButtonEventArgs)
  Dim dlg As New OpenFileDialog()
  dlg.Filter = "ficheros jpg (*.jpg)|*.jpg|" &
               "ficheros jpeg (*.jpeg)|*.jpeg|" &
               "ficheros png (*.png)|*.png|" &
               "Todos (*.*)|*.*"

  If dlg.ShowDialog() = True Then
    If dlg.File.Name Is Nothing OrElse dlg.File.Name.Length = 0 Then
      Return
    End If
    Dim imagen As New WriteableBitmap(0, 0)
    imagen.SetSource(dlg.File.OpenRead())
    imFoto.Source = imagen
  End If
End Sub
```

Obsérvese que este método primero construye el diálogo de tipo *OpenFileDialog*, después lo configura y lo muestra. El usuario selecciona una imagen en su sistema de ficheros, se obtiene el flujo de bytes (*dlg.File.OpenRead*) del fichero correspondiente y se almacena en un objeto **WriteableBitmap** invocando a su método **SetSource**, objeto imagen que se asigna a la propiedad **Source** del control *imFoto*, quedando así la foto cambiada. El método **SetSource** remplaza las dimensiones actuales, **PixelWidth** y **PixelHeight**, del objeto **WriteableBitmap**.

La clase **OpenFileDialog** de Silverlight permite a un usuario mostrar un diálogo para abrir un fichero en un equipo local o en un equipo en red. Para mostrar el diálogo hay que llamar al método **ShowDialog**. Por motivos de seguridad, este tipo de diálogos los debe iniciar el usuario. La propiedad **File** de este diálogo obtiene un objeto **FileInfo** asociado con el fichero seleccionado que proporciona métodos para crear, copiar, eliminar, mover y abrir ficheros; por ejemplo, el método **OpenRead** de **FileInfo**, cuando es llamado desde una aplicación de confianza, crea un objeto **FileStream** de sólo lectura.

Sólo queda guardar los cambios en la base de datos, cuestión de la que hablaremos a continuación. Después veremos cómo añadir y borrar un registro. Para realizar todas estas operaciones, tendremos que implementar los controladores de

los botones "Guardar", "Añadir" y "Borrar" de la interfaz gráfica de la página *PagMostrarPersona* que diseñamos anteriormente.

Guardar los cambios realizados

Las modificaciones realizadas sobre la colección de objetos resultante de la consulta sobre la supuesta base de datos no tendrán efecto si no se envían a la base de datos, según estudiamos en el apartado *Realizar cambios en los datos* del capítulo *LINQ*. En ese caso, tuvimos que obtener una referencia al contexto de objetos utilizado para hacer la consulta, con el fin de invocar a su método **SaveChanges** y enviar los cambios a la base de datos. En Silverlight, lo lógico es que la colección de objetos acceda a los datos de la base de datos a través de servicios webs, por eso, posponemos este proceso a un capítulo posterior.

Añadir un nuevo registro

Según expusimos en el capítulo *Navegación de tipo web*, añadir un nuevo registro supone solicitar los datos correspondientes al usuario y añadir el objeto correspondiente a la colección actual que subyace bajo la propiedad *ColTelefonos* definida en la clase *App*. Allí vimos que esa operación puede realizarse añadiendo, desde el controlador del botón "Aceptar", un registro con unos valores ficticios al final de la colección y modificando después ese registro, o mostrando una nueva página para la captura de los datos y creando un nuevo objeto a partir de los mismos, para añadirlo después a la colección. La primera forma es muy sencilla, así que vamos a implementar la segunda utilizando una ventana **ChildWindow**.

Continuando con la aplicación, vamos a agregar una ventana **ChildWindow** en la carpeta *Views* del proyecto con el fin de que devuelva el nuevo objeto *Telefono* que deseamos añadir a la colección.

Según esto, agregue una nueva ventana secundaria de tipo **ChildWindow** que solicite al usuario los datos de una persona. Para ello, haga clic con el botón secundario del ratón sobre la carpeta *Views* del proyecto y seleccione *Agregar > Nuevo elemento > Ventana secundaria de Silverlight*. Denomine a la ventana *VenSecAgregarTfno*, por ejemplo, y haga clic en el botón *Agregar*.

El siguiente paso es realizar el diseño de la ventana *VenSecAgregarTfno* para que permita introducir los datos para una nueva persona.

Evidentemente, al igual que en la página *PagMostrarPersona*, los controles de la interfaz gráfica de esta ventana estarán vinculados con las propiedades correspondientes del nuevo objeto de la clase *Telefono* que deseamos añadir a la colección, con lo que, pensando en el control imagen, tendremos que utilizar el

conversor diseñado anteriormente y los enlaces serán en ambas direcciones entre el destino y el origen de los datos. En fin, es un diseño muy parecido al de *PagMostrarPersona*, por lo que no abundaremos en más detalles. Su aspecto será el mostrado en la figura siguiente.



Observe que la finalidad de este formulario es solicitar al usuario los valores de las propiedades *Nombre*, *Direccion*, *NumTelefono*, *Observaciones* y *Foto* de un nuevo objeto de la clase *Telefono* que deseamos añadir a la colección. Este objeto estará referenciado por una propiedad, que vamos a denominar *NuevoObjTfno*, que definiremos en la clase de la ventana y que asignaremos al contexto de datos de la misma. También, iniciaremos dicho objeto con unos valores ficticios que serán mostrados como datos iniciales una vez cargada la ventana, según muestra la figura anterior, iniciación que realizaremos en el controlador del evento **Loaded** de la ventana. El identificador de la persona, pasado como argumento en el constructor *Telefono*, será el valor ordinal siguiente al del último registro de la colección. Lo expuesto implica escribir el código siguiente:

```
Imports System.Windows.Media.Imaging
Imports ColeccionDeDatos

Partial Public Class VenSecAgregarTfno
  Inherits ChildWindow
  Private m_NuevoObjTfno As Telefono
  Public Property NuevoObjTfno() As Telefono
    Get
      Return m_NuevoObjTfno
    End Get
    Private Set(value As Telefono)
      m_NuevoObjTfno = value
    End Set
  End Property
```

```
  Public Sub New()
    InitializeComponent()
    Dim id As Integer =
      TryCast(Application.Current, App).ColTelefonos.Count + 1
    NuevoObjTfno = New Telefono(id)
    Me.DataContext = NuevoObjTfno
  End Sub

  Private Sub ChildWindow_Loaded(sender As Object, e As RoutedEventArgs)
    NuevoObjTfno.Nombre = "Nombre"
    NuevoObjTfno.Direccion = "Dirección"
    NuevoObjTfno.NumTelefono = "000000000"
    NuevoObjTfno.Observaciones = "Observaciones"
    Dim wbm As WriteableBitmap =
      New WriteableBitmap(0, 0).FromResource("Assets/foto.jpg")
    imFoto.Source = wbm
  End Sub

  Private Sub OKButton_Click(sender As Object, e As RoutedEventArgs)
    Me.DialogResult = True
  End Sub

  Private Sub CancelButton_Click(sender As Object, e As RoutedEventArgs)
    Me.DialogResult = False
  End Sub

  ' ...
End Class
```

El siguiente paso es enlazar, en el fichero de código XAML, las propiedades del objeto *NuevoObjTfno* de la clase *Telefono* con los controles correspondientes de la página *VenSecAgregarTfno* según se indica a continuación:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="1"
       Style="{StaticResource EstiloImagen}"
       Source="{Binding Path=Foto, Mode=TwoWay,
       Converter={StaticResource conversorBytes_ImageSource}}"
       MouseLeftButtonDown="imFoto_MouseLeftButtonDown" />
<TextBox Name="ctID" Grid.Column="2" Grid.Row="1"
         Width="65" Margin="5"
         VerticalAlignment="Center" HorizontalAlignment="Right"
         Text="{Binding Path=IdPersona, Mode=TwoWay}"/>
<TextBox Name="ctNombre" Grid.Column="1" Grid.Row="2"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=Nombre, Mode=TwoWay}"/>
<TextBox Name="ctDireccion" Grid.Column="1" Grid.Row="3"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=Direccion, Mode=TwoWay}"/>
<TextBox Name="ctTelefono" Grid.Column="1" Grid.Row="4"
         Style="{StaticResource EstiloTextBox}"
```

```
          Text="{Binding Path=NumTelefono, Mode=TwoWay}"/>
<TextBox Name="ctObservaciones" Grid.Column="1" Grid.Row="5"
          Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
          Text="{Binding Path=Observaciones, Mode=TwoWay}"/>
```

Los datos solicitados al usuario a través del formulario son introducidos por el mismo en las respectivas cajas de texto, excepto la foto que, para cambiarla, habrá que hacer clic sobre ella y buscarla en el sistema de ficheros de la máquina donde se está ejecutando la aplicación. Esta última operación la realizaremos desde el controlador del evento **MouseLeftButtonDown** del control **Image** según puede verse en el código indicado a continuación:

```
Private Sub imFoto_MouseLeftButtonDown(sender As Object,
e As MouseButtonEventArgs)
  Dim dlg As New OpenFileDialog()
  dlg.Filter = "ficheros jpg (*.jpg)|*.jpg|" &
               "ficheros jpeg (*.jpeg)|*.jpeg|" &
               "ficheros png (*.png)|*.png|" &
               "Todos (*.*)|*.*"

  If dlg.ShowDialog() = True Then
    If dlg.File.Name Is Nothing OrElse dlg.File.Name.Length = 0 Then
      Return
    End If
    Dim imagen As New WriteableBitmap(0, 0)
    imagen.SetSource(dlg.File.OpenRead())
    imFoto.Source = imagen
  End If
End Sub
```

Cuando el usuario haga clic sobre el botón *Aceptar* o *Cancelar* de la ventana *VenSecAgregarTfno* se cerrará retornando el control a la ventana primaria. El valor **True** o **False** asignado a **DialogResult** por los controladores de estos botones permitirá saber qué botón se pulsó.

Esta ventana será mostrada desde el controlador del botón "Añadir" de la página *PagMostrarPersona*. Para ello, es necesario crear un objeto de la clase *VenSecAgregarTfno* e invocar a su método **Show** que no devuelve nada. Según esto, añada el controlador de ese botón y edítelo como se indica a continuación:

```
Partial Public Class PagMostrarPersona
  Inherits Page
  Private fuenteDeDatos As PagedCollectionView = Nothing
  Private ventanaHija As VenSecAgregarTfno = Nothing

  Public Sub New()
    ' ...
  End Sub
```

```
  ' ...
  Private Sub btAñadir_Click(sender As Object, e As RoutedEventArgs)
    ventanaHija = New VenSecAgregarTfno()
    ventanaHija.Show()
  End Sub

  ' ...
End Class
```

Para obtener los datos introducidos una vez cerrada la ventana hija después de que el usuario haya pulsado el botón *Aceptar*, es importante recordar que **Show** se ejecuta sin bloquear la ejecución del código de la página padre; esto es, si hubiera código después de la llamada a **Show**, éste seguiría ejecutándose. Entonces, para ejecutar la tarea de obtener los datos introducidos, tendremos que hacerlo desde el controlador del evento **Closed** de la ventana hija. Añadimos, entonces, el controlador para este evento y editamos el método que responde al mismo según se indica a continuación:

```
Private Sub btAñadir_Click(sender As Object, e As RoutedEventArgs)
  ventanaHija = New VenSecAgregarTfno()
  AddHandler ventanaHija.Closed, AddressOf ventanaHija_Closed
  ventanaHija.Show()
End Sub

Private Sub ventanaHija_Closed(sender As Object, e As EventArgs)
  If ventanaHija.DialogResult = True Then
    If ventanaHija.NuevoObjTfno IsNot Nothing Then
      TryCast(fuenteDeDatos.SourceCollection,
        List(Of Telefono)).Add(ventanaHija.NuevoObjTfno)
      fuenteDeDatos.MoveCurrentToLast()
      ' Añadir el objeto a la base de datos
    End If
  End If
End Sub
```

El método *ventanaHija_Closed* obtiene el nuevo objeto *Telefono* por medio de la propiedad *NuevoObjTfno* de la ventana hija y lo añade a la colección.

Pruebe ahora a ejecutar la aplicación y observará que todo funciona correctamente.

Borrar un registro

Según expusimos en el capítulo *Navegación de tipo web*, para borrar el elemento actualmente mostrado simplemente tendremos que eliminarlo de la colección sub-

yacente de la vista. Para ello, edite el controlador del evento **Click** del botón "Borrar" como se indica a continuación:

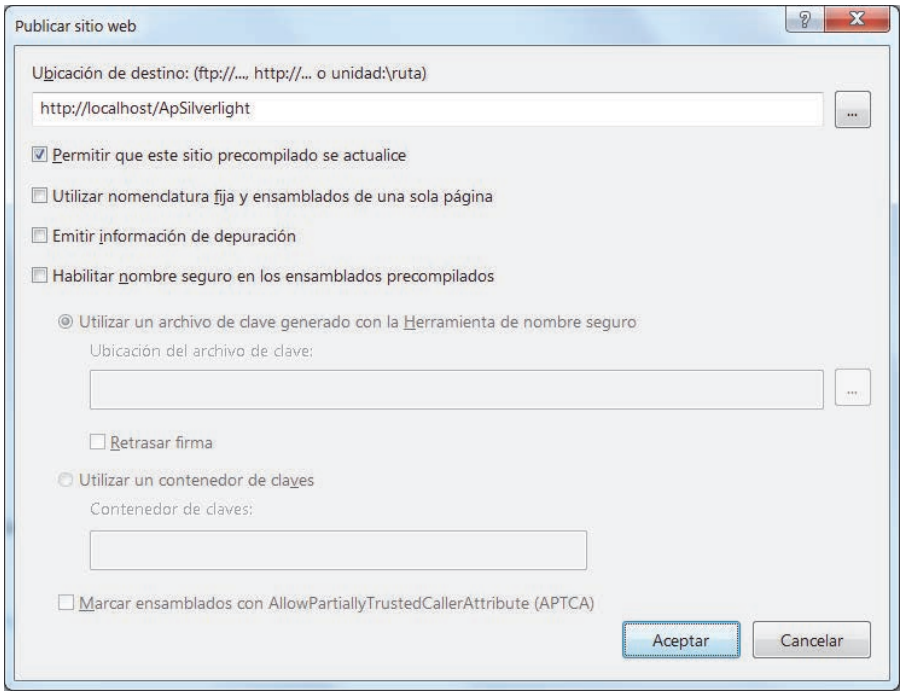
```
Private Sub btBorrar_Click(sender As Object, e As RoutedEventArgs)
  ' Quitar de la colección el elemento mostrado
  Dim tfno As Telefono = DirectCast(fuenteDeDatos.CurrentItem, Telefono)
  If tfno IsNot Nothing Then
    Dim siguientePosActual As Integer =
      (fuenteDeDatos.CurrentPosition - 1)
    TryCast(fuenteDeDatos.SourceCollection,
      List(Of Telefono)).Remove(tfno)
    fuenteDeDatos.Refresh()
    If siguientePosActual > -1 Then
      fuenteDeDatos.MoveCurrentToPosition(siguientePosActual)
      ' Borrar el objeto de la base de datos
    End If
  End If
End Sub
```

Este método quita de la colección el elemento actualmente mostrado y muestra el elemento anterior al eliminado. También lo tendría que eliminar de la supuesta base de datos.

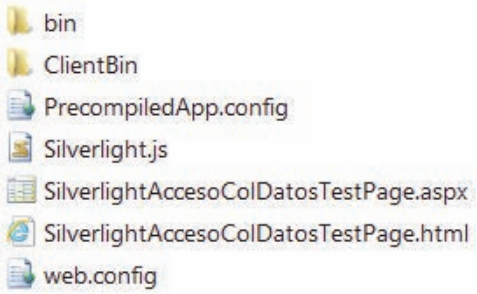
PUBLICAR LA APLICACIÓN

Publicar una aplicación Silverlight es parecido a publicar una aplicación web o una aplicación XBAP, según explicamos en el apartado *Publicar la aplicación* del capítulo *Navegación de tipo web* (eche una ojeada a este apartado). Según explicamos en ese apartado, para publicar una aplicación Silverlight, simplemente hay que mover el proyecto web al directorio *\Inetpub\wwwroot*.

Este proceso resulta muy sencillo si se utiliza el asistente para publicación de sitios webs que proporciona Visual Studio. Para lanzar este asistente, haga clic con el botón secundario del ratón sobre el nombre del proyecto web (no sobre el nombre del proyecto Silverlight) y elija la opción *Publicar sitio web* (tiene que haber iniciado Visual Studio como administrador). Después siga los pasos indicados por el asistente.



Cuando haya finalizado el trabajo con el asistente, podrá verificar que se ha creado una carpeta, en el ejemplo *ApSilverlight*, virtual vinculada con la carpeta física que contiene los ficheros que componen la aplicación Silverlight. El contenido de esta carpeta para nuestro ejemplo es el mostrado por la siguiente figura:



Para este ejemplo, la carpeta *bin* está vacía y la carpeta *ClienBin* contiene el fichero *.xap* que es, según explicamos en el apartado *Compilación de una aplicación Silverlight* al principio de este capítulo, el ensamblado correspondiente al proyecto Silverlight empaquetado. En cuanto al fichero *web.config*, especifica el atributo *targetFramework* de la etiqueta <*compilation*> para especificar cuál es la versión de .NET Framework de destino de un proyecto y generar únicamente código válido para esa versión:

```
<compilation debug="true" targetFramework="4.0"/>
```

Esto implica incluir a la aplicación dentro del grupo de aplicaciones ASP.NET v4.0, operación que haremos desde el administrador de IIS.

Realizar el proceso anterior manualmente es obvio. Cree en el sitio web la carpeta en la que desea copiar la aplicación, por ejemplo *\Inetpub\wwwroot\ApSilverlight*, y copie en la misma las carpetas y ficheros del sitio web del proyecto, según muestra la figura anterior.

Después, complete esta tarea especificando el nombre del directorio virtual. Para ello, abra el administrador de IIS, seleccione la carpeta *ApSilverlight* en el sitio web, haga clic sobre ella con el botón secundario del ratón, seleccione la opción *Convertir en aplicación* y acepte los valores predeterminados.

RESUMEN

En este capítulo hemos estudiado la tecnología *Silverlight* de Microsoft, la cual permite crear contenido interactivo que se ejecuta en el cliente, con soporte para gráficos dinámicos, contenido multimedia y animación, que va mucho más allá del HTML ordinario. Silverlight se implementa en los navegadores mediante un complemento ligero (un *plug-in*).

Empezamos el capítulo viendo la arquitectura de Silverlight y continuamos viendo cómo construir una aplicación, lo que nos permitió estudiar detalladamente la arquitectura de la misma, base para construir otras aplicaciones personalizadas. En este estudio vimos que para lanzar la ejecución de una aplicación Silverlight se necesita de una infraestructura que puede ser un fichero HTML, o una aplicación o sitio web ASP.NET.

Después, pensando en el diseño de una página Silverlight, estudiamos los controles Silverlight. En este estudio quedó claro que, en algunos casos, hay algunas diferencias con respecto a lo estudiado en WPF, en el sentido de que hay algunas restricciones en cuanto a los componentes que podemos utilizar y a las propiedades de esos componentes.

Respecto a los controles Silverlight, vimos los paneles de diseño del Silverlight, cómo redistribuir el espacio con el control GridSplitter, cómo manipular texto, imágenes, vimos los controles de contenido, cómo validar los datos, también vimos los controles de elementos, controles de rango definido, controles para gestionar fechas, degradados, ventanas y cajas de diálogo.

Dedicamos otro apartado al estudio de cómo Silverlight incorpora gráficos, animación y contenidos multimedia en una aplicación web.

Después estudiamos el modelo de navegación de Silverlight para saber cómo navegar entre las páginas que componen una aplicación.

Y finalizamos el capítulo con un estudio de cómo las aplicaciones Silverlight pueden obtener acceso a los datos de una base de datos. Aquí nos centramos en el acceso a datos por medio de un modelo de objetos, pensando en que tecnologías como Entity Framework, precisamente, lo que nos van a proporcionar son estos modelos. En este estudio desarrollamos una aplicación paso a paso viendo cómo acceder a los datos de una colección de objetos, cómo mostrar esa información paginada, cómo filtrar los datos, cómo incorporar imágenes a esos datos, así como añadir nuevos objetos o eliminarlos de la colección a través de la interfaz gráfica.

CAPÍTULO 10

© F.J.Ceballos/RA-MA

SERVICIOS WCF

¿Qué es WCF? WCF, acrónimo de *Windows Communication Foundation*, es un marco de trabajo unificado para hacer fácil la comunicación entre aplicaciones diversas en cualquier plataforma (.NET, J2EE, etc.) combinando en una sola tecnología lo que en versiones anteriores de Visual Studio existía en varias tecnologías: servicios webs ASMX, .NET Remoting (mover datos entre un cliente y un servidor), Enterprise Services (realizar comunicaciones por transacción) y Message Queue Server (comunicación mediante mensajes). De forma resumida, diríamos que los aspectos más importantes de WCF son:

- Unificación de las tecnologías de comunicación .NET Framework existentes.
- Compatibilidad para interoperabilidad entre proveedores, incluyendo confiabilidad, seguridad y transacciones.
- Orientación explícita al servicio.

Un servicio no es más que una aplicación que intercambia mensajes con otras aplicaciones, por lo tanto, es una unidad ideal para crear soluciones distribuidas. Piense en una solución distribuida usando programación orientada a objetos; como ya conocemos, esto nos obligaría a trabajar con tipos comunes perdiendo entonces la interoperabilidad porque no podríamos establecer una comunicación con otras plataformas diferentes que no reconocieran estos tipos de objetos. En cambio, en una solución orientada a servicios, como se intercambian mensajes, los dos extremos de la comunicación pueden perfectamente pertenecer a plataformas diferentes; es suficiente con que ambos extremos utilicen la misma política de comunicación.

Los servicios son autónomos y comparten esquemas (datos) y contratos (funcionalidad), no clases ni tipos en general y al intercambiar mensajes no tienen que asumir nada acerca de "qué es lo que hay al otro lado del extremo". En la vida ordinaria tenemos multitud de ejemplos; quién como cliente no ha llamado al "ser-

vicio de soporte técnico X" y, como resulta evidente, no hay una relación entre cliente y servidor, simplemente hay un intercambio de mensajes para comunicar la incidencia para la que deseamos nos den una solución. Evidentemente, de este ejemplo podemos deducir también que el servicio es autónomo; la infraestructura del servicio podrá cambiar, pero eso no tiene por qué afectar al cliente.

En resumen, los clientes consumen servicios y los servicios ofrecen soluciones a los clientes intercambiando mensajes, con lo que un servicio puede, a su vez, ser cliente de otro servicio. En WCF los mensajes tienen una cabecera y un cuerpo y son definidos en XML según el protocolo SOAP. A continuación se muestra cómo es el esqueleto de un mensaje SOAP:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/..."
 soap:encodingStyle="http://www.w3.org/...">

  <soap:Header>
    información específica de la aplicación, como la autenticación
  </soap:Header>

  <soap:Body>
    mensaje destinado al punto final de la comunicación
    <soap:Fault>
      para indicar mensajes de error
    </soap:Fault>
  </soap:Body>

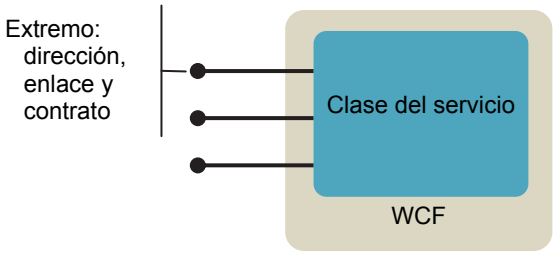
</soap:Envelope>
```

MODELO DE PROGRAMACIÓN DE WCF

WCF simplifica el desarrollo de aplicaciones distribuidas reuniendo la funcionalidad proporcionada por las tecnologías indicadas en el apartado anterior bajo un modelo de programación unificado fundamentado en la comunicación basada en mensajes entre dos entidades: un cliente WCF y un servicio WCF. El cliente es la aplicación que inicia la comunicación y el servicio es la aplicación que espera a que el cliente se comunique con él y responder así a su petición. Incluso, una única aplicación puede actuar como cliente y como servicio. Este modelo de programación orientado a servicios está definido por las clases agrupadas bajo el espacio de nombres **System.ServiceModel**.

Implementar un servicio WCF

Un servicio es una aplicación que expone uno o más extremos, donde cada uno de ellos expone una o más operaciones de servicio.



El *extremo* o punto final proporciona la única manera de comunicación con el servicio. Un *extremo* está compuesto de una *dirección*, un *enlace* y un *contrato*. La dirección define la ubicación del servicio; ésta podría ser una dirección URL, por ejemplo *http://servidor/miservicio*, una dirección FTP o una ruta de acceso local o de red. Un enlace define la manera de establecer la comunicación con el servicio, por ejemplo *BasicHttpBinding*, *PollingDuplexHttpBinding* o *WsHttpBinding*; los enlaces de WCF permiten especificar con facilidad un protocolo como HTTP o FTP, un mecanismo de seguridad como autenticación de Windows o nombres de usuario y contraseñas, y muchas otras características. Y un contrato incluye las operaciones expuestas por la clase de servicio WCF.

Un cliente Silverlight se comunica con un servicio utilizando el enlace *BasicHttpBinding*, que habilita SOAP 1.1 sobre HTTP (o HTTPS), configurado en el fichero *ServiceReferences.ClientConfig*, o el enlace *PollingDuplexHttpBinding*, que configura un extremo del servicio para la comunicación dúplex con un cliente (este tipo de enlace se debe configurar en el código, no en el fichero *ServiceReferences.ClientConfig* del proyecto).

Un servicio WCF puede exponer varios extremos, según indica la figura anterior, lo que permitirá a distintos clientes comunicarse con él de formas diferentes. Por ejemplo, un servicio de consulta de actas de notas en la universidad podría proporcionar un extremo para profesores y otro para los alumnos, cada uno mediante una dirección, enlace y/o contrato diferentes.

A continuación describiremos con un ejemplo cada una de las tareas necesarias para crear un servicio WCF y un cliente que llame al servicio. Un servicio WCF puede construirse como una *aplicación de servicios WCF* hospedada en IIS o como una *biblioteca de servicios WCF* e iniciarlo, al igual que otros servicios, en nuestra máquina, de forma manual o automática y como servicio local o de red.

Un servicio WCF, a diferencia de un servicio web XML (*.asmx*), sólo utiliza al servidor de aplicaciones, por ejemplo a IIS, como intermediario, pero se ejecuta siempre en el marco de ejecución de WCF, lo que le permite mantener su estado, mientras que los servicios webs XML no mantienen su estado y tienen un com-

portamiento y un tiempo de vida análogo al de las páginas webs (se crea un objeto de la clase del servicio, se ejecuta y se recicla).

Definir un contrato

Según lo expuesto anteriormente, la primera tarea para crear un servicio es definir el contrato. El contrato se corresponde con una *interfaz*, marcada con el atributo **ServiceContract**, que especifica las operaciones que ofrece el servicio. Cada una de estas operaciones será proporcionada por un método de la clase del servicio web marcado con el atributo **OperationContract**. Los métodos que no estén marcados con este atributo no serán expuestos a los clientes. Según esto, un contrato tendrá el aspecto siguiente:

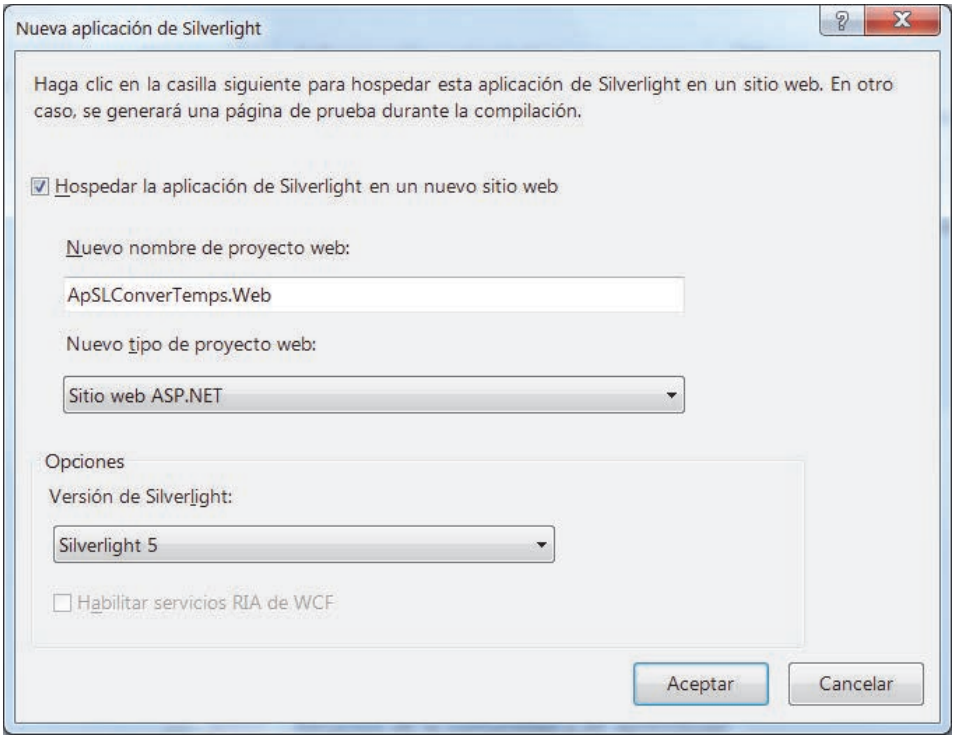
```
<ServiceContract()>
Public Interface IConvertirGrados
  ' Operaciones ofrecidas por el servicio
  <OperationContract()>
  Function ConvCentAFahr(gCent As Double) As Double

  <OperationContract()>
  Function ConvFahrACent(gFahr As Double) As Double
End Interface
```

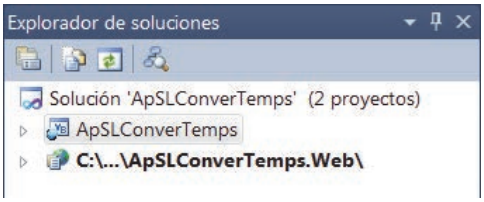
El atributo **ServiceContract** hace referencia a la clase **ServiceContractAttribute** para indicar que la interfaz *IConvertirGrados* define un contrato de servicio en una aplicación WCF y **OperationContract** hace referencia a la clase **OperationContractAttribute** para indicar que un método, como *ConvCentAFahr* o *ConvFahrACent*, define una operación que forma parte de un contrato. Utilizando esta funcionalidad, el entorno de ejecución de WCF traducirá el contrato de servicio en la estructura correspondiente en el CLR y viceversa, de forma que el desarrollador pueda abstraerse de la interoperabilidad y centrarse en la solución del problema.

Pasemos a la parte práctica. Como ejemplo vamos a crear un servicio WCF que muestre una interfaz para que una aplicación Silverlight cliente pueda solicitar convertir un valor en grados Fahrenheit a su correspondiente valor en grados centígrados y viceversa. En principio, construiremos la aplicación Silverlight y el servicio WCF sobre nuestro sistema de ficheros y después publicaremos ambos por medio de IIS. Entonces, abra Visual Studio como administrador y cree una nueva aplicación Silverlight según explicamos en el capítulo *Silverlight*: seleccione *Archivo > Nuevo > Proyecto > Visual Basic > Silverlight > Aplicación de Silverlight*. Denomine al proyecto *ApSLConverTemps*, por ejemplo. Seleccione también la casilla de verificación *Crear directorio para la solución* para agrupar en una misma carpeta los proyectos que se crearán. Después, haga clic en *Acep-*

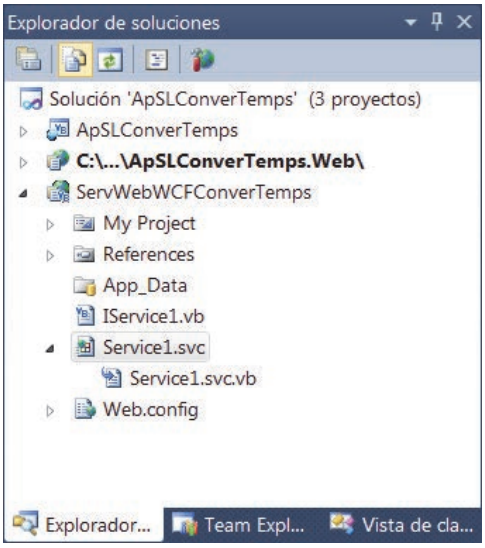
tar. Se visualizará un diálogo, que realizará otra serie de preguntas, según se muestra a continuación:



Hospedamos la aplicación Silverlight en un sitio web ASP.NET y hacemos clic en *Aceptar*. Se generará una solución con dos proyectos: *ApSLConverTemps*, proyecto Silverlight, y *ApSLConverTemps.Web*, proyecto web.



A continuación, creamos un servicio WCF. Para ello, añadimos a la solución anterior un nuevo proyecto de tipo *Aplicación de servicios WCF*: clic con el botón secundario del ratón sobre el nombre de la solución, *Agregar > Nuevo proyecto > WCF > Aplicación de servicios WCF*. Denomínelo *ServWebWCFConverTemps*. Después haga clic en *Aceptar*. La solución tiene ahora tres proyectos, según puede verse en la figura siguiente:



Observando el *Explorador de soluciones* vemos que se ha creado un nuevo proyecto *ServWebWCFConverTemps* que incluye el fichero *Service1.svc*, que actúa como punto de entrada al servicio WFC, que está vinculado con el fichero *Service1.svc.vb* que contiene la clase que construye el servicio (la lógica de negocio), la cual implementa la interfaz definida en *IService1.vb* que se corresponde con el contrato que define las operaciones que ofrece el servicio WFC.

Puede cambiar los nombres y adaptarlos al contexto de la aplicación. Por ejemplo, cambie el nombre del fichero *IService1.vb* por *IServWebWCFConverTemps.vb* y el nombre de la interfaz *IService1* por *IServWebWCFConverTemps*.

Cambie el nombre del fichero *Service1.svc* por *ServWebWCFConverTemps.svc* y el nombre de la clase *Service1* por *ServWebWCFConverTemps*. Este cambio exige hacerlo también en los atributos de la cláusula *ServiceHost* del fichero XML asociado *.svc*. Cambie también el nombre del fichero *Service1.svc.vb* por *ServWebWCFConverTemps.svc.vb*.

Finalmente, compile la solución y verifique que el servicio puede ser accedido utilizando el explorador: clic con el botón secundario del ratón sobre el nombre del fichero *ServWebWCFConverTemps.svc* > *Ver en el explorador*. Se mostrará una página en la que se puede ver el documento XML que describe el servicio en WSDL y la pauta a seguir para probar las operaciones del servicio; esto es, utilizando la herramienta *svcutil.exe* generaríamos el *proxy* (*ServWebWCFConverTemps.vb*) y el fichero de configuración (*output.config* > *app.config*) que incluiríamos en una aplicación de consola cliente del servicio desde la que podríamos llamar a las operaciones del servicio.



A continuación, vamos a definir el contrato del servicio (**ServiceContract**), esto es, las operaciones que ofrece el servicio. En nuestro caso, en principio, ofrece dos: la conversión de grados centígrados a Fahrenheit y viceversa. Aplique el atributo **OperationContract** a cada método que desea exponer públicamente como parte del contrato WCF.

```
<ServiceContract(Namespace:="http://websol.aut.uah.es/ServiciosWeb/")>
Public Interface IServWebWCFConverTemps
  ' Operaciones del servicio
  <OperationContract()>
  Function ConvCentAFahr(gCent As Double) As Double

  <OperationContract()>
  Function ConvFahrACent(gFahr As Double) As Double
End Interface
```

Observe la cláusula **Namespace** del atributo **ServiceContract**: especifica el espacio de nombres al que pertenecerá el servicio. Especificar el espacio de nombres explícitamente es un procedimiento recomendado porque distinguirá al servicio de otros con el mismo nombre, lo que podría ocurrir si utiliza, por omisión, el espacio de nombres predeterminado.

Además del contrato del servicio, puede utilizar un contrato de datos (**DataContract**), como se ilustra en el ejemplo siguiente, para agregar tipos compuestos

a las operaciones del servicio. Esto es, WCF utiliza un motor de forma predeterminada para seriar y deseriar los datos (convertirlos de y a XML) de tipo de contrato de datos. Todos los tipos primitivos de .NET Framework, como enteros y cadenas, así como ciertos tipos tratados como primitivos, como DateTime y XmlElement, y otros tipos .NET Framework son tipos de contrato de datos, por lo que se pueden seriar sin más. Esto es así porque la infraestructura WCF utiliza mensajes XML para el envío y recepción de datos entre el cliente y el servicio.

Entonces, los nuevos tipos complejos que se creen deben tener un contrato de datos definido para que sean seriables por el motor **DataContractSerializer**. Sólo pueden seriarse los atributos y propiedades de lectura y escritura públicos (no métodos) de estos tipos. La clase que defina ese contrato de datos irá marcada con el atributo **DataContract**, y las propiedades o campos, con el atributo **DataMember**, para que el motor **DataContractSerializer** pueda proceder a la seriación y deseriación.

Siguiendo con nuestro ejemplo, supongamos que queremos retornar un tipo más complejo que el simple valor en grados; por ejemplo, un objeto *Detalles* con tres atributos: información de si el valor convertido corresponde a grados centígrados o Fahrenheit, un literal que incluya el valor y la leyenda "grados centígrados/fahrenheit" y el valor en grados convertido. Según esto, el contrato de datos podría ser así:

```
<DataContract(Namespace:="http://websol.aut.uah.es/ServiciosWeb/")>
Public Class Detalles
  Private _sonCentigrados As Boolean
  Private _literal As String
  Private _grados As Double

  <DataMember()>
  Public Property SonCentigrados() As Boolean
    Get
      Return _sonCentigrados
    End Get
    Set(value As Boolean)
      _sonCentigrados = value
    End Set
  End Property

  <DataMember()>
  Public Property Literal() As String
    Get
      Return _literal
    End Get
    Set(value As String)
      _literal = value
    End Set
```

```
  End Property

  <DataMember()>
  Public Property Grados() As Double
    Get
      Return _grados
    End Get
    Set(value As Double)
      _grados = value
    End Set
  End Property
End Class
```

Ahora, apoyándonos en esta clase, añadimos un nuevo método, a la interfaz *IServWebWCFConverTemps*, que nos permita dar estos detalles:

```
<OperationContract()>
Function ResultadoDetallado(detalle As Detalles) As Detalles
```

Después de definir el contrato de servicio y sus operaciones, y de definir el contrato de datos (este último es opcional), definimos la clase del servicio. En nuestro caso se trata de la clase *ServWebWCFConverTemps* que implementa la interfaz *IServWebWCFConverTemps*:

```
Public Class ServWebWCFConverTemps
  Implements IServWebWCFConverTemps

  Public Function ConvCentAFahr(gCent As Double) As Double _
  Implements IServWebWCFConverTemps.ConvCentAFahr
    Dim grados As Double = Math.Round(9.0 / 5.0 * gCent + 32.0, 2)
    Return grados
  End Function

  Public Function ConvFahrACent(gFahr As Double) As Double _
  Implements IServWebWCFConverTemps.ConvFahrACent
    Dim grados As Double = Math.Round(((gFahr - 32.0) * 5.0) / 9.0, 2)
    Return grados
  End Function

  Public Function ResultadoDetallado(detalle As Detalles) As Detalles _
  Implements IServWebWCFConverTemps.ResultadoDetallado
    If detalle Is Nothing Then
      Throw New ArgumentNullException("detalle")
    End If
    If detalle.SonCentigrados Then
      detalle.Literal = ConvCentAFahr(detalle.Grados) &
                        " grados fahrenheit."
    Else
```

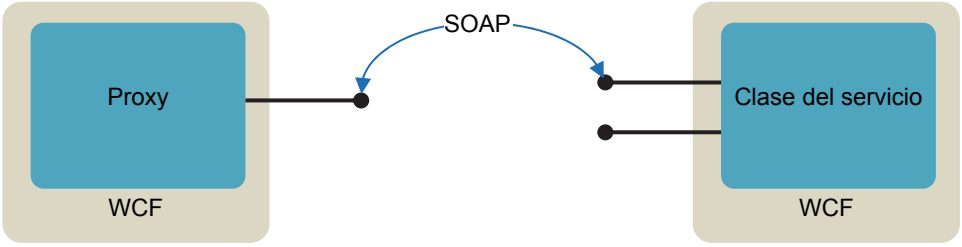
```
            detalle.Literal = ConvFahrACent(detalle.Grados) &
                              " grados centígrados."
        End If
        Return detalle
    End Function
End Class
```

Sepa también que una clase de servicio WCF puede implementar varios contratos de servicio.

El servicio WCF está finalizado. El siguiente paso es completar la aplicación Silverlight cliente para que consuma el servicio.

Implementar un cliente WCF

Un cliente WCF está compuesto de un *proxy* que habilita a una aplicación para poder establecer comunicación con un servicio WCF y un *extremo* que coincide con un extremo definido para el servicio. Para los servicios que exponen varios extremos, el cliente puede seleccionar el que mejor se ajuste a sus necesidades; por ejemplo, uno que permita establecer una comunicación a través de HTTP y sin autenticación.

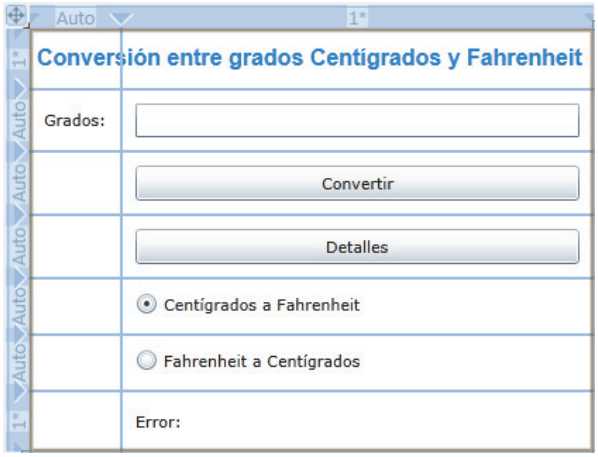


Un *proxy* se genera a partir de los metadatos obtenidos del servicio WCF e incluye información sobre los tipos y métodos expuestos por el servicio.

Pasemos a la parte práctica. Vamos a implementar un cliente WCF para consumir el servicio WCF que implementamos para convertir grados centígrados a Fahrenheit y viceversa anteriormente en este mismo capítulo. En nuestro caso, este cliente es la aplicación Silverlight, *ApSLConverTemps*, que construimos cuando iniciamos este ejemplo.

A continuación, diseñe la interfaz gráfica que mostrará esta aplicación para que tome el aspecto de la figura siguiente, en la cual se observa una etiqueta que pone título al formulario, otra etiqueta que especifica el dato que el usuario debe introducir en la caja de texto que hay a la derecha, un botón para convertir el valor en grados introducido en el sentido que indique la opción seleccionada, otro botón

que realiza la misma operación que el anterior pero dando una respuesta especificando literalmente si el resultado son grados centígrados o Fahrenheit, dos botones de opción para especificar en qué sentido se debe realizar la conversión y una etiqueta para mostrar un mensaje en el caso de que se produjera algún error.

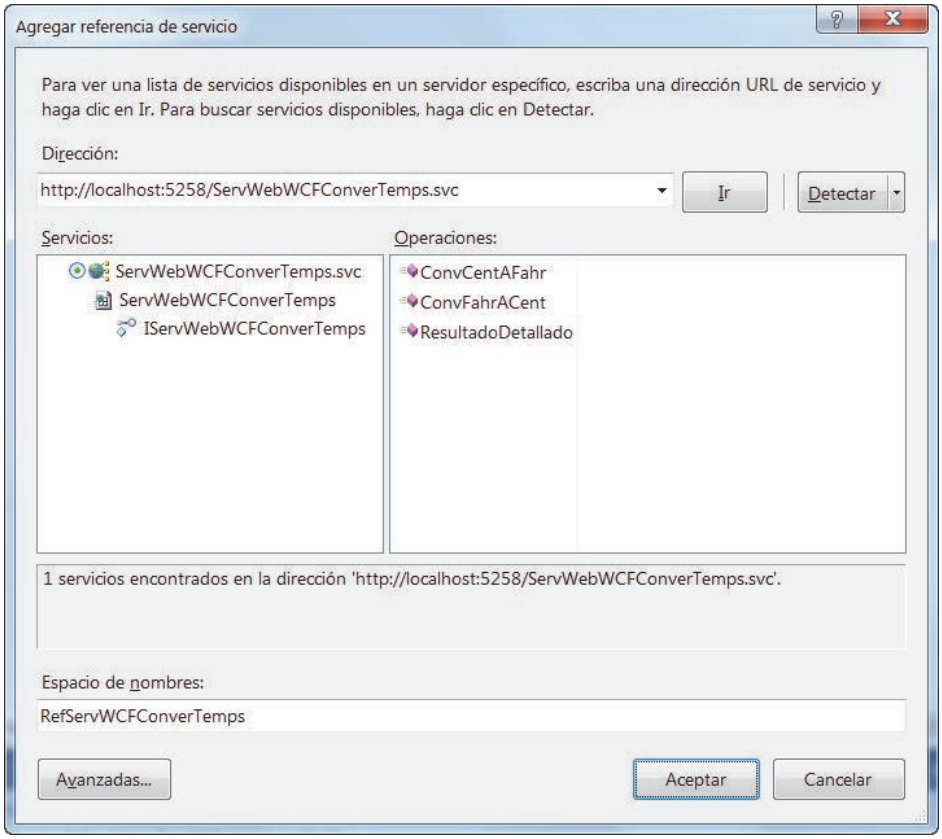


Pensemos ahora en cómo se sucederán los hechos cuando un usuario solicite que se ejecute esta aplicación Silverlight:

1. Se visualiza el formulario de la figura anterior.
2. El usuario selecciona el tipo de conversión que desea realizar haciendo clic sobre el botón de opción correspondiente.
3. Escribe en la caja de texto el valor en grados que desea convertir.
4. Hace clic en el botón *Convertir*. El controlador de este botón, en función del tipo de conversión seleccionado, invocará al método *ConvCentAFahr* o *ConvFahrACent* pasando como argumento el valor de la caja de texto.
5. O bien hace clic en el botón *Detalles*. El controlador de este botón, en función del tipo de conversión seleccionado, invocará al método *ResultadoDetallado* pasando como argumento el objeto *Detalles* que devolverá el resultado detallado en función de los valores fijados para sus propiedades *SonCentigrados* y *Grados*.
6. El resultado se muestra en la caja de texto.

Para llevar a cabo el punto 4 o el 5, la aplicación cliente debe tener un medio de comunicarse con el servicio WCF. Esto lo permite una infraestructura (un *proxy*) que se genera cuando agregamos al proyecto Silverlight una referencia al servicio. El *proxy* no es más que una clase que proporciona al cliente una representación local del servicio para interactuar con él. Entonces, para tener acceso a

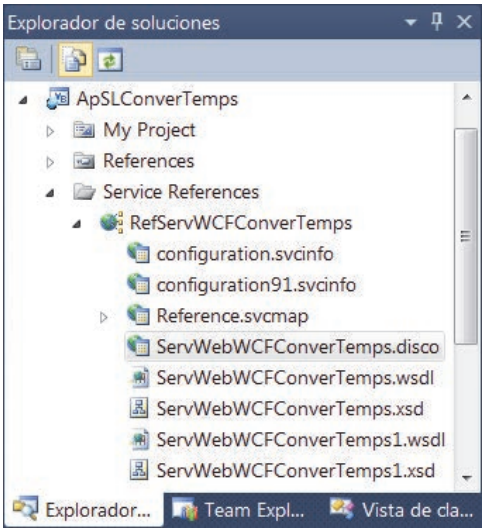
las operaciones proporcionadas por el servicio WCF *ServWebWCFConverTemps*, diríjase al explorador de soluciones y haga clic con el botón secundario del ratón sobre el nombre del proyecto *ApSLConverTemps* y seleccione la opción *Agregar referencia de servicio*. Después, por medio del diálogo que se visualiza, localice el servicio WCF, asigne un nombre para el espacio de nombres del *proxy* que se generará y haga clic en el botón *Aceptar*:



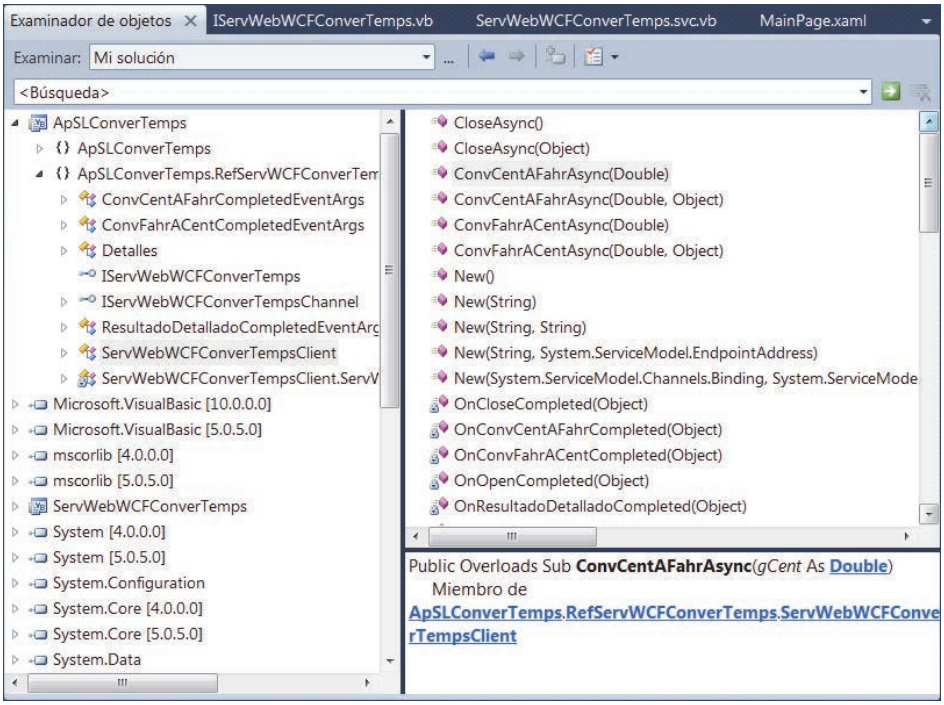
El hecho de añadir la referencia de servicio al proyecto desencadenó cosas interesantes. En la carpeta de referencias de servicio del proyecto (*Service References*) se ha creado una nueva subcarpeta con el mismo nombre dado al espacio de nombres (en nuestro caso *RefServWCFConverTemps*), en la que se han creado varios ficheros que puede observar en la figura siguiente.

Por ejemplo, observamos el fichero de descubrimiento (*.disco*), los ficheros de esquemas (*.xsd*) y los ficheros de descripción del servicio (*.wsdl*). A partir de estos metadatos se genera un fichero de código, el cliente WCF (el *proxy*) que en nuestro ejemplo está definido por la clase *ServWebWCFConverTempsClient* del

espacio de nombres *RefServWCFConverTemps*, y un fichero de configuración, *ServiceReferences.ClientConfig*, para el servicio.



Si hace clic con el botón secundario del ratón sobre el nodo *RefServWCFConverTemps* y elige *Ver en el examinador de objetos*, observará en el panel correspondiente la implementación del *proxy*, según muestra la figura siguiente:



En la figura anterior podemos observar que el diseño de la clase *ServWebWCFConverTempsClient* se apoya en el modelo asincrónico basado en eventos, por eso proporciona métodos denominados *NombreMetodoAsync* y eventos denominados *OnNombreMetodoCompleted* para notificar que el método al que se refiere finalizó. Estos métodos pueden reflejar también versiones sincrónicas, que realizan la misma operación en el subproceso actual.

Lo anterior pone de manifiesto que las aplicaciones que realizan simultáneamente muchas tareas, con frecuencia requieren un diseño basado en subprocesos (hilos). Precisamente, el modelo asincrónico basado en eventos pone a disposición del usuario las ventajas de las aplicaciones multiproceso, al tiempo que evita muchos de los problemas complejos inherentes a este tipo de diseño. Utilizar una clase que admita este modelo permite:

- Realizar tareas que exigen mucho tiempo, en segundo plano sin interrumpir la aplicación.
- Ejecutar simultáneamente varias operaciones y recibir la notificación correspondiente cuando finalice cada una de ellas.
- Esperar a que estén disponibles los recursos sin detener (colgar) la aplicación.
- Establecer comunicación con operaciones asíncronas pendientes utilizando el modelo habitual de eventos y delegados.

Precisamente, un objeto de la clase *ServWebWCFConverTempsClient* es el que nos permitirá acceder a las operaciones ofrecidas por el servicio.

Configuración del cliente

También se ha añadido al proyecto Silverlight el fichero de configuración *ServiceReferences.ClientConfig*:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IServWebWCFConverTemps"
                 maxBufferSize="2147483647"
                 maxReceivedMessageSize="2147483647">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint
        address="http://localhost:5258/ServWebWCFConverTemps.svc"
        binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IServWebWCFConverTemps"
```

```
        contract="RefServWCFConverTemps.IServWebWCFConverTemps"
        name="BasicHttpBinding_IServWebWCFConverTemps" />
    </client>
  </system.serviceModel>
</configuration>
```

La sección <*bindings*> describe los enlaces; un cliente Silverlight se comunica con un servicio utilizando el enlace *BasicHttpBinding* o el enlace *PollingDuplexHttpBinding*. Y la sección <*client*> contiene una lista de extremos (*endpoint*) que usa el cliente para conectarse a un servicio. En este caso, llama la atención la URL para conectarse al servicio: especifica un puerto, el correspondiente al servidor local de pruebas. Evidentemente, habrá que cambiar el valor del atributo *address* si el servicio cambia de ubicación, por ejemplo, cuando se publique.

Obtener acceso al servicio WCF

Una vez agregada en el cliente la referencia al servicio WCF, el siguiente paso es crear un objeto de la clase *ServWebWCFConverTempsClient* del *proxy* para acceder a los métodos del servicio. Cuando la aplicación cliente llama a estos métodos, es el *proxy* el que gestiona las comunicaciones entre la aplicación cliente y el servicio WCF. Según esto, edite el fichero de código subyacente *MainPage.xaml.vb* y agregue el código sombreado que se indica a continuación, lo que implica añadir el espacio de nombres *ApSLConverTemps.RefServWCFConverTemps*. También, hemos iniciado la etiqueta de mensajes de error con una cadena vacía, para que, mientras no haya errores, no se muestre nada al usuario.

```
Partial Public Class MainPage
  Inherits UserControl

  Private cliente As ServWebWCFConverTempsClient = Nothing

  Public Sub New()
    InitializeComponent()
    etError.Content = ""
    cliente = New ServWebWCFConverTempsClient()
  End Sub
  ' ...
End Class
```

Todas las llamadas que se realicen al servicio web desde Silverlight son *asíncronas*. Por ejemplo, consideremos la operación del servicio *ConvCentAFahr*. Realizar esta operación implica llamar al método *ConvCentAFahrAsync*. Una vez realizada la llamada al método asíncrono, el control se devuelve a la aplicación Silverlight para que el usuario pueda seguir interactuando con la misma. Cuando el servicio devuelva los datos que solicitamos, lo notificará a la aplicación generando el evento *ConvCentAFahrCompleted*, según puede observar en la informa-

ción mostrada anteriormente por el examinador de objetos. Entonces, tenemos que interceptar este evento para poder recoger el resultado. Ídem para las operaciones *ConvFahrACent* y *ResultadoDetallado*. Esto es lo que hace el código indicado a continuación:

```
Partial Public Class MainPage
  Inherits UserControl
  Private cliente As ServWebWCFConverTempsClient = Nothing

  Public Sub New()
    InitializeComponent()
    etError.Content = ""
    cliente = New ServWebWCFConverTempsClient()
    AddHandler cliente.ConvCentAFahrCompleted,
      AddressOf cliente_ConvCentAFahrCompleted
    AddHandler cliente.ConvFahrACentCompleted,
      AddressOf cliente_ConvFahrACentCompleted
    AddHandler cliente.ResultadoDetalladoCompleted,
      AddressOf cliente_ResultadoDetalladoCompleted
  End Sub
  ' ...
End Class
```

Cada uno de estos controladores de eventos indica que se debe llamar al método especificado cuando el servicio devuelva los datos solicitados. Por lo tanto, definimos estos métodos en el ámbito de la clase *MainPage* de forma que, no sólo proporcionen el resultado, sino que también controlen los errores que puedan ocurrir, como que el servicio no se encuentre disponible o que no esté configurado para admitir protocolos que Silverlight entienda o que la dirección del mismo no sea correcta. Para controlar estos errores, es preciso interrogar la propiedad **Error** del parámetro que proporciona los datos del evento, antes de obtener acceso a la propiedad **Result** que proporciona los datos solicitados. En otro caso, si se accede a la propiedad **Result** cuando se ha producido un error, se lanzará una excepción. El código mostrado a continuación muestra la implementación de estos controladores en los términos expuestos.

```
Private Sub cliente_ConvCentAFahrCompleted(sender As Object,
e As ConvCentAFahrCompletedEventArgs)
  If e.Error IsNot Nothing Then
    etError.Content = "Error en la conversión de grados C a F"
  Else
    ctGrados.Text = Convert.ToString(e.Result)
  End If
End Sub

Private Sub cliente_ConvFahrACentCompleted(sender As Object,
e As ConvFahrACentCompletedEventArgs)
  If e.Error IsNot Nothing Then
```

```
    etError.Content = "Error en la conversión de grados F a C"
  Else
    ctGrados.Text = Convert.ToString(e.Result)
  End If
End Sub

Private Sub cliente_ResultadoDetalladoCompleted(sender As Object,
e As ResultadoDetalladoCompletedEventArgs)
  If e.Error IsNot Nothing Then
    etError.Content = "Error al obtener detalles del resultado"
  Else
    ctGrados.Text = e.Result.Literal
  End If
End Sub
```

Sólo queda realizar las llamadas a los métodos asíncronos a los que nos hemos referido anteriormente. Éstas se realizarán desde los métodos que respondan al evento **Click** de los botones *Convertir* y *Detalles*. Por lo tanto, haga doble clic en estos botones para agregar a *MainPage* estos manejadores de eventos. Después, complételos como se indica a continuación:

```
Private Sub btConvertir_Click(sender As Object, e As RoutedEventArgs)
  etError.Content = ""

  ' Obtener el valor escrito en la caja de texto
  Try
    nGrados = Convert.ToDouble(ctGrados.Text)
    ' Realizar la conversión invocando al método correspondiente
    If boConvCF.IsChecked = True Then
      cliente.ConvCentAFahrAsync(nGrados)
    End If
    If boConvFC.IsChecked = True Then
      cliente.ConvFahrACentAsync(nGrados)
    End If
  Catch exc As Exception
    etError.Content = exc.Message
  End Try
End Sub

Private Sub btDetalles_Click(sender As Object, e As RoutedEventArgs)
  etError.Content = ""
  Try
    nGrados = Convert.ToDouble(ctGrados.Text)
    Dim detalle As New Detalles()
    detalle.Grados = nGrados
    detalle.SonCentigrados = CBool(boConvCF.IsChecked)
    cliente.ResultadoDetalladoAsync(detalle)
  Catch exc As Exception
    etError.Content = exc.Message
```

```
  End Try
End Sub
```

Ambos métodos utilizan la variable *Grados* para almacenar el valor introducido por el usuario en la caja de texto. Por lo tanto, defina esta variable como atributo de la clase:

```
Partial Public Class MainPage
  Inherits UserControl
  Private cliente As ServWebWCFConverTempsClient = Nothing
  Private nGrados As Double
  ...
End Class
```

Obsérvese que, en ambos métodos, se toma el valor proporcionado por el usuario en la caja de texto *ctGrados* y, utilizando el objeto de la clase *ServWebWCFConverTempsClient*, se llama al método adecuado del servicio WCF por medio del método correspondiente en el *proxy*. El resultado, valor devuelto por el servicio WCF, se muestra en la misma caja de texto.

En principio, la aplicación está finalizada, pero al ejecutarla en modo depuración observamos que se produce el siguiente error:

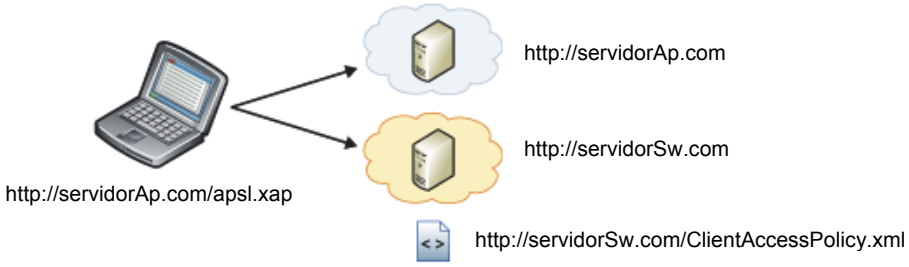
Se ha producido un error al intentar realizar una solicitud al URI 'http://localhost:5258/ServWebWCFConverTemps.svc'. Puede deberse a un intento de acceso a un servicio de ***configuración entre dominios*** *sin una política entre dominios adecuada en contexto o una política no apropiada para servicios SOAP. Es posible que necesite ponerse en contacto con el propietario del servicio para* ***publicar un archivo de política entre dominios*** *y para asegurarse de que permite enviar encabezados HTTP relacionados con SOAP…*

Veamos a continuación cómo se da solución a este problema.

Comunicación entre dominios

De forma predeterminada, Silverlight admite llamadas a servicios webs que se encuentran en el mismo dominio o sitio de origen. ¿Qué significa esto? Pues que las llamadas, por seguridad, deben utilizar el mismo subdominio, protocolo y puerto, lo cual evita falsificación entre dominios. Esto es, una aplicación Silverlight que usa la configuración predeterminada no puede llamar con un protocolo diferente (por ejemplo, HTTPS), no puede llamar a un servidor diferente, a un subdominio diferente o a un puerto diferente. Por lo tanto, para que una aplicación Silverlight pueda llamar a un servicio web de otro dominio, el servicio web tiene que usar un fichero de directivas entre dominios llamado *ClientAccessPolicy.xml*.

La figura siguiente ilustra cómo se habilita la comunicación entre dominios. Un usuario está ejecutando una aplicación Silverlight alojada en *servidorAp.com* la cual accede a un servicio web alojado en *servidorSw.com* que usa el fichero de directivas *ClientAccessPolicy.xml* para permitir a la aplicación acceder al mismo.



En general, cuando una aplicación Silverlight detecta que una solicitud es entre dominios, lo primero que hace es buscar en la raíz de la aplicación del servicio web el fichero de directivas *ClientAccessPolicy.xml* que le permita el acceso a dicho servicio (entre dominios de Adobe Flash el fichero es *CrossDomain.xml*).

El fichero de directivas es un fichero XML que tiene un formato simple. Por ejemplo, el fichero mostrado a continuación tiene dos entradas de directiva.

```
<?xml version="1.0" encoding="utf-8"?>
  <access-policy>
    <cross-domain-access>
      <policy>
        <allow-from http-request-headers="SOAPAction">
          <domain uri="*"/>
        </allow-from>
        <grant-to>
          <resource path="/sw/" include-subpaths="true"/>
        </grant-to>
     </policy>
      <policy >
        <allow-from http-methods="*">
          <domain uri="www.servidorAp.com"/>
        </allow-from>
        <grant-to>
          <resource path="/sw/" include-subpaths="true"/>
        </grant-to>
     </policy>
    </cross-domain-access>
  </access-policy>
```

La primera directiva habilita las solicitudes GET y POST en la ruta de acceso *sw* del dominio y el campo de cabecera *SOAPAction* para todos los llamadores

(*). La segunda directiva habilita todas las solicitudes de método del dominio *www.servidorAp.com* en la ruta de acceso *sw*.

Según esto, vamos a añadir al proyecto *ServWebWCFConverTemps* que implementa nuestro servicio web el fichero de directivas *ClientAccessPolicy.xml* siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
  <access-policy>
    <cross-domain-access>
      <policy>
        <allow-from http-request-headers="*">
          <domain uri="*"/>
        </allow-from>
        <grant-to>
          <resource path="/" include-subpaths="true"/>
        </grant-to>
      </policy>
    </cross-domain-access>
  </access-policy>
```

De esta forma, cuando una aplicación Silverlight *ApSLConverTemps.xap* detecte la solicitudes entre dominios por el hecho de llamar al servicio web, buscará en la raíz de la aplicación *ServWebWCFConverTemps* el fichero de directivas *ClientAccessPolicy.xml* que le permita el acceso a dicho servicio. En este caso, el fichero tiene una entrada de directiva que habilita las solicitudes GET y POST en la ruta de acceso / del dominio y todos los campos de cabecera para todos los llamadores (*).

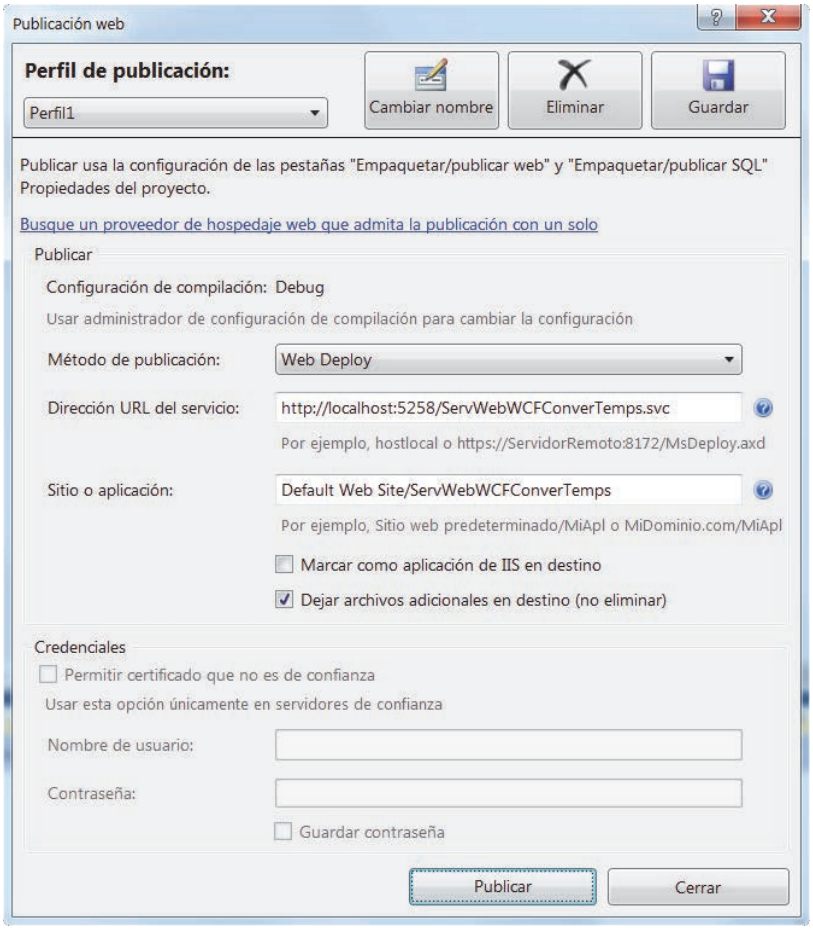
Pruebe ahora la aplicación y observará que todo funciona correctamente.

Publicar la aplicación

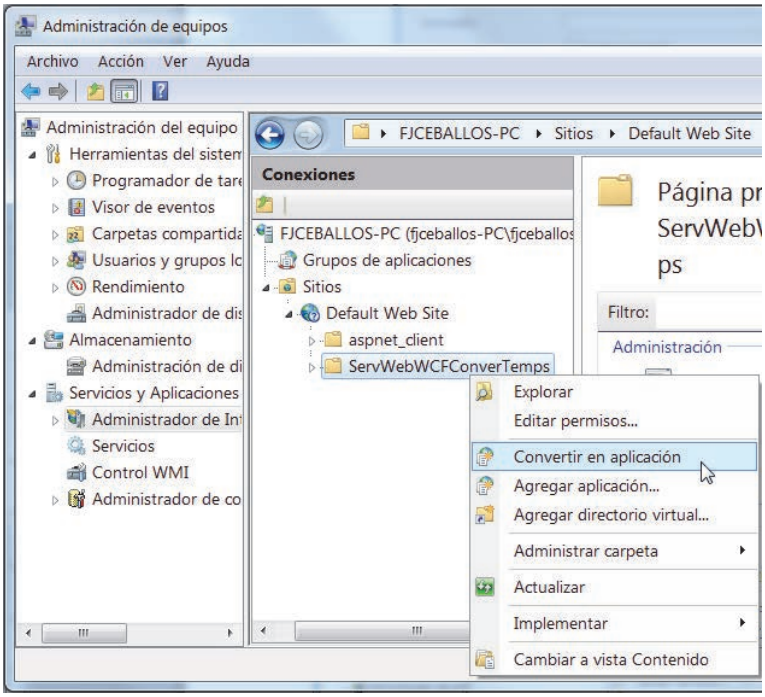
Publicar en un servidor IIS una aplicación Silverlight que accede a un servicio web que no pertenece al mismo dominio que la aplicación implica publicar el servicio web, actualizar la referencia al servicio de la aplicación Silverlight para que apunte a esta nueva ubicación y publicar la aplicación Silverlight.

Según explicamos en el capítulo *Silverlight*, este proceso resulta muy sencillo si se utiliza el asistente para publicación de sitios webs que proporciona Visual Studio. Empecemos por publicar el servicio. Para ello, necesita conocer la dirección URL del servicio, clic con el botón secundario del ratón sobre el nombre del fichero *ServWebWCFConverTemps.svc* > *Ver en el explorador*, y el sitio web donde se va a alojar. Una vez conocidos estos datos, haga clic con el botón secundario del ratón sobre el nombre del proyecto del servicio web (*ServWeb-*

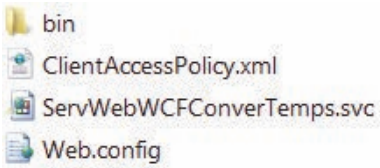
WCFConverTemps) y elija la opción *Publicar* (tiene que haber iniciado Visual Studio como administrador). Después complete el formulario análogamente a como indica la figura siguiente:



Una vez publicado el servicio, complete esta tarea especificando el nombre del directorio virtual. Para ello, abra el administrador de IIS, seleccione la carpeta *ServWebWCFConverTemps* en el sitio web, haga clic sobre ella con el botón secundario del ratón, seleccione la opción *Convertir en aplicación* y seleccione como grupo de aplicaciones ASP.NET v4.0.



Cuando haya finalizado, podrá verificar que se ha creado una carpeta, en el ejemplo *C:\inetpub\wwwroot\ServWebWCFConverTemps*, virtual vinculada con la carpeta física localizada en la misma ruta que contiene los ficheros que componen el servicio WCF. El contenido de esta carpeta es el mostrado por la siguiente figura:



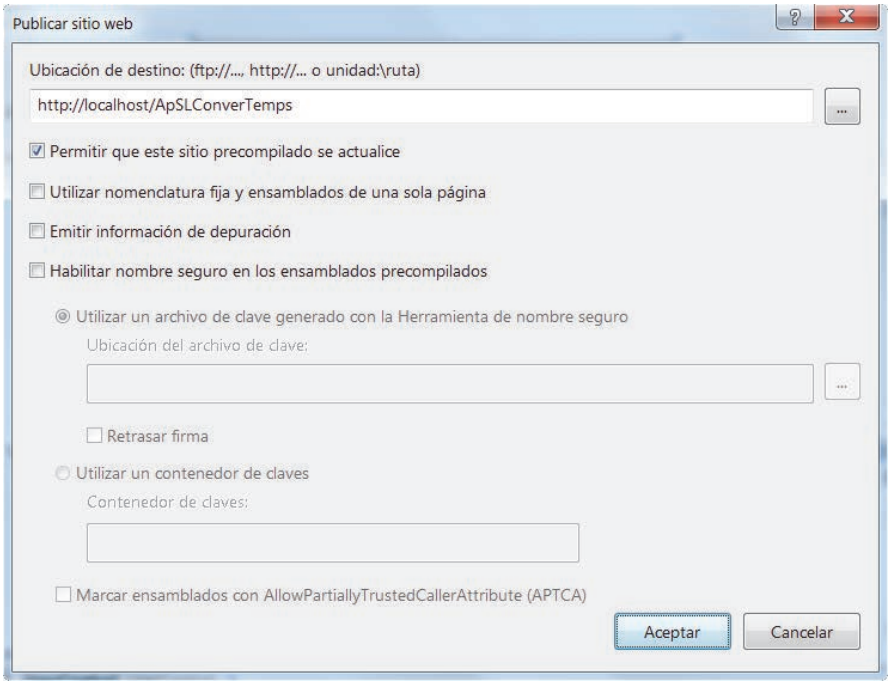
La carpeta *bin* contiene el servicio *ServWebWCFConverTemps.dll*.

Ahora, la aplicación Silverlight tendrá que hacer referencia a este servicio. Por lo tanto, elimine en dicha aplicación la referencia de servicio anterior, *RefServWCFConverTemps*, y añada una nueva para esta nueva ubicación del servicio WCF:

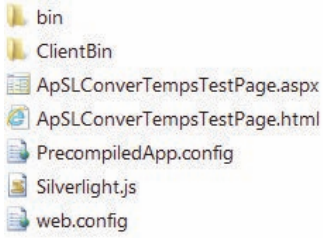
```
http://localhost/ServWebWCFConverTemps/ServWebWCFConverTemps.svc
```

Una vez añadida la referencia al servicio ubicado en IIS, compile la aplicación y publíquela. Para ello, haga clic con el botón secundario del ratón sobre el

nombre del proyecto web (no sobre el proyecto Silverlight) y elija la opción *Publicar sitio web* (tiene que haber iniciado Visual Studio como administrador). Después complete el formulario análogamente a como indica la figura siguiente:



Cuando haya finalizado el trabajo con el asistente, podrá verificar que se ha creado una carpeta, en el ejemplo *C:\inetpub\wwwroot\ApSLConverTemps*, virtual vinculada con la carpeta física localizada en la misma ruta que contiene los ficheros que componen la aplicación Silverlight. El contenido de esta carpeta para nuestro ejemplo es el mostrado por la siguiente figura:



Para este ejemplo, la carpeta *bin* está vacía y la carpeta *ClienBin* contiene el fichero *.xap* que es, según explicamos en el capítulo *Silverlight*, el ensamblado correspondiente al proyecto Silverlight empaquetado. En cuanto al fichero *web.config*, especifica el atributo *targetFramework* de la etiqueta <*compilation*>

para especificar cuál es la versión de .NET Framework de destino de un proyecto y generar únicamente código válido para esa versión:

```
<compilation debug="true" targetFramework="4.0"/>
```

Esto implica incluir la aplicación en el grupo de aplicaciones ASP.NET v4.0, operación que haremos desde el administrador de IIS.

Para verificar que todo funciona correctamente, abra el explorador y solicite desde el mismo la ejecución de la aplicación:

```
http://localhost/ApSLConverTemps/ApSLConverTempsTestPage.aspx
```

SERVICIOS WCF HABILITADOS PARA SILVERLIGHT

Con frecuencia, las aplicaciones de Silverlight necesitan tener acceso a datos o funcionalidad que puede ser suministrada por el propio servidor web en el que se aloja la aplicación mediante servicios webs. Por ejemplo, si va a escribir una aplicación para administrar una lista de teléfonos, el servidor web podría contener una base de datos con los teléfonos y demás datos de las personas que la aplicación Silverlight necesite recuperar o si, como en el ejemplo anterior, necesitamos un servicio que convierta grados centígrados a Fahrenheit y viceversa, el servidor web podría proporcionar este servicio junto con la aplicación Silverlight.

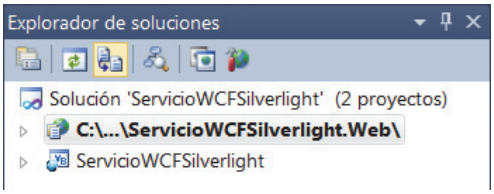
Sin duda alguna, la forma más efectiva que tiene una aplicación Silverlight para aprovechar el código del lado del servidor es mediante servicios webs. Por eso, en este apartado se describe cómo agregar un servicio WCF habilitado para Silverlight a un proyecto Silverlight existente. La idea básica es incluir un servicio web en el sitio web ASP.NET que lanza la aplicación Silverlight, para que dicha aplicación pueda interactuar con la interfaz programática que ofrece ese servicio. De esta forma, el servicio puede proporcionar contenido generado en el lado del servidor que no está disponible en el lado del cliente. Dicho contenido puede provenir de cálculos computacionales o de consultas a bases de datos. Estos servicios pueden ser servicios webs tradicionales (servicios *.asmx*) o servicios WCF habilitados para Silverlight.

Como ejemplo, vamos a reproducir la aplicación desarrollada anteriormente, pero utilizando ahora un servicio WCF habilitado para Silverlight.

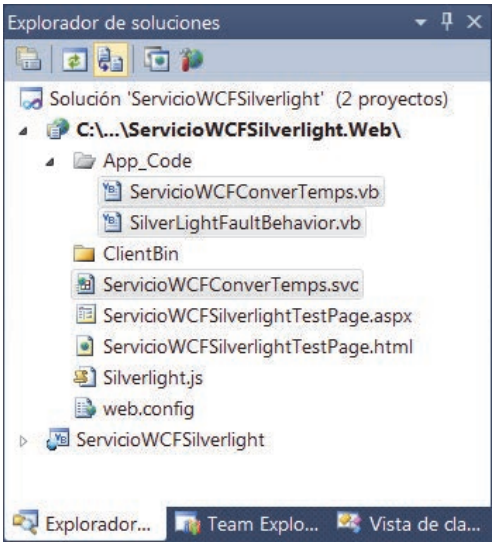
Crear un servicio WCF habilitado para Silverlight

Para poner en práctica lo expuesto en el párrafo anterior, vamos a crear una aplicación Silverlight que incluya un servicio WCF que proporcione a la aplicación

Silverlight las operaciones que permitan convertir un valor en grados Fahrenheit a su correspondiente valor en grados centígrados y viceversa. Empecemos entonces por crear la aplicación. Abra Visual Studio y cree una nueva aplicación Silverlight: seleccione *Archivo* > *Nuevo* > *Proyecto* > *Visual Basic* > *Silverlight* > *Aplicación de Silverlight*. Denomine al proyecto *ServicioWCFSilverlight*, por ejemplo. Seleccione también la casilla de verificación *Crear directorio para la solución* para agrupar en una misma carpeta los proyectos que se crearán. Después, haga clic en *Aceptar* y hospede la aplicación Silverlight en un sitio web ASP.NET. Se generará una solución con dos proyectos: *ServicioWCFSilverlight*, proyecto Silverlight, y *ServicioWCFSilverlight.Web*, proyecto web.



A continuación, añada un servicio WCF habilitado para Silverlight al sitio web. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto web, *ServicioWCFSilverlight.Web*, y seleccione *Agregar nuevo elemento* > *Servicio WCF habilitado para Silverlight*. Denomínelo *ServicioWCFConverTemps*. Después haga clic en *Aceptar*.



La acción de añadir un servicio WCF al sitio web ha agregado al proyecto web una nueva carpeta *App_Code* con dos ficheros: *ServicioWCFConverTemps.vb*, que define el contrato del servicio, *SilverlightFaultBehavior.vb*, que

define la conducta a seguir para que los errores de WCF lleguen al cliente Silverlight, y el fichero *ServicioWCFConverTemps.svc* que define el extremo del servicio (este fichero incluye una línea de marcado que le dice a ASP.NET dónde encontrar el código del servicio web).

A continuación, los pasos que tenemos que seguir para completar la aplicación son los mismos que expusimos cuando realizamos este ejemplo anteriormente, por lo que no abundaremos en detalles. En primer lugar, vamos a definir el contrato del servicio (**ServiceContract**) y el contrato de datos (**DataContract**). Ambos serán definidos en el fichero *ServicioWCFConverTemps.vb*. Este fichero de código empieza básicamente con dos atributos: el atributo **ServiceContract**, que representa el contrato del servicio, y el atributo **AspNetCompatibilityRequirements**, que indica si el servicio se puede ejecutar en modo compatible con ASP.NET, lo cual le permite utilizar características ASP.NET como, por ejemplo, autorización de archivo, autorización URL y estado de sesión HTTP; de forma predeterminada no hay compatibilidad. A continuación se escribe la clase que define el servicio; todas las operaciones que debe exponer el servicio a los clientes estarán definidas por métodos de la clase precedidos por el atributo **OperationContract**.

```
Imports System.ServiceModel
Imports System.ServiceModel.Activation
Imports System.Runtime.Serialization

<ServiceContract(Namespace:="http://websol.aut.uah.es/ServiciosWeb/")>
<SilverLightFaultBehavior()>
<AspNetCompatibilityRequirements(RequirementsMode:=
   AspNetCompatibilityRequirementsMode.Allowed)>
Public Class ServicioWCFConverTemps

  <OperationContract()>
  Public Function ConvCentAFahr(gCent As Double) As Double
    Dim grados As Double = Math.Round(9.0 / 5.0 * gCent + 32.0, 2)
    Return grados
  End Function

  Public Function ConvFahrACent(gFahr As Double) As Double
    Dim grados As Double = Math.Round(((gFahr - 32.0) * 5.0) / 9.0, 2)
    Return grados
  End Function

  Public Function ResultadoDetallado(detalle As Detalles) As Detalles
    If detalle Is Nothing Then
      Throw New ArgumentNullException("detalle")
    End If
    If detalle.SonCentigrados Then
      detalle.Literal = ConvCentAFahr(detalle.Grados) &
        " grados fahrenheit."
    Else
```

```
      detalle.Literal = ConvFahrACent(detalle.Grados) &
        " grados centígrados."
    End If
    Return detalle
  End Function
End Class

<DataContract(Namespace:="http://websol.aut.uah.es/ServiciosWeb/")>
Public Class Detalles
  Private _sonCentigrados As Boolean
  Private _literal As String
  Private _grados As Double

  <DataMember()>
  Public Property SonCentigrados() As Boolean
    Get
      Return _sonCentigrados
    End Get
    Set(value As Boolean)
      _sonCentigrados = value
    End Set
  End Property

  <DataMember()>
  Public Property Literal() As String
    Get
      Return _literal
    End Get
    Set(value As String)
      _literal = value
    End Set
  End Property

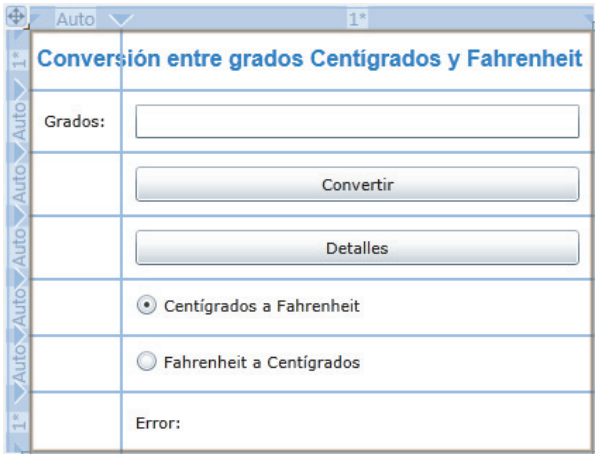
  <DataMember()>
  Public Property Grados() As Double
    Get
      Return _grados
    End Get
    Set(value As Double)
      _grados = value
    End Set
  End Property
End Class
```

Además del contrato del servicio, puede utilizar un contrato de datos (**DataContract**), como se ilustra en el ejemplo anterior, para agregar tipos compuestos a las operaciones del servicio.

Implementar un cliente WCF

El cliente es la propia aplicación Silverlight, *ServicioWCFSilverlight*, que construimos cuando iniciamos este ejemplo. Por lo tanto, como siguiente paso, diseñe una interfaz gráfica igual a la del ejemplo anterior (véase la figura siguiente).

Según esa interfaz, cuando el usuario seleccione el tipo de conversión que desea realizar y escriba en la caja de texto el valor en grados que desea convertir, hará clic en el botón *Convertir* o bien en el botón *Detalles*, para obtener el valor convertido, o convertido y detallado, operación que será llevada a cabo por los controladores del evento clic de estos botones, para lo cual, la aplicación cliente necesita un medio para comunicarse con el servicio WCF: un *proxy*.



Añadir una referencia al servicio

Para obtener acceso a las operaciones proporcionadas por el servicio WCF vamos a añadir al proyecto Silverlight una referencia al servicio: diríjase al explorador de soluciones y haga clic con el botón secundario del ratón sobre el nombre *ServicioWCFSilverlight* del proyecto Silverlight y seleccione la opción *Agregar referencia de servicio*. Después, por medio del diálogo que se visualiza, localice el servicio WCF, asigne un nombre para el espacio de nombres del *proxy* que se generará, por ejemplo *RefServWCFConverTemps*, y haga clic en el botón *Aceptar*.

Una vez agregada en el cliente la referencia al servicio WCF, el siguiente paso es crear un objeto de la clase *ServicioWCFConverTempsClient* del *proxy* para acceder a los métodos del servicio, lo que implica añadir el espacio de nombres *ServicioWCFSilverlight.RefServWCFConverTemps*. Recuerde que todas las llamadas que se realicen al servicio web desde Silverlight son *asíncronas*, según puede ver en el código expuesto a continuación, correspondiente al fichero

MainPage.xaml.vb. Para cualquier detalle sobre este desarrollo, repase el desarrollo pormenorizado que hicimos anteriormente en este mismo capítulo.

```
Imports ServicioWCFSilverlight.RefServWCFConverTemps

Partial Public Class MainPage
  Inherits UserControl
  Private cliente As ServicioWCFConverTempsClient = Nothing
  Private nGrados As Double

  Public Sub New()
    InitializeComponent()
    etError.Content = ""
    cliente = New ServicioWCFConverTempsClient()

    AddHandler cliente.ConvCentAFahrCompleted,
      AddressOf cliente_ConvCentAFahrCompleted
    AddHandler cliente.ConvFahrACentCompleted,
      AddressOf cliente_ConvFahrACentCompleted
    AddHandler cliente.ResultadoDetalladoCompleted,
      AddressOf cliente_ResultadoDetalladoCompleted
  End Sub

  Private Sub cliente_ConvCentAFahrCompleted(sender As Object,
  e As ConvCentAFahrCompletedEventArgs)
    If e.Error IsNot Nothing Then
      etError.Content = "Error en la conversión de grados C a F"
    Else
      ctGrados.Text = Convert.ToString(e.Result)
    End If
  End Sub

  Private Sub cliente_ConvFahrACentCompleted(sender As Object,
  e As ConvFahrACentCompletedEventArgs)
    If e.Error IsNot Nothing Then
      etError.Content = "Error en la conversión de grados F a C"
    Else
      ctGrados.Text = Convert.ToString(e.Result)
    End If
  End Sub

  Private Sub cliente_ResultadoDetalladoCompleted(sender As Object,
  e As ResultadoDetalladoCompletedEventArgs)
    If e.Error IsNot Nothing Then
      etError.Content = "Error al obtener detalles del resultado"
    Else
      ctGrados.Text = e.Result.Literal
    End If
  End Sub
```

```
  Private Sub btConvertir_Click(sender As Object, e As RoutedEventArgs)
    etError.Content = ""
    ' Obtener el valor escrito en la caja de texto
    Try
      nGrados = Convert.ToDouble(ctGrados.Text)
      ' Realizar la conversión invocando al método correspondiente
      If boConvCF.IsChecked = True Then
        cliente.ConvCentAFahrAsync(nGrados)
      End If
      If boConvFC.IsChecked = True Then
        cliente.ConvFahrACentAsync(nGrados)
      End If
    Catch exc As Exception
      etError.Content = exc.Message
    End Try
  End Sub

  Private Sub btDetalles_Click(sender As Object, e As RoutedEventArgs)
    etError.Content = ""
    Try
      nGrados = Convert.ToDouble(ctGrados.Text)
      Dim detalle As New Detalles()
      detalle.Grados = nGrados
      detalle.SonCentigrados = CBool(boConvCF.IsChecked)
      cliente.ResultadoDetalladoAsync(detalle)
    Catch exc As Exception
      etError.Content = exc.Message
    End Try
  End Sub
End Class
```

Compile, ejecute la aplicación y observe los resultados. Como conclusión, esta forma de incluir un servicio para ser consumido por una aplicación Silverlight resulta más sencilla y no presenta los problemas de la comunicación entre dominios que expusimos anteriormente, por lo que, siempre que satisfaga nuestras necesidades, es recomendable utilizarla.

Publicar la aplicación

Publicar en un servidor IIS una aplicación Silverlight que accede a un servicio web que pertenece al mismo dominio que la aplicación, simplemente, requiere desplegar la aplicación Silverlight, puesto que el servicio web forma parte del proyecto web utilizado para ejecutar la aplicación.

Por lo que hemos estudiado hasta ahora, este proceso resulta muy sencillo si se utiliza el asistente para publicación de sitios webs que proporciona Visual Stu-

dio. No obstante, antes de realizar el despliegue tenemos que hacer algunos ajustes relativos a la configuración del cliente.

Si observamos el fichero *ServiceReferences.ClientConfig* de configuración del cliente Silverlight, vemos que el atributo *address* del extremo del servicio hace referencia a un servidor concreto (en el ejemplo, a *localhost:2551*):

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding
          name="CustomBinding_ServicioWCFConverTemps">
          <binaryMessageEncoding />
          <httpTransport maxReceivedMessageSize="2147483647"
                         maxBufferSize="2147483647" />
        </binding>
      </customBinding>
    </bindings>
    <client>
      <endpoint
        address="http://localhost:2551/ServicioWCFSilverlight.Web/
                 ServicioWCFConverTemps.svc"
        binding="customBinding"
        bindingConfiguration="CustomBinding_ServicioWCFConverTemps"
        contract="RefServWCFConverTemps.ServicioWCFConverTemps"
        name="CustomBinding_ServicioWCFConverTemps" />
    </client>
  </system.serviceModel>
</configuration>
```

Según vimos anteriormente en el apartado *Configuración del cliente*, cuando se usa un enlace estándar en Silverlight como, por ejemplo, *BasicHttpBinding*, el elemento <*security*> de la sección <*bindings*> especifica el modo de seguridad: *None* o *TransportCredentialOnly* en HTTP. En cambio, cuando se usa *CustomBinding*, como es el caso, la seguridad de nivel de transporte se controla mediante <*httpTransport*> o <*httpsTransport*>. Y la sección <*client*>, como ya dijimos, contiene una lista de extremos (*endpoint*) que usa el cliente para conectarse a un servicio. Pero si observamos el valor del atributo *address*, tiene asignado una URL correspondiente al servidor local de pruebas. Evidentemente, habrá que cambiar el valor de este atributo si el servicio va a cambiar de ubicación.

Pero esto no sería suficiente, ya que cuando añadimos la referencia de servicio al proyecto Silverlight, el *proxy* generado de forma automática incluye la URL actual del servicio web, lo que hace que no tengamos que especificar la URL cuando creamos un objeto (en nuestro ejemplo *cliente*) de la clase del *proxy*. Pero esto plantea un problema y es que esa URL es absoluta, ya que no se permiten

URL relativas (observe, por ejemplo, el número del puerto; si cambia, ya no se puede invocar al servicio). Según esto, tendríamos que actualizar también esta URL cuando vayamos a publicar la aplicación final en un servidor web de producción, pero hay una solución más fácil: cambiar la URL (el atributo *address* del extremo del objeto de la clase del *proxy*) dinámicamente.

Para cambiar la URL dinámicamente desde el código de la aplicación, tenemos que saber que la clase *ServicioWCFConverTempsClient* del *proxy* proporciona varios constructores, alguno de ellos, como los mostrados a continuación, con un primer parámetro que hace referencia al nombre del extremo del servicio (véase el fichero *ServiceReferences.ClientConfig*) y un segundo parámetro que hace referencia al atributo *address* de dicho extremo.

```
Public Sub New(endpointConfigurationName As String,
remoteAddress As String)
    MyBase.New(endpointConfigurationName, remoteAddress)
End Sub

Public Sub New(endpointConfigurationName As String,
remoteAddress As EndpointAddress)
    MyBase.New(endpointConfigurationName, remoteAddress)
End Sub
```

Según esto, el URL donde actualmente se aloja el servicio lo podemos proporcionar en el momento de construir el objeto de la clase del *proxy*, así:

```
Public Sub New()
  InitializeComponent()
  etError.Content = ""

  Dim uri As New Uri(Application.Current.Host.Source,
                     "../ServicioWCFConverTemps.svc")
  cliente = New ServicioWCFConverTempsClient(
    "CustomBinding_ServicioWCFConverTemps", uri.AbsoluteUri)
  ...
End Sub
```

El código anterior crea un objeto *uri* mediante la combinación del URI del fichero *.xap* que empaqueta la aplicación Silverlight y de la ruta relativa al *.xap* del fichero *.svc* y, después, crea un objeto de la clase del *proxy* pasando como argumentos el nombre del extremo del servicio y la dirección URL absoluta obtenida, que en nuestro caso será:

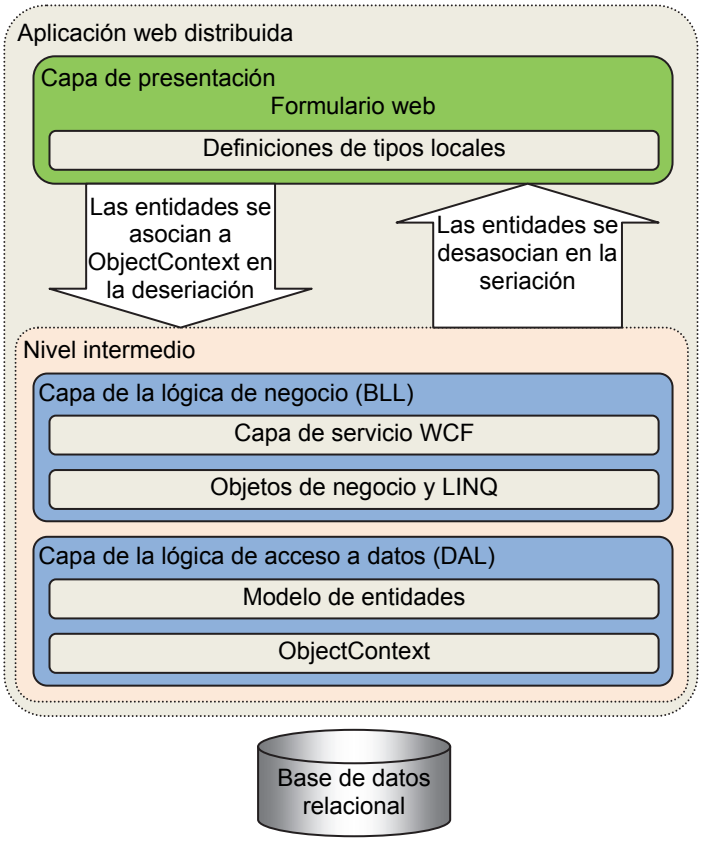
```
http://servidor/ServicioWCFSilverlight.Web/ServicioWCFConverTemps.svc
```

donde *servidor* dependerá del servidor donde esté alojado el servicio (por ejemplo: *localhost:2551*, *localhost*, *www.mi_servidor.com*, etc.).

SERVICIOS WEB Y LINQ

En el capítulo de *LINQ* vimos cómo utilizar ADO.NET *Entity Framework* para crear aplicaciones que acceden a bases de datos. Es aconsejable que repase los conceptos expuestos en ese capítulo.

Pues bien, utilizando un servicio web en una aplicación de N niveles, el nivel intermedio contendrá el contexto de objetos y las clases de entidad y, también, definirá la interfaz con los métodos que envolverán las consultas de LINQ que los clientes utilizarán para recuperar, insertar y actualizar datos. Por otra parte, el nivel de presentación debe contar con las definiciones de tipo para los datos que recibirá desde el nivel intermedio; por ejemplo, el nivel de presentación podrá generar los tipos a partir de la infraestructura de un servicio WCF. Esos tipos pueden ser clases de entidad propiamente dichas o clases especiales que envuelvan sólo ciertos campos de las clases de entidad.



En la figura anterior se puede observar que la capa de presentación llamará a los métodos de la interfaz remota de nivel intermedio y, en este nivel, la capa de

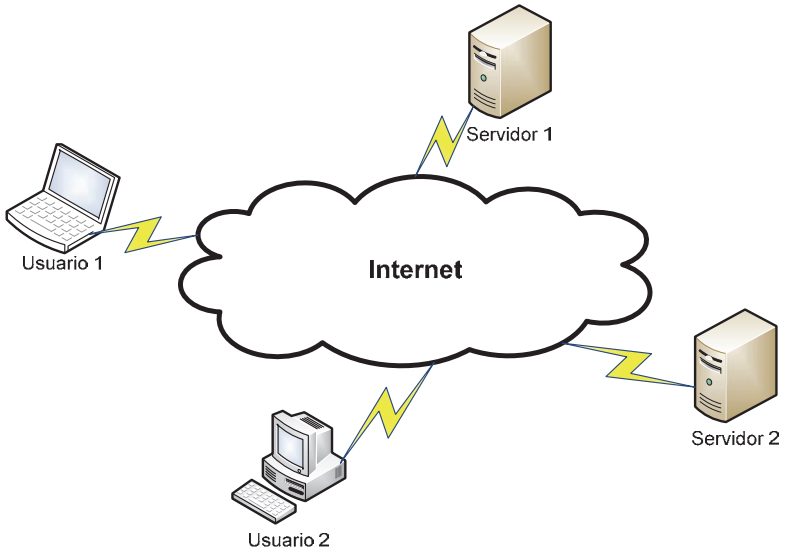
acceso a datos (DAL, *Data Access Layer*) ejecutará las consultas o procedimientos almacenados asignados a métodos de **ObjectContext**. El nivel intermedio devuelve los datos a los clientes generalmente como representaciones XML de entidades u objetos.

En el nivel intermedio, los objetos de negocio y las consultas sobre el modelo de objetos se encuentran en la capa de negocio (BLL, *Business Logic Layer*) y las clases de entidad y el contexto de objetos, en la capa de acceso a datos (DAL). Evidentemente, algunas de estas capas intermedias son opcionales, esto es, el que estén o no dependerá del diseño que se haga en cada caso. En este nivel, el contexto de objetos realiza el seguimiento del estado de las entidades que son creadas (véase el capítulo *LINQ*) y que están asociadas a dicho contexto. Sin embargo, una vez que las entidades se envían a otro nivel a través de la seriación, quedan desasociadas, lo cual significa que el **ObjectContext** ya no realiza el seguimiento de su estado, por eso, las entidades que el cliente devuelve para actualizar la base de datos se tienen que volver a asociar al contexto de objetos para que *Entity Framework* pueda enviar los cambios a dicha base de datos.

Cuando los datos vayan a ser grabados en la base de datos podría haber problemas de concurrencia o simultaneidad que, por ejemplo, podríamos solucionar utilizando valores originales para las comprobaciones de simultaneidad optimista y *Entity Framework* admite la comprobación de simultaneidad optimista basada en valores originales.

Arquitectura de N capas lógicas y N niveles físicos

Como ejemplo vamos a construir a continuación una aplicación de N capas (*N-Layer architecture*), generalmente, tres o más capas lógicas: presentación, lógica de negocio y acceso a datos, y N niveles (*N-Tier architecture*), cada nivel se refiere a una parte de la aplicación que se encuentra físicamente en un equipo independiente. De aquí que no todas las aplicaciones de N capas serán de N niveles, pero sí todas las aplicaciones de N niveles deberán estar diseñadas como de N capas. Por ejemplo, según lo expuesto, la capa de presentación y lógica de negocio propia de ésta, de la aplicación que vamos a desarrollar, podrá estar desplegada en un servidor con IIS (llamemos a esta parte nivel de cliente) y se comunicará con la base de datos a través de un servicio WCF que podrá estar desplegado, por ejemplo junto a la base de datos, en otro servidor (llamemos a esta otra parte nivel de servidor). La figura siguiente clarifica lo expuesto:



El despliegue propuesto para la aplicación requiere que el nivel de cliente se comunique con el nivel de servidor a través de un servicio WCF, por ejemplo. Ahora, cuando el cliente haga una petición al servidor para obtener unos determinados datos, el servicio WCF hará una consulta a la base de datos y mantendrá el resultado como una entidad en el contexto de *Entity Framework*. A continuación, la entidad obtenida será enviada al cliente, para lo cual, antes será desconectada (**Detach**) del contexto de objetos (ya que posteriormente tendrá que ser conectada de nuevo, y conectar una entidad no desconectada puede producir un error) y seriada por WCF. De esta forma, trabajando de forma desconectada, los datos de las tablas de la base de datos no serán bloqueados mientras el usuario trabaja en el cliente, lo que mejora la escalabilidad.

En el cliente, tal vez el usuario necesite modificar los datos de la entidad obtenida y enviar los cambios al servidor para que sean registrados en la base de datos. En este caso, podrían surgir problemas de simultaneidad, que podríamos solucionar, según dijimos anteriormente, utilizando los valores originales para las comprobaciones de simultaneidad optimista.

Crear la base de datos

Empecemos entonces por crear una base de datos que, en nuestro caso, vamos a llamar *bd_telefonos3* y que va a tener una tabla denominada *telefonos*:

```
CREATE TABLE telefonos
(
  idpersona     INTEGER IDENTITY(1,1) PRIMARY KEY,
```

```
  nombre        VARCHAR(16) NOT NULL,
  apellido1     VARCHAR(12) NOT NULL,
  apellido2     VARCHAR(12),
  direccion     VARCHAR(30) NOT NULL,
  telefono      VARCHAR(12) NOT NULL,
  observaciones VARCHAR(240),
  foto          VARBINARY(MAX)
)
```

La propiedad *IDENTITY* define la columna como una columna de identidad. Las columnas de identidad se utilizan normalmente con las restricciones PRIMARY KEY como identificadores de fila únicos de la tabla. En este caso, deben especificarse el valor de iniciación y el incremento, o ninguno, en cuyo caso se asumen los valores (1,1). Esto quiere decir que cuando se agregue una fila nueva a la tabla, el motor de base de datos proporciona un valor incremental único para la columna. En nuestro caso el primer *idpersona* será 1 y el incremento, 1.

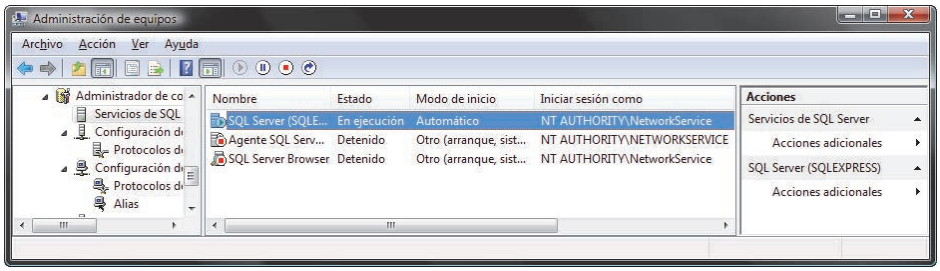
El tipo *binary[(n)]* permite definir tipos de datos binarios de longitud fija y *varbinary[(n|max)]*, variable. Utilice *binary* cuando los tamaños de las entradas de datos de columna sean coherentes, *varbinary* cuando los tamaños varíen considerablemente y *varbinary(max)* cuando superen los 8.000 bytes.

No olvide dar permisos al usuario *NT AUTHORITY\Servicio de Red* (*servidor\ASPNET* en Windows XP) para que pueda acceder a esta base de datos.

Obtener acceso a la base de datos

De forma predeterminada, y dependiendo de la versión de IIS que tenga en su sistema operativo Windows, las aplicaciones webs se ejecutan con la identidad de la cuenta *NT AUTHORITY\Servicio de Red*, *IIS APPPOOL\DefaultAppPool* u otra (en Windows XP lo hacen con la identidad de *ASPNET*). Se trata de cuentas con unos privilegios específicos (no tienen permisos para acceder a determinados recursos de la máquina), normalmente pocos, entre otras cosas, por seguridad.

Así mismo, cada servicio de SQL Server, para poder iniciarse y ejecutarse, debe tener una cuenta de usuario que se configura durante la instalación. Las cuentas de inicio utilizadas para iniciar y ejecutar SQL Server pueden ser cuentas del sistema integradas (por ejemplo *NT AUTHORITY\NetworkService*, *IIS APPPOOL\DefaultAppPool*, *NT AUTHORITY\SERVICIO LOCAL* o *NT AUTHORITY\SYSTEM*), cuentas de usuario local o cuentas de usuario de dominio.



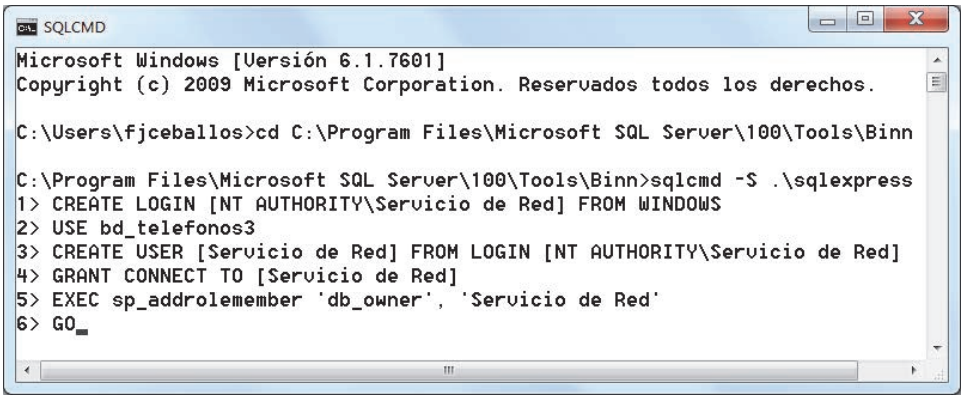
Por ejemplo, la cuenta *Servicio de red* es una cuenta integrada especial parecida a una cuenta de usuario autenticado, con un mayor nivel de acceso a los recursos y objetos que los miembros del grupo *usuarios*.

Entonces, para poder acceder a *bd_telefonos3* hay que dar permisos a este usuario para acceder a esa base de datos. Una forma de hacerlo es ejecutando el siguiente *script* desde la utilidad *SQLCMD* o desde *MS SQL Server Management Express* (de aquí en adelante me referiré al usuario *Servicio de Red* o *ASPNET*, pero, dependiendo de cómo configuremos IIS, puede ser otro).

```
CREATE LOGIN [NT AUTHORITY\Servicio de Red] FROM WINDOWS
USE bd_telefonos3
CREATE USER [Servicio de Red] FROM LOGIN [NT AUTHORITY\Servicio de Red]
GRANT CONNECT TO [Servicio de Red]
EXEC sp_addrolemember 'db_owner', 'Servicio de Red'
GO
```

Este *script* lo que hace es lo siguiente:

- Crea un inicio de sesión a partir de una cuenta de dominio de Windows (*nombreDeDominio\nombreDeInicioDeSesión*), en el ejemplo *NT AUTHORITY\Servicio de Red*. Hay cuatro tipos de inicio de sesión: de SQL Server, de Windows, asignado a un certificado y asignado a una clave asimétrica. Esta primera línea puede omitirse si la entidad de seguridad de servidor *NT AUTHORITY\Servicio de Red* ya existe.
- Crea un usuario (*Servicio de Red*) para la base de datos especificada (en nuestro caso *bd_telefonos3*) correspondiente al inicio de sesión.
- Concede permiso al usuario para acceder a la base de datos.
- Agrega el usuario *Servicio de Red* al *rol* de base de datos *db_owner* en la base de datos actual. Los miembros de la función *db_owner* pueden realizar todas las tareas de configuración y mantenimiento de la base de datos.



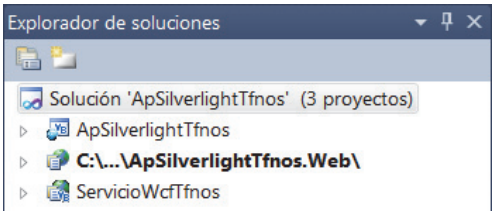
Estas operaciones puede también realizarlas directamente desde el entorno gráfico de la utilidad *MS SQL Server Management Express*.

Crear el servicio WCF

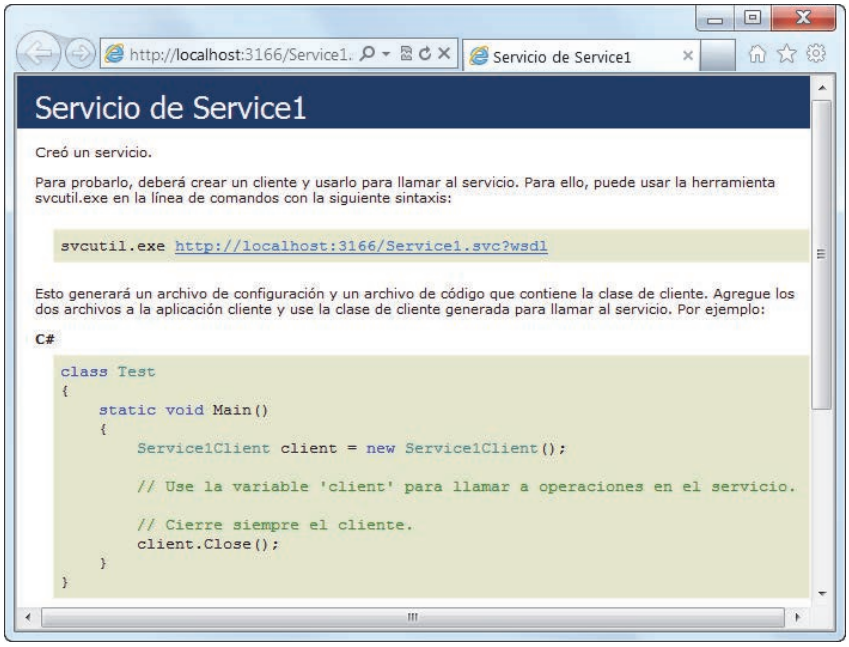
Una vez creada la base de datos, pasamos a crear el servicio WCF. Para ello, abrimos Visual Studio y creamos una nueva solución Silverlight que agrupe los proyectos que formen parte de esta aplicación.

En principio, construiremos la aplicación Silverlight y el servicio WCF sobre nuestro sistema de ficheros y después publicaremos ambos por medio de IIS. Entonces, abra Visual Studio como administrador y cree una nueva aplicación Silverlight: *Archivo* > *Nuevo* > *Proyecto* > *Visual Basic* > *Silverlight* > *Aplicación de Silverlight*. Denomine al proyecto *ApSilverlightTfnos*, por ejemplo. Seleccione también la casilla de verificación *Crear directorio para la solución* para agrupar en una misma carpeta los proyectos que se crearán. Hospede la aplicación Silverlight en un nuevo sitio web, *ApSilverlightTfnos.Web*, tipo *Sitio web ASP.NET*.

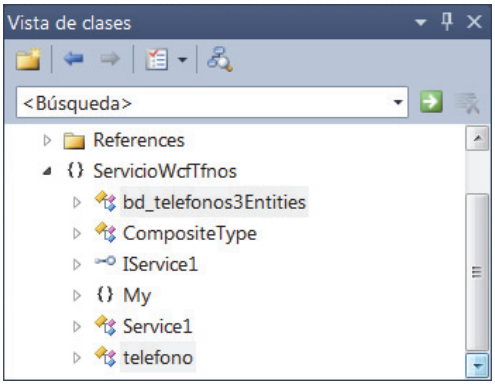
A continuación, creamos un servicio WCF. Para ello, añadimos a la solución anterior un nuevo proyecto de tipo *Aplicación de servicios WCF*: clic con el botón secundario del ratón sobre el nombre de la solución, *Agregar* > *Nuevo proyecto* > *WCF* > *Aplicación de servicios WCF*. Denomínelo *ServicioWcfTfnos*. Después haga clic en *Aceptar*. La solución tiene ahora tres proyectos, según puede verse en la figura siguiente:



Finalmente, compile la solución y verifique que el servicio puede ser accedido utilizando el explorador: clic con el botón secundario del ratón sobre el nombre del fichero *Service1.svc* > *Ver en el explorador*. Se mostrará una página en la que se puede ver el documento XML que describe el servicio en WSDL.



A continuación, abra el explorador de bases de datos/explorador de servidores y añada una conexión a la base de datos *bd_telefonos3*. Después, añada el modelo de entidades correspondiente a la base de datos. Para ello, abra el *asistente para el EDM* y agregue al proyecto *ServicioWcfTfnos* un nuevo elemento de tipo *ADO.NET Entity Data Model* denominado *Model.edmx* generado desde la base de datos *bd_telefonos3*. Se generará la clase del contexto de objetos *bd_telefonos3-Entities* y la clase de entidad *telefono*:



Como siguiente paso, definimos el contrato de servicio y el contrato de datos. El contrato de servicio declara las operaciones: obtener la lista de apellidos y teléfonos de la tabla *telefonos*, obtener la fila de la tabla *telefonos* correspondiente a un teléfono determinado, actualizar una fila, agregar una nueva fila y borrar una fila de la tabla *telefonos*. Para obtener la lista de apellidos y teléfonos el contrato de servicio utiliza en sus operaciones el tipo compuesto *Detalles*, según se muestra a continuación:

```
<ServiceContract()>
Public Interface IService1
  <OperationContract()>
  Function ObtenerTfnos() As List(Of Detalles)

  <OperationContract()>
  Function ObtenerTfno(tf As String) As telefono

  <OperationContract()>
  Sub ActualizarTfno(entidad As telefono)

  <OperationContract()>
  Sub AgregarTfno(entidad As telefono)

  <OperationContract()>
  Sub BorrarTfno(entidad As telefono)
End Interface

<DataContract()>
Public Class Detalles
  Private _apellido1 As String
  Private _telefono As String
  Public Sub New(ap As String, tf As String)
    _apellido1 = ap
    _telefono = tf
  End Sub

  <DataMember()>
  Public Property apellido1() As String
    Get
      Return _apellido1
    End Get
    Set(value As String)
      _apellido1 = value
    End Set
  End Property

  <DataMember()>
  Public Property telefono() As String
    Get
      Return _telefono
```

```
    End Get
    Set(value As String)
      _telefono = value
    End Set
  End Property
End Class
```

Después de definir el contrato de servicio y sus operaciones, y de definir el contrato de datos, definimos la clase del servicio. En nuestro caso, se trata de la clase *Service1* que implementa la interfaz *IService1*, pero antes de pasar a definir esta clase, vamos a escribir otra clase en la capa BLL, que utilizando *EF* y *LINQ to Entities* defina las operaciones de consulta y actualización que deseamos realizar sobre la base de datos. Entonces, como siguiente paso, añada una nueva clase al sitio web, denominada *telefonosBll*, y complétela como se indica a continuación (para organizar los componentes del sitio web, el autor añadirá esta clase en la carpeta *BLL*):

```
Public Class telefonosBll
  Public Sub New()
  End Sub

  Public Function ObtenerTfnoBll(tf As String) As telefono
    Using contexto As New bd_telefonos3Entities()
      Dim consulta = From tfno In contexto.telefonos
                     Where tfno.telefono1 = tf
                     Select tfno
      Dim registro As telefono = consulta.First()
      contexto.Detach(registro)
      Return registro
    End Using
  End Function

  Public Function ObtenerTfnosBll() As List(Of Detalles)
    Using contexto As New bd_telefonos3Entities()
      Dim consulta = From persona In contexto.telefonos
                     Select New With {persona.apellido1,
                                      persona.telefono1}
      Dim lista As New List(Of Detalles)()
      For Each e In consulta
        lista.Add(New Detalles(e.apellido1, e.telefono1))
      Next e
      Return lista
    End Using
  End Function

  Public Sub ActualizarTfnoBll(entidad As telefono)
    ' Obtener una entidad conectada con la clave especificada,
    ' aplicar los cambios y salvarlos en la base de datos
  End Sub
```

```
  Public Sub AgregarTfnoBll(entidad As telefono)
    ' Si no hay una entidad con el mismo teléfono, añadirla
    ' al contexto actual y salvar los cambios
  End Sub

  Public Sub BorrarTfnoBll(entidad As telefono)
    ' Obtener la entidad conectada correspondiente, borrarla
    ' del contexto actual y salvar los cambios
  End Sub

  Private Function ExisteTelefono(entidad As telefono,
  contexto As bd_telefonos3Entities) As telefono
    Try
      Dim consulta = From tfno In contexto.telefonos
                     Where tfno.telefono1 = entidad.telefono1
                     Select tfno
      Dim registro As telefono = consulta.First()
      Return registro
    Catch e1 As Exception
      Return Nothing
    End Try
  End Function
End Class
```

Echemos un vistazo a los métodos de la clase *telefonosBll*. El método *ObtenerTfnoBll* utilizando LINQ devuelve una entidad *telefono* (la correspondiente al teléfono pasado como argumento), *ObtenerTfnosBll* utilizando LINQ devuelve una lista con el primer apellido y el teléfono de todas las filas de la tabla *telefonos*, *ActualizarTfnoBll* permite actualizar una fila de la tabla *telefonos* (la correspondiente a la entidad pasada como argumento), *AgregarTfnoBll* permite agregar a la tabla *telefonos* una nueva fila, la correspondiente a la entidad pasada como argumento, y *BorrarTfnoBll* permite borrar la fila especificada de la tabla *telefonos*. Lo importante a tener en cuenta es que: estamos devolviendo a las capas superiores (servicio y capa de presentación) y recibiendo de ellas conjuntos de datos desconectados o desasociados. Esto significa, con respecto a *EF*, que estamos trabajando en un contexto (objeto **ObjectContext**) de "corta duración". Entonces, cuando finalicemos el ciclo (se envió al nivel de presentación una entidad *telefono*, se modificó y se devolvió al nivel de servidor), tendremos que volver a conectar la entidad devuelta por el cliente con estado desconectada (*Detached*) que se actualizó al margen del **ObjectContext**. Éste es un escenario muy común cuando estamos utilizando arquitecturas de N-capas y N-niveles.

Resumiendo, al seriar una entidad, por ejemplo un objeto *telefono*, para enviarla a través de la red al nivel de cliente, esa entidad se desconecta de su contexto de datos, lo que impide realizar un seguimiento de los cambios en sus datos o en sus relaciones con otros objetos. Esto no presenta ningún problema mientras el

cliente sólo esté leyendo los datos, pero sí cuando se trate de actualizar esos datos en la base de datos; en este caso, antes de llamar al método **SaveChanges** del **ObjectContext** actual habrá que asociar la entidad a este contexto de datos.

Volviendo a nuestro código, las operaciones que va a realizar el servicio WCF serán seriar y devolver una o más entidades desconectadas, actualizar la base de datos con los datos de una entidad, añadir nuevas entidades y borrar una o más entidades. Según lo expuesto, la clase que define el contrato de servicio puede ser la siguiente:

```
Public Class Service1
  Implements IService1
  Public Function ObtenerTfno(tf As String) As telefono _
    Implements IService1.ObtenerTfno
    Return New telefonosBll().ObtenerTfnoBll(tf)
  End Function

  Public Function ObtenerTfnos() As List(Of Detalles) _
    Implements IService1.ObtenerTfnos
    Return New telefonosBll().ObtenerTfnosBll()
  End Function

  Public Sub ActualizarTfno(entidad As telefono) _
    Implements IService1.ActualizarTfno
    CType(New telefonosBll(), telefonosBll).ActualizarTfnoBll(entidad)
  End Sub

  Public Sub AgregarTfno(entidad As telefono) _
    Implements IService1.AgregarTfno
    CType(New telefonosBll(), telefonosBll).AgregarTfnoBll(entidad)
  End Sub
  Public Sub BorrarTfno(entidad As telefono) _
    Implements IService1.BorrarTfno
    CType(New telefonosBll(), telefonosBll).BorrarTfnoBll(entidad)
  End Sub
End Class
```

Obsérvese que los métodos de esta clase se limitan a invocar a sus homólogos en la capa BLL.

Posteriormente, una aplicación cliente (la aplicación Silverlight) consumirá el servicio WCF y mostrará los datos de una entidad *telefono*, la de aquélla que el usuario seleccione de la lista. A la vista de los datos, el usuario podrá cambiar algunos de ellos o todos con el fin de actualizar esa entidad o agregar una nueva, enviándola de nuevo al servidor para actualizar la base de datos, o bien borrar esa entidad. No olvide que la aplicación cliente estará trabajando con una entidad des-

conectada, por lo que no hay un seguimiento de los cambios que se realicen sobre ella o sobre sus relaciones con otros objetos.

Para enviar la entidad modificada al nivel de servidor, el cliente invocará al método *ActualizarTfno* del servicio WCF pasando como argumento dicha entidad, que a su vez invoca al método de negocio *ActualizarTfnoBll* de la capa BLL que tiene como fin conectar de nuevo la entidad recibida al contexto de objetos actual (de esto se encargará el método *VincularObjeto*) e invocar al método **SaveChanges** de este contexto para almacenar los cambios en la base de datos, ya que **SaveChanges** sólo trabaja sobre las entidades asociadas al contexto que lo invoca, porque el seguimiento que éste realiza sobre ellas le permite conocer los cambios que hay que registrar.

```
Public Sub ActualizarTfnoBll(entidad As telefono)
  Using contexto As New bd_telefonos3Entities()
    contexto.VincularObjeto(entidad) ' método extensor
    contexto.SaveChanges()
  End Using
End Sub
```

Pero, ¿por qué no utilizamos para conectar la entidad el método **Attach** de **ObjectContext** en lugar del método personalizado *VincularObjeto*? Porque si lo prueba no funciona, debido a que la entidad que fue desconectada y enviada al cliente, ahora es nueva para el contexto y por lo tanto no sabe nada acerca de ella ya que no realizó el seguimiento de los cambios.

También hay que tener en cuenta que cuando *Entity Framework* vaya a grabar los datos en la base de datos utilizará los valores originales de la entidad para realizar las comprobaciones de simultaneidad optimista. Entonces, el método *VincularObjeto*, para conectar la entidad al contexto, tendrá que:

1. Obtener una entidad nueva con los valores reales, de clave la de la entidad desconectada (método **TryGetObjectByKey**). Dicha entidad, por el hecho de crearla en el contexto actual, estará conectada al mismo.

2. Aplicar a la entidad nueva los cambios registrados en la entidad desconectada (método **ApplyPropertyChanges**).

3. A continuación, invocando el método **GetAllRelatedEnds** del administrador de relaciones de la entidad desconectada, se obtienen todos los extremos que definen las relaciones con otras entidades (objetos **EntityReference**) de ésta y se establecen en la entidad nueva (**GetRelatedEnd**).

El proceso descrito es la parte más complicada de este apartado ya que requiere un conocimiento de la estructura de las entidades y de cómo se realiza el seguimiento de las relaciones entre los objetos en el contexto de objetos.

Observe en *ActualizarTfnoBll* la llamada al método *VincularObjeto*; se trata de un método del contexto de la clase *bd_telefonos3Entities* derivada de **ObjectContext**. Esto significa que lo tenemos que implementar como un método extensor de la clase *bd_telefonos3Entities*. Para ello, añada en *ServicioWcfTfnos\BLL* un nuevo módulo (**Module**) denominado *bd_telefonos3EntitiesEx* y complételo según lo expuesto. La solución puede verla en el CD que acompaña al libro, en la carpeta *Cap10\ApSilverlightTfnos\ServicioWcfTfnos\BLL*.

Para enviar una nueva entidad al nivel de servidor, el cliente invocará al método *AgregarTfno* del servicio WCF pasando como argumento dicha entidad, que a su vez invoca al método de negocio *AgregarTfnoBll* de la capa BLL que tiene como fin verificar que no hay ya una entidad con el mismo teléfono, añadir la entidad al contexto actual y llamar a **SaveChanges** para almacenar los cambios en la base de datos.

```
Public Sub AgregarTfnoBll(entidad As telefono)
  Using contexto As New bd_telefonos3Entities()
    If ExisteTelefono(entidad, contexto) IsNot Nothing Then
      Throw New InvalidOperationException("El teléfono existe")
    End If
    contexto.telefonos.AddObject(entidad)
    contexto.SaveChanges()
  End Using
End Sub
```

Para enviar al nivel de servidor la entidad a eliminar, el cliente invocará al método *BorrarTfno* del servicio WCF pasando como argumento dicha entidad, que a su vez invoca al método de negocio *BorrarTfnoBll* de la capa BLL que tiene como fin obtener la entidad conectada correspondiente, borrarla del contexto actual y llamar a **SaveChanges** para almacenar los cambios en la base de datos.

```
Public Sub BorrarTfnoBll(entidad As telefono)
  Using contexto As New bd_telefonos3Entities()
    Try
      entidad = ExisteTelefono(entidad, contexto)
      If entidad Is Nothing Then
        Throw New InvalidOperationException("El teléfono no existe")
      End If
      contexto.DeleteObject(entidad)
      contexto.SaveChanges()
    Catch ex As OptimisticConcurrencyException
      Throw New InvalidOperationException(
        "El elemento especificado no puede ser borrado")
```

```
    End Try
  End Using
End Sub
```

Finalmente, según explicamos en el apartado *Comunicación entre dominios* al principio de este capítulo, para que la aplicación Silverlight que vamos a diseñar a continuación pueda llamar a un servicio web de otro dominio, el servicio web tiene que usar un fichero de directivas entre dominios *ClientAccessPolicy.xml*.

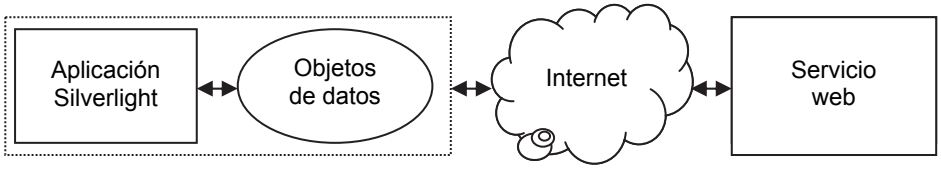
Según esto, vamos a añadir al proyecto *ServicioWcfTfnos* que implementa nuestro servicio web el fichero de directivas *ClientAccessPolicy.xml* siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
  <access-policy>
    <cross-domain-access>
      <policy>
        <allow-from http-request-headers="*">
          <domain uri="*"/>
        </allow-from>
        <grant-to>
          <resource path="/" include-subpaths="true"/>
        </grant-to>
      </policy>
    </cross-domain-access>
  </access-policy>
```

De esta forma, cuando la aplicación Silverlight *ApSilverlightTfnos.xap* detecte la solicitudes entre dominios por el hecho de llamar al servicio web, buscará en la raíz de la aplicación *ServicioWcfTfnos* el fichero de directivas *ClientAccessPolicy.xml* que le permita el acceso a dicho servicio.

Cliente Silverlight

Una vez finalizado el nivel de servidor, el problema que tenemos que resolver ahora es construir un cliente Silverlight que consuma el servicio web que acabamos de construir.



La estructura básica de la aplicación Silverlight que accederá al servicio ya la tenemos construida: proyecto *ApSilverlightTfnos*.

A continuación diseñe la página según la figura mostrada a continuación. La página incluye una rejilla de ocho filas por cuatro columnas con los elementos siguientes:

- Un control **ComboBox** que tendrá como origen de datos la colección de objetos *Detalles* devuelta por el método *ObtenerTfnos* de la interfaz del servicio WCF. Esta lista desplegable mostrará el primer apellido de todas las filas de la tabla *telefonos* de la base de datos. Recuerde que un objeto *Detalles* almacena ese apellido y el número de teléfono correspondiente.

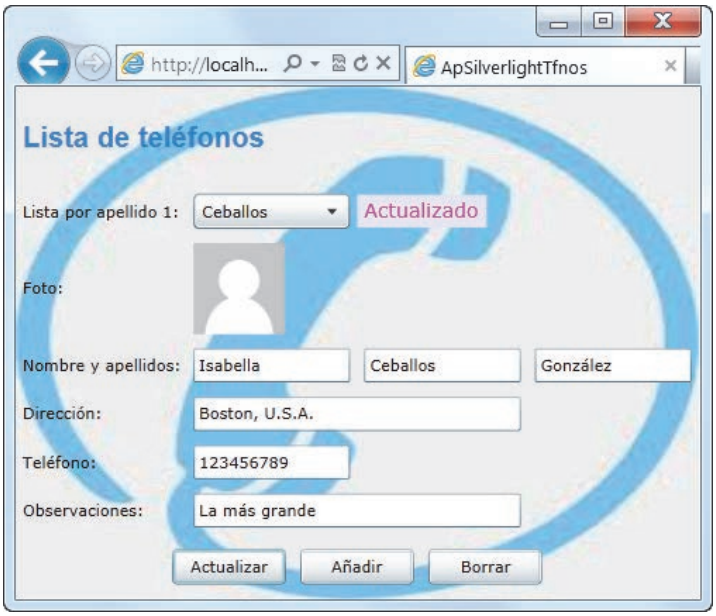
- Un conjunto de cajas de texto, con las etiquetas correspondientes, para mostrar los datos de la fila correspondiente al elemento seleccionado en la lista. Esto implica establecer el contexto de datos del contenedor (el **Grid**) con el objeto *telefono* correspondiente al elemento seleccionado de la lista.

- El usuario podrá modificar los datos de las cajas con la finalidad de registrar los cambios en la base de datos. Para enviar estos cambios, añada al formulario un botón *Actualizar*.

- También podrá modificar los datos de las cajas con la finalidad de añadir una nueva fila a la tabla *telefonos* de la base de datos. Para enviar los datos de la nueva fila, añada al formulario un botón *Añadir*.

- Y también podrá borrar la fila de la base de datos correspondiente a los datos mostrados. Para enviar los datos de la fila a borrar, añada al formulario un botón *Borrar*.

- Otros controles añadidos son: la etiqueta que muestra el título del formulario, un **StackPanel** que actúa como contenedor de los botones y un control **Popup** (a la derecha de la lista desplegable) que contiene un **Border** que, a su vez, contiene un **TextBlock** para mostrar mensajes acerca de la última operación realizada o mensajes de error. El **Popup** desaparecerá al hacer clic sobre él o sobre la página.



- También hemos añadido al objeto *App* un diccionario de recursos que define los estilos utilizados en el diseño de la interfaz gráfica del usuario.

```
<Application
  ...
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Recursos/Styles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

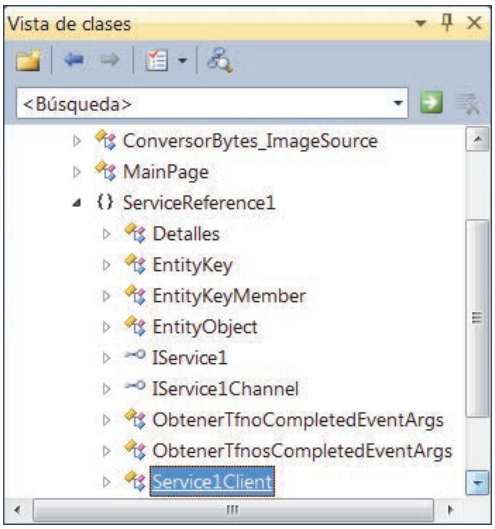
Pensemos ahora en cómo se sucederán los hechos cuando un usuario solicite desde un explorador que se ejecute nuestra aplicación Silverlight:

```
http://localhost/ApSilverlightTfnos.Web/ApSilverlightTfnosTestPage.aspx
```

1. Se visualiza el formulario web.
2. El usuario selecciona una persona de la lista.
3. Las cajas de texto muestran los datos correspondientes a esa persona.
4. El usuario quizás haya solicitado esos datos como simple información o porque desea realizar alguna actualización o borrar el registro correspondiente. También puede optar por añadir un registro nuevo.

5. En el caso de realizar una actualización, modificará en las cajas de texto los datos implicados y hará clic en el botón *Actualizar*. Si lo que desea es borrar el registro mostrado, hará clic en el botón *Borrar* y si lo que quiere es añadir un nuevo registro, introducirá los datos en las cajas y hará clic en el botón *Añadir*.

Para llevar a cabo los puntos anteriores, el cliente debe agregar una referencia al servicio WCF *ServicioWcfTfnos*. Para ello, diríjase al explorador de soluciones y haga clic con el botón secundario del ratón sobre el nombre del proyecto *ApSilverlightTfnos* y seleccione la opción *Agregar referencia de servicio*. Después, por medio del diálogo que se visualiza, localice el servicio WCF, asigne un nombre para el espacio de nombres del *proxy* que se generará, en el ejemplo *ServiceReference1*, y haga clic en el botón *Aceptar*. La figura siguiente nuestra el espacio de nombres generado y los elementos que incluye, como *Service1Client*:



Una vez agregada en el cliente la referencia al servicio, el siguiente paso es crear un objeto de la clase *Service1Client* del *proxy* para acceder a los métodos del servicio. Según esto, edite el fichero de código subyacente *MainPage.xaml.vb* y agregue el código sombreado que se indica a continuación, lo que implica añadir el espacio de nombres *ApSilverlightTfnos.ServiceReference1*.

```
Imports ApSilverlightTfnos.ServiceReference1

Partial Public Class MainPage
  Inherits UserControl
  Private cliente As Service1Client = Nothing

  Public Sub New()
```

```
    InitializeComponent()
    cliente = New Service1Client()
  End Sub
End Class
```

Recuerde que todas las llamadas que se realicen al servicio WCF desde Silverlight son *asíncronas*. Por ejemplo, consideremos la operación del servicio *ObtenerTfnos*. Realizar esta operación implica llamar al método *ObtenerTfnosAsync*. Una vez realizada la llamada al método asíncrono, el control se devuelve a la aplicación Silverlight para que el usuario pueda seguir interactuando con la misma. Cuando el servicio devuelva los datos que solicitamos, lo notificará a la aplicación generando el evento *ObtenerTfnosCompleted*. Entonces, tenemos que interceptar este evento para poder recoger el resultado. Ídem para el resto de las operaciones.

```
Public Sub New()
  InitializeComponent()
  cliente = New Service1Client()

  AddHandler cliente.ObtenerTfnoCompleted,
    AddressOf cliente_ObtenerTfnoCompleted
  AddHandler cliente.ObtenerTfnosCompleted,
    AddressOf cliente_ObtenerTfnosCompleted
  AddHandler cliente.ActualizarTfnoCompleted,
    AddressOf cliente_ActualizarTfnoCompleted
  AddHandler cliente.AgregarTfnoCompleted,
    AddressOf cliente_AgregarTfnoCompleted
  AddHandler cliente.BorrarTfnoCompleted,
    AddressOf cliente_BorrarTfnoCompleted
End Sub
```

Cada uno de los controladores de eventos anteriores indica a qué método se debe llamar cuando el servicio devuelva los datos solicitados. Por lo tanto, defina estos métodos en el ámbito de la clase *MainPage*:

```
Private Sub cliente_ObtenerTfnosCompleted(sender As Object,
e As ObtenerTfnosCompletedEventArgs)
  ' ...
End Sub

Private Sub cliente_ObtenerTfnoCompleted(sender As Object,
e As ObtenerTfnoCompletedEventArgs)
  ' ...
End Sub

Private Sub cliente_ActualizarTfnoCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  ' ...
End Sub
```

```
Private Sub cliente_AgregarTfnoCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  ' ...
End Sub

Private Sub cliente_BorrarTfnoCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  ' ...
End Sub
```

Llenar la lista

A continuación, configure la lista para que muestre el atributo *apellido1* de la colección de objetos *Detalles* que vincularemos con la misma:

```
<ComboBox Name="lstPersonas" Grid.Column="1" Grid.Row="1"
          Style="{StaticResource EstiloComboBox}"
          SelectionChanged="lstPersonas_SelectionChanged">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding apellido1}" />
      </StackPanel>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

El enlace anterior requiere asignar a la propiedad **ItemsSource** del **ComboBox**, *lstPersonas*, la colección de objetos devuelta por el método *ObtenerTfnosAsync*. Esta colección la podemos obtener cuando se cargue la página. Añada, por lo tanto, el controlador del evento **Loaded** de la página y complételo para que invoque a ese método:

```
Private Sub UserControl_Loaded(sender As Object, e As RoutedEventArgs)
  cliente.ObtenerTfnosAsync()
End Sub
```

Cuando la operación anterior se complete, se producirá el evento *ObtenerTfnosCompleted*, instante en el que podemos disponer de la colección de objetos de la clase *Detalles*, devuelta por medio de la propiedad *e.Result*, que asignaremos a la propiedad **ItemSource** de la lista *lstPersonas*:

```
Private Sub cliente_ObtenerTfnosCompleted(sender As Object,
e As ObtenerTfnosCompletedEventArgs)
  lstPersonas.ItemsSource = e.Result
  lstPersonas.SelectedIndex = 0
End Sub
```

Si ejecuta ahora la aplicación, observará que la lista muestra el primer apellido de todas las filas de la tabla *telefonos* de la base de datos.

El parámetro *e* de la clase *ObtenerTfnosCompletedEventArgs*, derivada de **AsyncCompletedEventArgs**, recibirá información sobre el resultado de la operación asíncrona ejecutada, a través de sus propiedades. Por ejemplo, hemos visto que **Result** hace referencia al resultado obtenido una vez ejecutada la tarea asíncrona; si necesitamos determinar si se canceló la tarea asíncrona, disponemos de la propiedad **Cancelled** y para determinar si se produjo una excepción durante la ejecución de la tarea asíncrona, disponemos de la propiedad **Error**.

Según lo expuesto, vamos a modificar el método anterior para verificar si se produce alguna excepción durante la ejecución de la tarea asíncrona:

```
Private Sub cliente_ObtenerTfnosCompleted(sender As Object,
e As ObtenerTfnosCompletedEventArgs)
  If e.Error IsNot Nothing Then
    tbkInfo.Text = "Error: " & e.Error.Message.ToString()
    popInfo.IsOpen = True
  Else
    lstPersonas.ItemsSource = e.Result
    lstPersonas.SelectedIndex = 0
  End If
End Sub
```

Mensajes para el usuario

Para mostrar información al usuario, como los mensajes de error, si recuerda, añadimos a la interfaz gráfica un control **Popup**, con un **Border** con un **TextBlock** como contenido. Este **TextBlock** será el contenedor de los mensajes.

```
<Popup x:Name="popInfo" Grid.Column="2" Grid.Row="1"
       Grid.ColumnSpan="2" VerticalOffset="6">
  <Border MouseLeftButtonDown="popInfo_MouseLeftButtonDown"
          Background="Lavender">
    <TextBlock Name="tbkInfo"
               Style="{StaticResource EstiloTextBlock}" />
  </Border>
</Popup>
```

El control **Popup** fue expuesto en el apartado *Ventanas y cajas de diálogo* del capítulo *Silverlight*. Allí dijimos que para mostrarlo hay que asignar explícitamente a su propiedad **IsOpen** el valor **True**, operación que hacemos si se produce un error, según muestra el código Visual Basic anterior, y para ocultarlo, el valor **False**, operación que realizaremos desde el método que responda al evento **MouseLeftButtonDown** del **Border** y de la página *MainPage*. Por lo tanto, añada pa-

ra estos elementos los controladores de ese evento y complete el método que se ejecutará según se muestra a continuación:

```
Private Sub popInfo_MouseLeftButtonDown(sender As Object,
e As MouseButtonEventArgs)
  popInfo.IsOpen = False
End Sub
```

Ahora, cuando se muestre la ventana **Popup** porque se produjo un error, para ocultarla el usuario, simplemente, hará clic en dicha ventana o en la página.

Ordenar la lista

Continuando con el desarrollo de la aplicación, si observamos el contenido de la lista *lstPersonas* vemos que no está ordenado, lo cual no facilita al usuario la elección de cualquier elemento de la misma. Por lo tanto, vamos a ordenar ascendentemente los elementos de la colección por el atributo *apellido1*, para lo cual utilizaremos la clase **PagedCollectionView** del espacio de nombres **System.Windows.Data**. Esta clase, según vimos en el capítulo *Silverlight*, representa una vista para agrupar, ordenar, filtrar y navegar por una colección de datos que implemente la interfaz **IEnumerable**. Según esto, creamos la vista de la colección devuelta por *e.Result*, ordenamos sus elementos, nos posicionamos en el primero y la asignamos a la propiedad **ItemSource** de la lista *lstPersonas*:

```
Private Sub cliente_ObtenerTfnosCompleted(sender As Object,
e As ObtenerTfnosCompletedEventArgs)
  ...
  Else
    vistaListaDetalles = New PagedCollectionView(e.Result)
    vistaListaDetalles.SortDescriptions.Add(
      New SortDescription("apellido1",
                          ListSortDirection.Ascending))
    vistaListaDetalles.MoveCurrentToFirst()
    lstPersonas.ItemsSource = vistaListaDetalles
  End If
End Sub
```

La propiedad **SortDescriptions** de la vista describe cómo serán ordenados los elementos de la colección en la vista.

Defina la referencia *vistaListaDetalles* de tipo **PagedCollectionView** como atributo privado de la clase *MainPage*:

```
Partial Public Class MainPage
  Inherits UserControl
  Private cliente As Service1Client = Nothing
  Private vistaListaDetalles As PagedCollectionView = Nothing
```

```
  ...
End Class
```

Mostrar datos

El hecho de situarnos inicialmente en la primera posición de la lista hace que la propiedad **SelectedIndex** tome el valor 0 y que, por lo tanto, se genere el evento **SelectionChanged** de la misma, evento que podemos aprovechar para mostrar los datos de la fila de la tabla *telefonos* correspondiente al elemento seleccionado. Esta operación requiere llamar al método *ObtenerTfnoAsync* del *proxy*, pasando como argumento el número de teléfono del elemento seleccionado, y responder al evento *ObtenerTfnoCompleted* para obtener el objeto *telefono* correspondiente, que asignaremos al contexto de datos del **Grid**.

```
Private Sub lstPersonas_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  Dim lista As ComboBox = TryCast(sender, ComboBox)
  If lista.SelectedIndex < 0 Then Return
  Dim elementoSeleccionado As Detalles =
    TryCast(lista.SelectedItem, Detalles)
  If elementoSeleccionado.telefono IsNot Nothing Then
    cliente.ObtenerTfnoAsync(elementoSeleccionado.telefono)
  End If
End Sub

Private Sub cliente_ObtenerTfnoCompleted(sender As Object,
e As ObtenerTfnoCompletedEventArgs)
  If e.Error IsNot Nothing Then
    tbkInfo.Text = "Error: el teléfono no existe"
    popInfo.IsOpen = True
  Else
    objTelefonoSeleccionado = e.Result
    ' Establecer el contexto de datos del Grid
    LayoutRoot.DataContext = objTelefonoSeleccionado
    ' Poner una foto ficticia si este campo es Nothing
    If objTelefonoSeleccionado.foto IsNot Nothing Then Return
    Dim wbm As WriteableBitmap =
      New WriteableBitmap(0, 0).FromResource("Recursos/foto.jpg")
    imFoto.Source = wbm
  End If
End Sub
```

Defina la referencia *objTelefonoSeleccionado* de tipo *telefono* como atributo privado de la clase *MainPage*:

```
Partial Public Class MainPage
  Inherits UserControl
  Private cliente As Service1Client = Nothing
```

```
  Private vistaListaDetalles As PagedCollectionView = Nothing
  Private objTelefonoSeleccionado As telefono = Nothing
  ...
End Class
```

Observe que, además de asignar el objeto *telefono*, correspondiente al elemento seleccionado de la lista, al contexto de datos del **Grid**, hemos puesto una foto ficticia en el control *imFoto* de la interfaz gráfica, sólo en el caso de que el campo *foto* del objeto *telefono* obtenido sea *Nothing*. Posteriormente, veremos cómo cambiar esta foto desde el controlador del evento *MouseLeftButtonDown* de este control.

El trabajo con imágenes fue ampliamente explicado en el apartado *Trabajar con imágenes* del capítulo *Silverlight*, por lo que en este capítulo no daremos más explicaciones, simplemente nos limitaremos a utilizar lo allí descrito. Recuerde que allí utilizamos la biblioteca *WriteableBitmapEx.dll* y tuvimos que añadir un conversor de bytes a imagen y viceversa que implementamos bajo la clase *ConversorBytes_ImageSource*. Estos recursos, así como las imágenes utilizadas para la foto y para el fondo de la página, han sido añadidos en una carpeta *Recursos* de la aplicación Silverlight.

Para que los controles de la página *MainPage* muestren los datos referenciados por las propiedades *foto*, *nombre*, *apellido1*, *apellido2*, *direccion*, *telefono1* y *observaciones* del objeto *telefono* asignado al contexto de datos del **Grid**, hemos enlazado cada una de esas propiedades con la propiedad adecuada de sus respectivos controles en la página, y el modo del enlace lo hemos establecido en las dos direcciones:

```
<Image Name="imFoto" Grid.Column="1" Grid.Row="2"
       Style="{StaticResource EstiloImagen}"
       Source="{Binding Path=foto, Mode=TwoWay,
       Converter={StaticResource conversorBytes_ImageSource}}"
       MouseLeftButtonDown="imFoto_MouseLeftButtonDown" />
<TextBox Name="ctNombre" Grid.Column="1" Grid.Row="3"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=nombre, Mode=TwoWay}" />
<TextBox Name="ctApellido1" Grid.Column="2" Grid.Row="3"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=apellido1, Mode=TwoWay}" />
<TextBox Name="ctApellido2" Grid.Column="3" Grid.Row="3"
         Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=apellido2, Mode=TwoWay}" />
<TextBox Name="ctDireccion" Grid.Column="1" Grid.Row="4"
         Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
         Text="{Binding Path=direccion, Mode=TwoWay}"/>
<TextBox Name="ctTelefono" Grid.Column="1" Grid.Row="5"
         Style="{StaticResource EstiloTextBox}"
```

```
          Text="{Binding Path=telefono1, Mode=TwoWay}"/>
<TextBox Name="ctObservaciones" Grid.Column="1" Grid.Row="6"
          Grid.ColumnSpan="2" Style="{StaticResource EstiloTextBox}"
          Text="{Binding Path=observaciones, Mode=TwoWay}"/>
```

Si ejecuta ahora la aplicación, observará cómo la ventana muestra ya los datos del objeto *telefono* correspondiente al elemento seleccionado.

Actualizar datos

Cuando el usuario quiera modificar los datos, simplemente hará esas modificaciones sobre las cajas de texto, o cambiará la foto, y hará clic en el botón *Actualizar* para enviar los cambios a la base de datos. Dese cuenta de que el objeto *telefono* que actualmente muestra la página, según vimos anteriormente en el apartado *Mostrar datos*, está referenciado por *objTelefonoSeleccionado* y vinculado con el contexto de datos del **Grid**, con lo que todas las actualizaciones realizadas sobre las cajas se reflejan automáticamente en ese objeto, que será el que se envíe para realizar la actualización sobre la base de datos.

Esta operación requiere llamar al método *ActualizarTfnoAsync* del *proxy* pasando como argumento el objeto *objTelefonoSeleccionado*, llamada que realizaremos desde el método que responda al evento **Click** del botón *Actualizar*, y responder al evento *ActualizarTfnoCompleted* para, una vez que la operación de actualización haya sido realizada con éxito, actualizar también los datos correspondientes en la lista desplegable.

```
Private Sub btActualizar_Click(sender As Object, e As RoutedEventArgs)
  cliente.ActualizarTfnoAsync(objTelefonoSeleccionado)
End Sub

Private Sub cliente_ActualizarTfnoCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  If e.Error IsNot Nothing Then
    tbkInfo.Text = "Error: " & e.Error.Message.ToString()
    popInfo.IsOpen = True
  Else
    ' Actualizar los datos del ComboBox
    Dim objDetallesSeleccionado As Detalles =
      TryCast(lstPersonas.SelectedItem, Detalles)
    objDetallesSeleccionado.apellido1 =
      objTelefonoSeleccionado.apellido1
    objDetallesSeleccionado.telefono =
      objTelefonoSeleccionado.telefono1
    tbkInfo.Text = "Actualizado"
    popInfo.IsOpen = True
  End If
End Sub
```

Actualizar la foto

Para actualizar la foto, el usuario hará clic sobre la misma, lo que hará que se genere el evento **MouseLeftButtonDown**. Como respuesta a este evento, se mostrará el diálogo *Abrir* de la clase **OpenFileDialog** para que el usuario pueda seleccionar de su sistema de ficheros la nueva foto, la cual será asignada a la propiedad **Source** del control *imFoto*. Según esto, añada el controlador para este evento y complete el método correspondiente así:

```
Private Sub imFoto_MouseLeftButtonDown(sender As Object,
e As MouseButtonEventArgs)
  Dim dlg As New OpenFileDialog()
  dlg.Filter = "ficheros jpg (*.jpg)|*.jpg|" &
               "ficheros jpeg (*.jpeg)|*.jpeg|" &
               "ficheros png (*.png)|*.png|" &
               "Todos (*.*)|*.*"

  If dlg.ShowDialog() = True Then
    If dlg.File.Name Is Nothing OrElse dlg.File.Name.Length = 0 Then
      Return
    End If
    Dim imagen As New WriteableBitmap(0, 0)
    imagen.SetSource(dlg.File.OpenRead())
    imFoto.Source = imagen
  End If
End Sub
```

Para más detalles, repase el apartado *Trabajar con imágenes* del capítulo titulado *Silverlight*.

Agregar datos

Para agregar un nuevo objeto *telefono* (una nueva fila a la tabla *telefonos*) el usuario, simplemente, editará cada uno de los campos mostrados por la interfaz gráfica y hará clic en el botón *Añadir* para enviar los cambios a la base de datos. Igual que dijimos en el apartado *Actualizar datos*, el objeto *telefono* que actualmente muestra la página, referenciado por *objTelefonoSeleccionado*, está vinculado con el contexto de datos del **Grid**, con lo que todas las actualizaciones realizadas sobre las cajas se reflejan automáticamente en ese objeto, que será el que se envíe para realizar la actualización sobre la base de datos.

Esta operación requiere llamar al método *AgregarTfnoAsync* del *proxy* pasando como argumento el objeto *objTelefonoSeleccionado*, llamada que realizaremos desde el método que responda al evento **Click** del botón *Añadir*, y responder al evento *AgregarTfnoCompleted* para, una vez que la operación haya sido realizada con éxito, agregar también el objeto *Detalles* correspondiente a la vista asociada

con la lista desplegable. Puesto que la vista está ordenada ascendentemente, este objeto será insertado automáticamente en el lugar que le corresponde.

Para añadir un nuevo objeto a la vista hay que invocar a su método **AddNew**. Se añadirá un objeto vacío referenciado por la propiedad **CurrentAddItem** de la vista. Complete a continuación las propiedades del nuevo objeto con los valores adecuados y, a continuación, invoque al método **CommitNew** de la vista para finalizar las operaciones programadas; en nuestro caso, la ordenación.

```
Private Sub btAgregar_Click(sender As Object, e As RoutedEventArgs)
  cliente.AgregarTfnoAsync(objTelefonoSeleccionado)
End Sub

Private Sub cliente_AgregarTfnoCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  If e.Error IsNot Nothing Then
    tbkInfo.Text = "Error: ¿existe el teléfono?"
    popInfo.IsOpen = True
  Else
    ' Añadir un nuevo elemento a la vista
    vistaListaDetalles.AddNew()
    Dim objDetallesAgregado As Detalles =
      TryCast(vistaListaDetalles.CurrentAddItem, Detalles)
    objDetallesAgregado.apellido1 =
      objTelefonoSeleccionado.apellido1
    objDetallesAgregado.telefono =
      objTelefonoSeleccionado.telefono1
    vistaListaDetalles.CommitNew()

    tbkInfo.Text = "Añadido"
    popInfo.IsOpen = True
  End If
End Sub
```

Borrar datos

Para borrar un objeto *telefono* (una fila de la tabla *telefonos*) el usuario seleccionará el elemento correspondiente en la lista desplegable y hará clic en el botón *Borrar* para ejecutar dicha operación sobre la base de datos.

Esta operación requiere llamar al método *BorrarTfnoAsync* del *proxy* pasando como argumento el objeto *objTelefonoSeleccionado* correspondiente a la selección, llamada que realizaremos desde el método que responda al evento **Click** del botón *Borrar*, y responder al evento *BorrarTfnoCompleted* para, una vez que la operación haya sido realizada con éxito, borrar también el objeto *Detalles* correspondiente de la vista asociada con la lista desplegable.

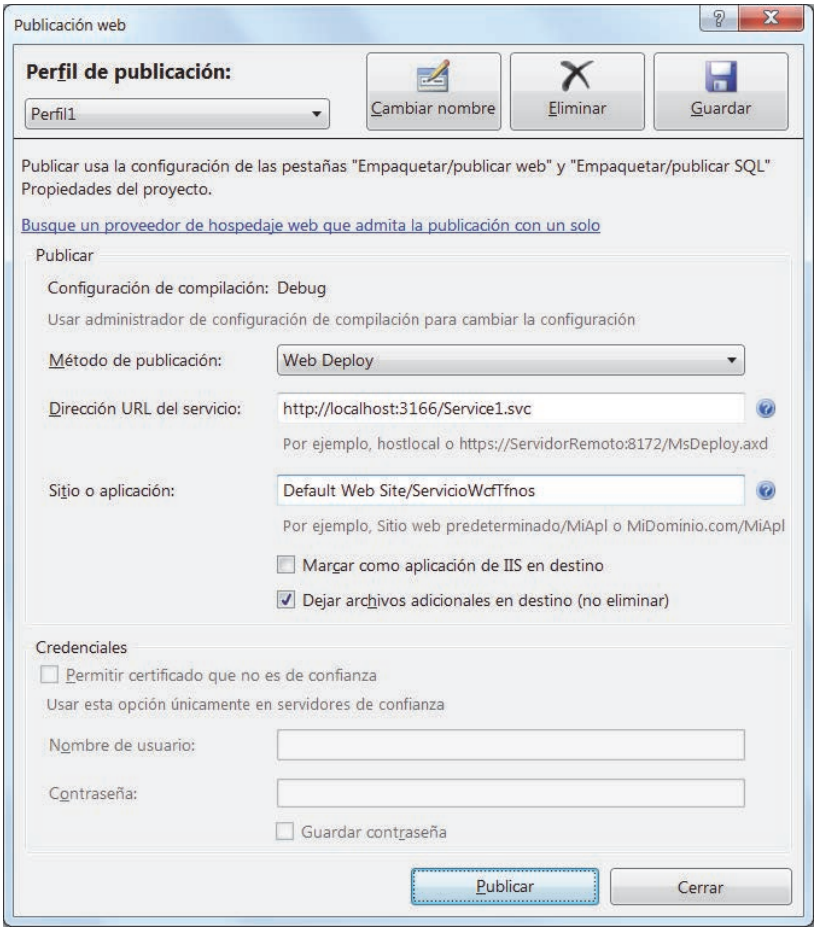
```
Private Sub btBorrar_Click(sender As Object, e As RoutedEventArgs)
  cliente.BorrarTfnoAsync(objTelefonoSeleccionado)
End Sub

Private Sub cliente_BorrarTfnoCompleted(sender As Object, e As
AsyncCompletedEventArgs)
  If e.Error IsNot Nothing Then
    tbkInfo.Text = "Error: ¿existe el teléfono?"
    popInfo.IsOpen = True
  Else
    ' Actualizar los datos del ComboBox
    vistaListaDetalles.Remove(vistaListaDetalles.CurrentItem)
    vistaListaDetalles.MoveCurrentToFirst()
    tbkInfo.Text = "Borrado"
    popInfo.IsOpen = True
  End If
End Sub
```

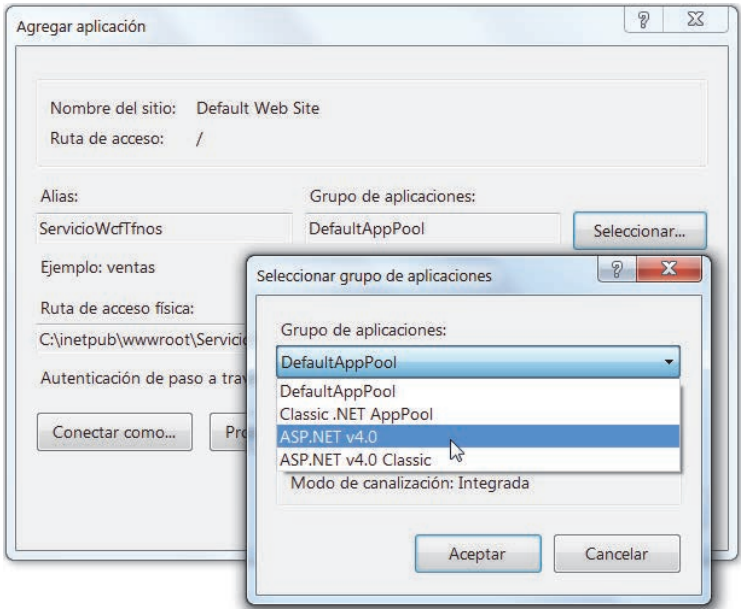
Publicar el servicio WCF y la aplicación Silverlight

La aplicación Silverlight que acabamos de construir y que queremos publicar en un servidor IIS accede a un servicio web que no pertenece al mismo dominio que la aplicación. Por lo tanto, publicaremos primero el servicio web, después actualizaremos la referencia del servicio web de la aplicación Silverlight para que apunte a esta nueva URL y, finalmente, publicaremos la aplicación Silverlight.

Para publicar el servicio WCF se necesita conocer la dirección URL actual del servicio y la dirección del sitio donde se va a publicar. Conocidos estos datos, y según explicamos anteriormente en este mismo capítulo, vamos a realizar esta tarea utilizando el asistente para publicación de sitios webs: haga clic con el botón secundario del ratón sobre el nombre del proyecto del servicio web (*ServicioWcfTfnos*) y elija la opción *Publicar* (tiene que haber iniciado Visual Studio como administrador). Después complete el formulario análogamente a como indica la figura siguiente:



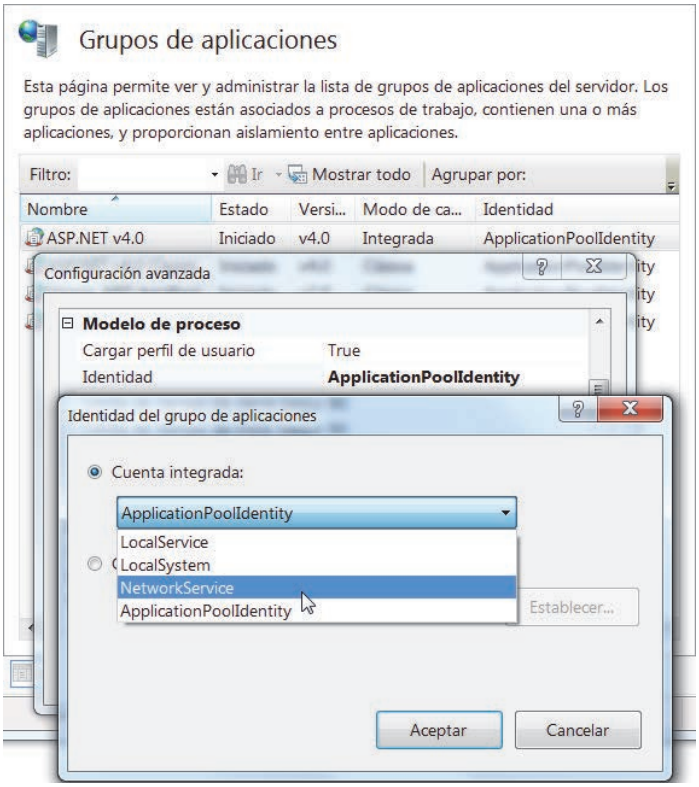
Una vez publicado el servicio, complete esta tarea especificando el nombre del directorio virtual. Para ello, abra el administrador de IIS, seleccione la carpeta *ServicioWcfTfnos* en el sitio web, haga clic sobre ella con el botón secundario del ratón, seleccione la opción *Convertir en aplicación* y seleccione como grupo de aplicaciones ASP.NET v4.0.



Los grupos de aplicaciones establecen límites para las aplicaciones que contienen; estos límites evitan que las aplicaciones de un grupo de aplicaciones afecten a las aplicaciones de otro grupo de aplicaciones.

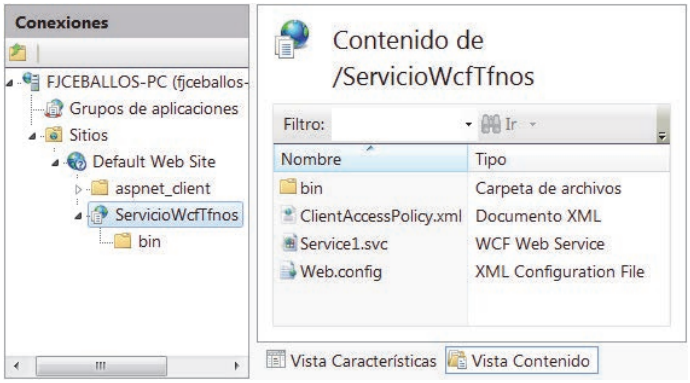
Para mostrar los grupos de aplicaciones existentes en su servidor de aplicaciones, haga clic en el nodo *Grupo de aplicaciones*; se le mostrará la página de la figura siguiente. Por ejemplo, observemos las características del grupo de aplicaciones al cual pertenece nuestro servicio web:

- *Nombre*. Nombre del grupo de aplicaciones.
- *Estado*. Indica si el grupo de aplicaciones está iniciado o no.
- *Versión de .NET Framework*. Versión de .NET Framework que utiliza el grupo de aplicaciones. Si el grupo de aplicaciones se configuró para no utilizar .NET Framework, el valor será *Sin código administrado*.
- *Modo de canalización administrada*. Modo de procesamiento de solicitudes que utiliza el grupo de aplicaciones: *Integrada* o *Clásica*.
- *Identidad*. Identidad utilizada por el grupo de aplicaciones: *Servicio de red*, *Servicio local*, *Sistema local* o el nombre de la cuenta de usuario para una identidad personalizada.
- *Aplicaciones*. Número de aplicaciones que contiene el grupo de aplicaciones.



Nuestro servicio web tiene que acceder a una base de datos SQL Server y SQL Server, para iniciarse y ejecutarse, según explicamos anteriormente en el apartado *Obtener acceso a la base de datos*, utiliza la cuenta *NT AUTHORITY\NetworkService*. Por lo tanto, nuestra aplicación servicio web tiene que ejecutarse con la identidad de la cuenta *NT AUTHORITY\NetworkService*. Para establecer esta identidad, haga clic con el botón secundario del ratón sobre el nombre del grupo de aplicaciones al que pertenece nuestra aplicación, seleccione *Configuración avanzada* y asigne como identidad *NetworkService*.

Cuando haya finalizado, podrá verificar que se ha creado una carpeta, en el ejemplo *C:\inetpub\wwwroot\ServWebWCFConverTemps*, virtual vinculada con la carpeta física localizada en la misma ruta que contiene los ficheros que componen el servicio WCF. El contenido de esta carpeta es el mostrado por la siguiente figura:



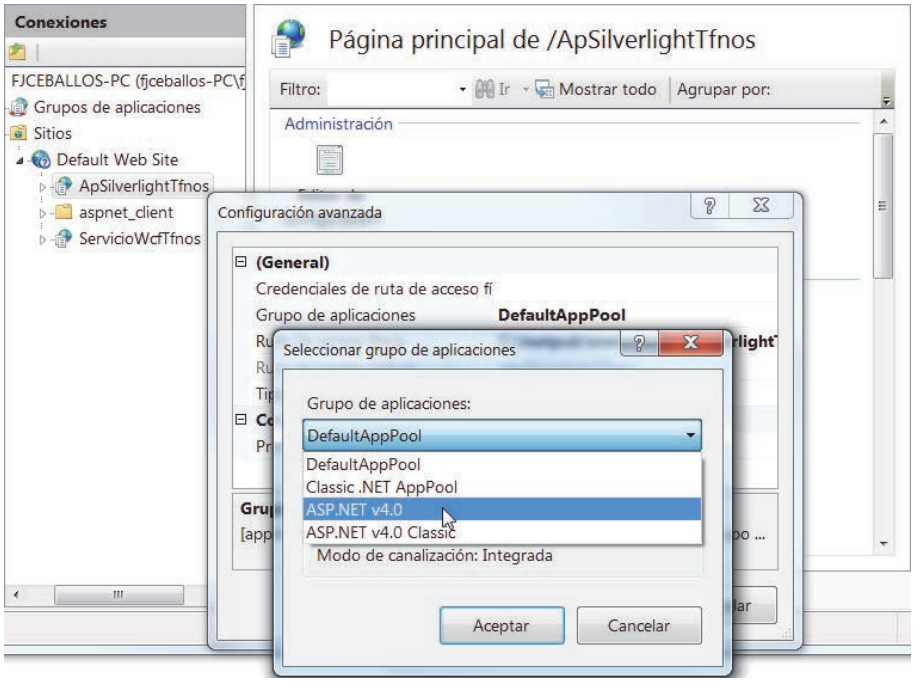
La carpeta *bin* contiene el servicio *ServWebWCFConverTemps.dll*.

El siguiente paso es actualizar la referencia de servicio web de la aplicación Silverlight para que apunte a esta nueva URL. Por lo tanto, elimine en dicha aplicación la referencia de servicio anterior y añada una nueva para esta nueva ubicación del servicio WCF:

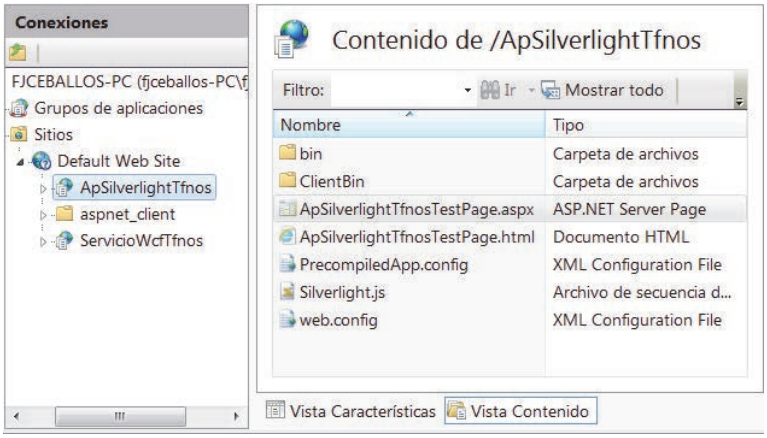
```
http://localhost/ServicioWcfTfnos/Service1.svc
```

Una vez añadida la referencia al servicio ubicado en IIS, compile la aplicación y publíquela (publique el sitio web *ApSilverlightTfnos.Web*).

Una vez publicado el sitio web, si ya fue convertido en aplicación, haga que pertenezca al grupo de aplicaciones ASP.NET v4.0. Para ello, según muestra la figura siguiente, haga clic con el botón secundario del ratón sobre el nombre del sitio, *ApSilverlightTfnos*, seleccione *Administrar aplicación > Configuración avanzada > Grupo de aplicaciones > ASP.NET v4.0*.



Cuando haya finalizado la publicación, podrá verificar que se ha creado una carpeta, en el ejemplo *C:\inetpub\wwwroot\ApSilverlightTfnos*, virtual vinculada con la carpeta física localizada en la misma ruta que contiene los ficheros que componen la aplicación Silverlight. El contenido de esta carpeta para nuestro ejemplo es el mostrado por la siguiente figura:



Para verificar que todo funciona correctamente, abra el explorador y solicite desde el mismo la ejecución de la aplicación:

```
http://localhost/ApSilverlightTfnos/ApSilverlightTfnosTestPage.aspx
```

RESUMEN

En este capítulo hemos estudiado los servicios WCF: aplicaciones que intercambian mensajes con otras aplicaciones, por lo tanto, son ideales para crear soluciones distribuidas.

Expusimos los fundamentos de un servicio web y describimos con un ejemplo cada una de las tareas necesarias para crear un servicio WCF, para crear un cliente que llame al servicio, y cómo publicar el servicio y el cliente en IIS.

Después expusimos los servicios WCF habilitados para Silverlight. En este caso, el servicio web no era independiente, sino que era proporcionado por el propio sitio web que lanza la aplicación Silverlight. En este caso también describimos con un ejemplo todas las tareas necesarias para la creación de este tipo de aplicaciones Silverlight.

Finalmente, construimos una aplicación de N capas (*N-Layer architecture*) y N niveles (*N-Tier architecture*) con las siguientes premisas: poder desplegar la parte correspondiente al nivel de cliente (capa de presentación y lógica de negocio propia de ésta) en un servidor IIS de forma que se pudiera comunicar con la parte nivel de servidor (un servicio WCF para acceder a una base de datos) desplegada en otro servidor IIS; que el nivel de cliente y el nivel de servidor se comunicaran a través de un servicio WCF; que la aplicación Silverlight, a través del servicio web, permitiera al usuario realizar operaciones de selección, actualización, inserción y borrado sobre la base de datos utilizando ADO.NET *Entity Framework*; y publicar el servicio WCF y la aplicación Silverlight en IIS. Todas estas tareas fueron expuestas mediante el desarrollo de una aplicación paso a paso.

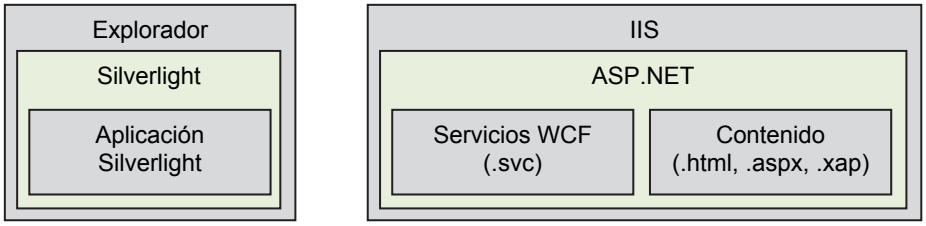
CAPÍTULO 11

© F.J.Ceballos/RA-MA

AUTENTICACIÓN Y AUTORIZACIÓN

La autenticación es el proceso de solicitar y comprobar la identidad de un usuario mediante el examen de sus credenciales y la validación de las mismas consultando a una autoridad determinada. Y la autorización es el proceso de determinar si se permite a un usuario autenticado acceder a un recurso específico. La autorización tiene lugar después de la autenticación y utiliza información relativa a la identidad y funciones del usuario.

Respecto a las aplicaciones Silverlight es importante recordar a la hora de trabajar con la autenticación y autorización que Silverlight es una tecnología del lado del cliente, lo cual significa que el cliente donde se ejecuta la aplicación Silverlight se tendrá que comunicar con el servidor para llevar a cabo las operaciones de autenticación y autorización. Recuerde que Silverlight funciona dentro del recinto de seguridad de explorador y tiene un conjunto reducido de capacidades, por lo que proporciona un mayor grado de seguridad. Por esta razón, Silverlight no se puede conectar con una base de datos directamente, por lo tanto, deberá crear una capa de servicios que proporcione acceso a sus datos y a la lógica empresarial. Ahora bien, como normalmente la ejecución de una aplicación Silverlight se lanza a través de una aplicación ASP.NET, es posible tomar ventaja de los servicios de aplicación de ASP.NET: autenticación, rol y perfil del usuario.

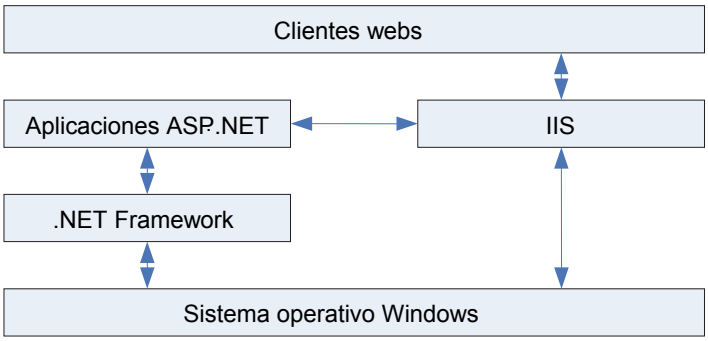


Lo expuesto anteriormente requerirá añadir código extra en el lado del servidor a fin de exponer estos servicios a la aplicación Silverlight para que pueda utilizarlos.

SERVICIOS DE AUTENTICACIÓN

ASP.NET proporciona servicios de aplicación que permiten a distintas aplicaciones cliente (aplicaciones cliente escritas en lenguajes diferentes y que se ejecutan en sistemas operativos diferentes) usar la información de autenticación, rol y perfil del usuario. El único requisito es que estos clientes tienen que comunicarse con el servidor usando el protocolo SOAP.

La figura siguiente muestra una descripción general sobre las relaciones entre los subsistemas y la infraestructura de ASP.NET, con respecto a la seguridad.



Obsérvese que todos los clientes webs se comunican con las aplicaciones ASP.NET a través del servidor de aplicaciones IIS, el cual descifra y, opcionalmente, realiza la autenticación de la solicitud. Si la opción *Autenticación anónima* está establecida a **True**, no se efectuará ninguna autenticación. IIS también busca el recurso solicitado (como una aplicación ASP.NET) y, si se autoriza al cliente, devolverá el recurso correspondiente.

IIS proporciona distintas clases de autenticación para comprobar la identidad del usuario: anónima, básica, de texto implícita, autenticación de Windows y autenticación mediante formularios. El tipo de autenticación que se desee utilizar puede ser seleccionado por medio de los servicios de IIS (*inetmgr.exe*).

Pues bien, ASP.NET implementa el proceso de autenticación a través de los proveedores de autenticación de Windows y mediante formularios (un proveedor de autenticación es un módulo de código; por ejemplo, **WindowsAuthenticationModule** para la autenticación de Windows o **FormsAuthenticationModule**

para la autenticación mediante formularios). El proveedor de autenticación de Windows define cómo utilizar la autenticación de Windows junto con la autenticación de IIS y el proveedor de autenticación mediante formularios permite utilizar un formulario de inicio de sesión para autenticar a los usuarios.

Los dos escenarios más comunes de autenticación son la autenticación de Windows para un sitio de Intranet local o la autenticación mediante formularios para un sitio de Internet. Cuando un usuario solicita una dirección URL que contiene una aplicación ASP.NET, la información sobre la autenticación y la solicitud se entrega a la aplicación.

Autenticación de Windows

La autenticación de Windows aprovecha *Local Security Authority* o *Active Directory* para validar las credenciales del usuario. La autenticación de Windows puede utilizar cualquier esquema compatible con IIS como la autenticación básica, implícita, autenticación integrada (NTLM/Kerberos) o de certificados.

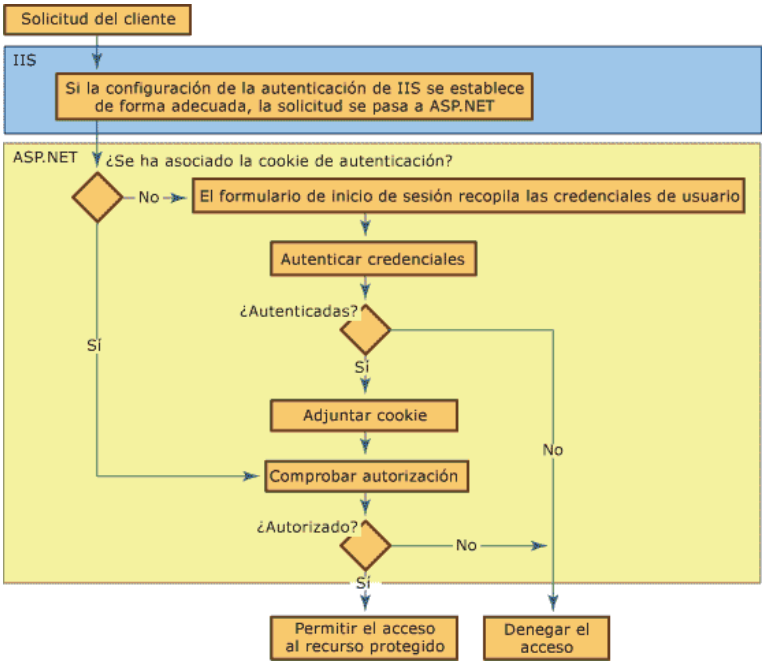
La autenticación integrada es la elección más común para su uso con autenticación de Windows, porque los usuarios no tienen que proporcionar sus nombres de usuario y contraseñas una segunda vez. Esto es, una vez que un usuario inicia una sesión en Windows, el explorador puede reenviar las credenciales que confirman la identidad de la persona. Existen algunos inconvenientes mediante la autenticación integrada, ya que tanto el cliente como el servidor necesitan visibilidad del dominio del usuario. Esto hace que su uso se dirija a escenarios de Intranet. Además, aunque funciona con Microsoft Internet Explorer automáticamente, otros exploradores, como Mozilla Firefox, requieren configuración adicional. Este inconveniente puede ser salvado utilizando la autenticación básica o la implícita ya que ambas forman parte de la especificación de HTTP, por lo que funcionan en la mayoría de los exploradores.

En cuanto a Silverlight, como aprovecha el explorador para la comunicación, resulta fácil la autenticación de Windows con cualquiera de los métodos de autenticación de IIS que acabamos de indicar.

Autenticación mediante formularios

La autenticación mediante formularios es un mecanismo que proporciona una forma simple de autenticación personalizada en ASP.NET. Como tal, es específica de HTTP, lo que significa que también es fácil de usar en Silverlight.

En este caso, el usuario introduce una combinación de nombre de usuario y contraseña, que se envía al servidor para su comprobación. El servidor comprueba las credenciales con respecto a un origen de datos de confianza (a menudo una base de datos de usuarios) y si son correctas, devuelve una cookie que el cliente utilizará para presentarla en las solicitudes posteriores. La cookie está firmada y cifrada para que sólo el servidor pueda descifrarla.



La forma de cómo invocar la autenticación mediante formularios varía en función de cómo se implemente la página de inicio de sesión. Por ejemplo, si se utiliza un formulario web de ASP.NET que redirige el control a la aplicación Silverlight una vez validadas las credenciales del usuario, probablemente todo el trabajo de autenticación ya esté hecho; la cookie ya habrá sido enviada al explorador y la aplicación Silverlight la continuará utilizando siempre que se haga una solicitud a ese dominio. Ahora, si se implementa la página de inicio de sesión dentro de la aplicación de Silverlight, será necesario crear un servicio que exponga los métodos de autenticación y envíe la cookie correspondiente. Afortunadamente, ASP.NET ya proporciona lo que se necesita: el servicio de autenticación. Sólo es necesario habilitarlo en la aplicación.

El mecanismo mediante el cual se comprueba el nombre de usuario y la contraseña es descrito por lo que se denomina proveedor de suscripciones. Afortunadamente, hay varios proveedores de suscripciones disponibles como parte de ASP.NET, incluido uno que puede utilizar bases de datos de SQL Server y otro

que utiliza Active Directory. No obstante, si un proveedor no satisface determinados requerimientos, es sencillo crear una implementación personalizada.

Clase FormsAuthentication

La clase **FormsAuthentication** administra los servicios de autenticación mediante formularios para las aplicaciones webs. Se puede utilizar sola o junto con la clase **Membership** con objeto de crear un sistema completo para autenticar a los usuarios de una aplicación o sitio web. La clase **Membership** permite crear nuevos usuarios, almacenar la información (nombres de usuario, contraseñas, direcciones de correo electrónico y datos compatibles) en SQL Server o en un almacén de datos alternativo, autenticar a los usuarios que visitan el sitio, administrar contraseñas que incluyen su creación, cambio, recuperación y restablecimiento, etc.

Con la autenticación mediante formularios, la información sobre el usuario se almacena en un origen de datos externo, como una base de datos SQL Server, o en el fichero de configuración de la aplicación.

La autenticación de formularios se habilita al establecer en el fichero de configuración de la aplicación el atributo **mode** del elemento de configuración **<authentication>** en *Forms*:

```
<!-- Fichero web.config -->
<authentication mode="Forms">
   <forms name=".ASPXFORMSAUTH" loginUrl="login.aspx">
   </forms>
</authentication>
<authorization>
  <deny users="?" />
</authorization>
```

El atributo **name** especifica el nombre de la cookie HTTP utilizada para la autenticación. De forma predeterminada, el nombre de la cookie que contiene las credenciales del usuario utilizadas para la autenticación es .ASPXFORMSAUTH. Si se ejecutan varias aplicaciones en un mismo servidor y cada una requiere una cookie única, debe configurar el nombre de la misma en el fichero *web.config* de cada aplicación.

El atributo *loginUrl* especifica la dirección URL a la que debe redirigirse la solicitud de inicio de sesión si no se encuentra ninguna cookie de autenticación válida. El valor predeterminado es *login.aspx*.

Si fuera preciso configurar dentro de **<forms>** un almacén de credenciales, podría hacerse a través de la etiqueta **<credentials>**. Por ejemplo:

```
<credentials passwordFormat="Clear">
  <user name="usuario1" password="contraseña1"/>
  <user name="usuario2" password="contraseña2"/>
</credentials>
```

En este caso, las credenciales proporcionadas se validarían, comparándolas con las contenidas en el almacén de credenciales configurado, así:

```
If (FormsAuthentication.Authenticate( _
      ctUsuario.Text, ctContraseña.Text)) Then
  ' Retornar a la petición (URL) inicial
  FormsAuthentication.RedirectFromLoginPage( _
      ctUsuario.Text, cvRecordarContraseña.Checked)
Else
  etError.Text = "Error: Las credenciales no son válidas. Inténtelo ota vez."
End If
```

En la página de inicio de sesión (*login.aspx*), el usuario (cliente) escribe las credenciales requeridas (por lo general, el nombre y la contraseña).

El código de la aplicación comprueba las credenciales para confirmar su autenticidad, normalmente, en un controlador de eventos. Si se autentican las credenciales, la aplicación crea una cookie que incluye el nombre de usuario, pero no la contraseña. Si la autenticación falla, la solicitud se devuelve normalmente con el mensaje "Acceso denegado" o se vuelve a presentar el formulario de inicio de sesión.

Si se autentica al usuario, ASP.NET comprueba la autorización. Si se autoriza al usuario, ASP.NET puede permitir el acceso al recurso protegido que se ha solicitado en un principio o redirigir la solicitud a otra página, dependiendo del diseño de la aplicación. También puede dirigir la solicitud a un módulo de autorización personalizado donde se prueban las credenciales para autorizar el acceso al recurso protegido. Si la autorización falla, ASP.NET siempre redirige a la página de inicio de sesión.

Autenticación mediante formularios en Silverlight

Hacer uso de la autenticación mediante formularios en una aplicación Silverlight es sencillo. Para empezar, vamos a crear una nueva aplicación de navegación Silverlight. Para ello, abra Visual Studio o Visual Web Developer y seleccione *Archivo > Nuevo > Proyecto > Visual Basic > Silverlight > Aplicación de navegación Silverlight*. Denomine al proyecto, por ejemplo, *ApNavSilverlight* y continúe según explicamos en el capítulo *Silverlight*. La solución creada incluirá el proyecto de cliente *ApNavSilverlight* y el proyecto de servidor (sitio web ASP.NET) *ApNavSilverlight.Web*.

El primer cambio que vamos a hacer es sobre el fichero de configuración del proyecto de servidor para habilitar la autenticación de formularios. Para ello, hay que establecer en el fichero *web.config* el atributo **mode** del elemento de configuración **<authentication>** en *Forms*:

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0"/>
    <authentication mode="Forms">
      <forms>
        <credentials passwordFormat="Clear">
          <user name="usuario1" password="contraseña1"/>
          <user name="usuario2" password="contraseña2"/>
        </credentials>
      </forms>
    </authentication>
  </system.web>
</configuration>
```

Para hacer una prueba rápida y evitar construir una base de datos de usuarios, el elemento **<credentials>** incluye las credenciales correspondientes a dos usuarios. Este elemento tiene un atributo **passwordFormat** cuyos valores posibles son: *Clear*, especifica que las contraseñas no se cifran, *MD5*, especifica que las contraseñas se cifran con el algoritmo hash MD5, y *SHA1*, especifica que las contraseñas se cifran con el algoritmo hash SHA1.

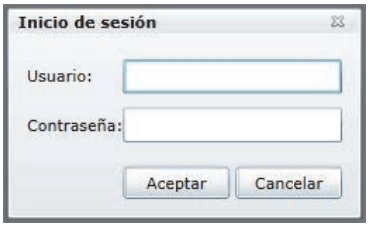
Obviamente, especificar las contraseñas sin cifrar no es una buena opción. El método **HashPasswordForStoringInConfigFile** de la clase **FormsAuthentication**,

```
Public Shared Function HashPasswordForStoringInConfigFile ( _
   password As String, _
   passwordFormat As String _
) As String
```

crea un valor de contraseña con algoritmo hash que se puede utilizar al almacenar las credenciales de autenticación de formularios en el archivo de configuración de una aplicación. Por ejemplo, las credenciales anteriores cifradas serían:

```
<credentials passwordFormat="SHA1">
  <user name="usuario1"
        password="DF9D6B3574AF0E25FFA4BF3286CA551D4F7D2A2B" />
  <user name="usuario2"
        password="48E9CF9644D16A6D363D91FCC453E9EFC7625A32" />
</credentials>
```

Una vez configurada la aplicación, se debe proporcionar un formulario de inicio de sesión:

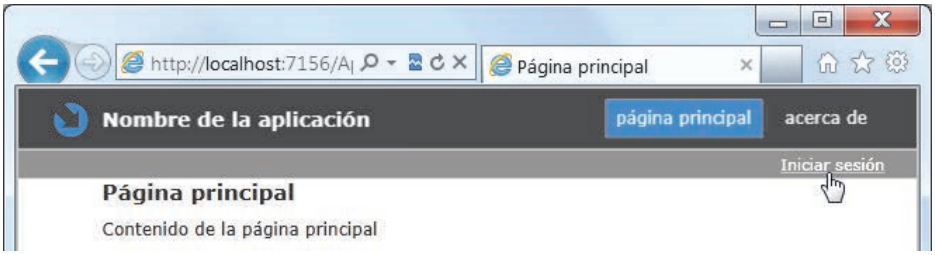


Esta ventana será un objeto de la clase *Login* derivada de **ChildWindow**. Su diseño, incluidos los controladores de eventos, es el mismo que el que expusimos en el capítulo *Silverlight*, por lo cual no volveremos a repetirlo aquí. Por lo tanto, añada a la carpeta *Views* del proyecto cliente un nuevo elemento de tipo *ventana secundaria de Silverlight* denominado *Login* y realice el diseño según expusimos en el apartado *ChildWindow* del capítulo *Silverlight*. La clase *Login* implementa dos propiedades, *Usuario* y *Contraseña*, para devolver los datos correspondientes introducidos por el usuario en las cajas de texto.

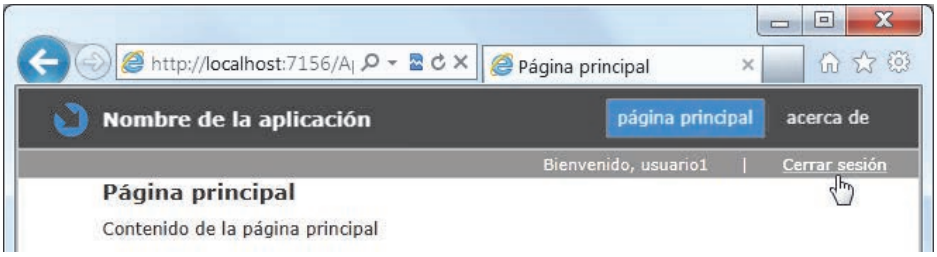
Esta ventana será mostrada desde un elemento "Iniciar sesión" de tipo **HyperlinkButton**, denominado *hlInicioSesion*, de la página principal, *MainPage*. Añadimos entonces este elemento, junto con otros que se indican en el código XAML siguiente, a dicha página:

```
<UserControl ... >
  ...
    </Grid>
    <Border Style="{StaticResource LoginContainerStyle}">
      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Right">
        <TextBlock Name="tbBienvenido"
                   Style="{StaticResource TextBlockStyle}" />
        <TextBlock Name="tbSeparador" Text="|"
                   Visibility="Collapsed"
                   Style="{StaticResource TextBlockStyle}" />
        <HyperlinkButton Name="hlInicioSesion"
                   Content="Iniciar sesión"
                   Style="{StaticResource HyperlinkButtonStyle}"
                   Click="hlInicioSesion_Click" />
      </StackPanel>
    </Border>
  </Grid>
</UserControl>
```

Este diseño dará lugar a un borde relleno de gris con el elemento "Iniciar sesión" a la derecha.



Los otros elementos incluidos en el diseño de este borde gris aparecerán cuando el usuario se haya autenticado. Obsérvese en la figura siguiente que, una vez autenticado el usuario, se le da la bienvenida y el elemento "Iniciar sesión" pasa a contener "Cerrar sesión". El borde volverá a su estado inicial cuando se cierre la sesión.



A continuación, añadimos el controlador del evento **Click** del elemento *hlInicioSesion* desde el que mostraremos el formulario de inicio de sesión, para lo cual será necesario crear un objeto de la clase *Login* e invocar a su método **Show**.

También, es importante recordar que aunque la ventana hija bloquea la interacción con la página padre, **Show** no bloquea la ejecución del código de la misma, que seguirá ejecutándose en segundo plano. Por eso, para obtener los datos introducidos una vez cerrada la ventana hija después de que el usuario haya pulsado el botón *Aceptar*, tendremos que hacerlo desde el controlador del evento **Closed** de la misma.

Según lo expuesto, añada el código siguiente a la página *MainPage*.

```
Partial Public Class MainPage
  Inherits UserControl

  Private formLogin As New Login()

  Public Sub New()
    InitializeComponent()
    AddHandler formLogin.Closed, AddressOf formLogin_Closed
  End Sub
```

```
    ' ...

  Private Sub hlInicioSesion_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    If hlInicioSesion.Content.ToString() = "Iniciar sesión" Then
      formLogin.Show()
    Else
      tbBienvenido.Text = ""
      tbSeparador.Visibility = Visibility.Collapsed
      hlInicioSesion.Content = "Iniciar sesión"
    End If
  End Sub

  Private Sub formLogin_Closed(sender As Object, e As EventArgs)
    If formLogin.DialogResult = True Then
      ' Aquí verificaríamos si las credenciales son válidas

      'formLogin.txtContraseña.Password
      'formLogin.txtUsuario.Text
    End If
  End Sub
End Class
```

Obsérvese que la propiedad **DialogResult** de **ChildWindow** sólo valdrá **True** si el usuario hizo clic en el botón *Aceptar*. Si esta condición se da, se verificaría si el usuario y la contraseña son correctos y se procedería según el resultado. En nuestro ejemplo, como veremos a continuación, nos limitaremos a dar la bienvenida al usuario o a notificar que las credenciales no son correctas.

¿Cómo verificamos las credenciales? Según dijimos anteriormente, será necesario crear un servicio que exponga los métodos de autenticación. Por lo tanto, vamos a añadir al proyecto de servidor (*ApNavSilverlight.Web*) un servicio WCF habilitado para Silverlight denominado *Login*.

A continuación, los pasos que tenemos que seguir para completar el servicio son los mismos que expusimos en el apartado *Crear un servicio WCF habilitado para Silverlight* del capítulo *Servicios WCF*. Según lo explicado allí, vamos a definir el contrato del servicio (**ServiceContract**) según se muestra a continuación:

```
Imports System.ServiceModel
Imports System.ServiceModel.Activation

<ServiceContract(Namespace:="")>
<SilverLightFaultBehavior>
<AspNetCompatibilityRequirements(RequirementsMode:= _
AspNetCompatibilityRequirementsMode.Allowed)>
Public Class Login
```

```
  <OperationContract()>
  Public Function Authenticate(usuario As String,
  password As String) As Boolean
    If FormsAuthentication.Authenticate(usuario, password) Then
      Return True
    End If
    Return False
  End Function

  <OperationContract()>
  Public Sub SignOut()
    FormsAuthentication.SignOut()
  End Sub
End Class
```

Observamos que la clase *Login* tiene dos métodos: *Authenticate* y *SignOut*. *Authenticate* invoca al método del mismo nombre de la clase **FormsAuthentication** para validar el nombre de usuario y la contraseña pasados como argumentos contra las credenciales almacenadas en el fichero de configuración *web.config* del proyecto de servidor; el resultado será un valor **True** o **False**. Y *SignOut* se puede utilizar para que el usuario cierre la sesión y se borre la cookie de autenticación de formularios.

El cliente de este servicio WCF lógicamente es la propia aplicación Silverlight (el proyecto de cliente). Para que esta aplicación tenga acceso a las operaciones proporcionadas por el servicio WCF vamos añadir al proyecto de cliente una referencia al servicio que vamos a denominar *RefServicioLogin*.

Una vez agregada en el cliente la referencia al servicio WCF, el siguiente paso es crear un objeto de la clase *RefServicioLogin.Loging* del *proxy* para acceder a los métodos del servicio. Recuerde que todas las llamadas que se realicen al servicio web desde Silverlight son asíncronas, de ahí que tengamos que añadir los controladores de los eventos que se generan cada vez que se completa cada una de las operaciones solicitadas. También tendremos que garantizar que la URL actual del servicio sea la que tiene que ser, independientemente de dónde se instale dicho servicio (recuerde que, por ejemplo, el número de puerto del servidor puede cambiar cuando se instale en un servidor web de producción). Esto lo haremos proporcionando la URL absoluta del servicio en el instante de crear el objeto de la clase *RefServicioLogin.Loging*. Para cualquier otro detalle sobre cómo implementar y publicar un servicio WCF habilitado para Silverlight, repase el capítulo titulado *Servicios WCF*. Según lo expuesto, añada el siguiente código a la clase *MainPage* que da lugar a la página principal:

```
Imports System.Windows.Navigation

Partial Public Class MainPage
```

```
  Inherits UserControl

  Private formLogin As New Login()
  Private svc As RefServicioLogin.LoginClient = Nothing

  Public Sub New()
    InitializeComponent()
    AddHandler formLogin.Closed, AddressOf formLogin_Closed

    Dim uri As New Uri(Application.Current.Host.Source,
                       "../Login.svc")
    svc = New RefServicioLogin.LoginClient(
      "CustomBinding_Login", uri.AbsoluteUri)

    AddHandler svc.AuthenticateCompleted,
      AddressOf svc_AuthenticateCompleted

    AddHandler svc.SignOutCompleted,
      AddressOf svc_SignOutCompleted
  End Sub

  Private Sub ContentFrame_Navigated(sender As Object,
  e As NavigationEventArgs)
    ' ...
  End Sub

  Private Sub ContentFrame_NavigationFailed(sender As Object,
  e As NavigationFailedEventArgs)
    ' ...
  End Sub

  Private Sub hlInicioSesion_Click(sender As Object,
  e As RoutedEventArgs)
    ' ...
  End Sub

  Private Sub formLogin_Closed(sender As Object, e As EventArgs)
    ' ...
  End Sub

  Private Sub svc_AuthenticateCompleted(sender As Object,
  e As RefServicioLogin.AuthenticateCompletedEventArgs)

  End Sub

  Private Sub svc_SignOutCompleted(sender As Object,
  e As System.ComponentModel.AsyncCompletedEventArgs)

  End Sub
End Class
```

Entonces, para verificar las credenciales habrá que invocar al método *AuthenticateAsync* del servicio, una vez se cierre el formulario de inicio de sesión:

```
Private Sub formLogin_Closed(sender As Object, e As EventArgs)
  If formLogin.DialogResult = True Then
    svc.AuthenticateAsync(formLogin.txtUsuario.Text,
                          formLogin.txtContraseña.Password)
  End If
End Sub
```

Y cuando la operación de verificación de las credenciales finalice, si son correctas, daremos la bienvenida al usuario y cambiaremos el título "Iniciar sesión" por "Cerrar sesión", y si no son correctas, mostraremos un mensaje de error al usuario. Para mostrar un mensaje de error podemos utilizar una ventana secundaria de la clase *ErrorWindow* proporcionada por la aplicación Silverlight.

```
Private Sub svc_AuthenticateCompleted(sender As Object,
e As RefServicioLogin.AuthenticateCompletedEventArgs)
  If e.Result Then
    tbSeparador.Visibility = Visibility.Visible


    tbBienvenido.Text = "Bienvenido, " & formLogin.txtUsuario.Text
    hlInicioSesion.Content = "Cerrar sesión"
  Else
    Dim errorWin As ChildWindow = New ErrorWindow("Error:",
                      "Las credenciales no son correctas")
    errorWin.Show()
  End If
End Sub
```

Y para cerrar la sesión habrá que invocar al método *SignOutAsync* del servicio una vez el usuario haga clic en "Cerrar sesión":

```
Private Sub hlInicioSesion_Click(sender As Object, e As RoutedEventArgs)
  If hlInicioSesion.Content.ToString() = "Iniciar sesión" Then
    formLogin.Show()
  Else
    tbBienvenido.Text = ""
    tbSeparador.Visibility = Visibility.Collapsed
    hlInicioSesion.Content = "Iniciar sesión"
    svc.SignOutAsync()
  End If
End Sub
```

Y cuando la operación de cerrar sesión finalice, simplemente verificaremos si terminó satisfactoriamente o si ocurrió algún error. En este último caso, mostra-

remos un mensaje de error al usuario por medio de la ventana secundaria de la clase *ErrorWindow*.

```
Private Sub svc_SignOutCompleted(sender As Object,
e As System.ComponentModel.AsyncCompletedEventArgs)
  If e.Error IsNot Nothing Then
    Dim s As String = e.Error.Message
    Dim errorWin As ChildWindow = New ErrorWindow("Error:", s)
    errorWin.Show()
  End If
End Sub
```

El proceso de autenticación está finalizado. Pruebe la aplicación y observe los resultados. Después, puede publicarla y observar que todo sigue funcionando correctamente.

SERVICIOS DE APLICACIÓN DE ASP.NET

Hemos visto cómo utilizar la funcionalidad proporcionada por la clase **FormsAuthentication** para implementar la autenticación en una aplicación Silverlight. Pero ASP.NET proporciona servicios de aplicación, a los que se puede obtener acceso en la web, que permiten a las aplicaciones cliente usar la información de autenticación, rol y perfil del usuario. Cualquier aplicación cliente que pueda comunicarse usando el protocolo SOAP, independientemente del lenguaje y del sistema operativo utilizado, puede tener acceso a estos servicios.

Los servicios de aplicación de perfiles, roles y autenticación de formularios de ASP.NET a los que nos hemos referido, están implementados bajo el espacio de nombres **System.Web.ApplicationServices** y pueden incluirse en una aplicación ASP.NET como servicios WCF y, por lo tanto, ser utilizados en una aplicación Silverlight.

De forma resumida, este espacio de nombres incluye las clases siguientes:

- **AuthenticationService**. Permite el acceso a la autenticación de formularios de ASP.NET como servicio web WCF.

- **RoleService**. Permite el acceso al proveedor de funciones de ASP.NET como un servicio web WCF.

- **ProfileService**. Permite el acceso a los perfiles de usuario de ASP.NET a través de un servicio web WCF.

- **ApplicationServicesHostFactory**. Clase para crear instancias del servicio de autenticación, del servicio de funciones y del servicio de perfiles en entornos administrados.

- **AuthenticatingEventArgs**. Proporciona datos para el evento **Authenticating** que se genera cuando se validan las credenciales del usuario.

- **CreatingCookieEventArgs**. Proporciona datos para el evento **CreatingCookie** que se genera cuando se establece la cookie de autenticación.

- **SelectingProviderEventArgs**. Proporciona datos para el evento **SelectingProvider** que se genera cuando una instancia de **RoleService** determina qué proveedor de función se va a utilizar para comprobar las funciones del usuario.

- **ValidatingPropertiesEventArgs**. Proporciona datos para el evento **ValidatingProperties** que se genera cuando se establecen las propiedades del perfil de un usuario.

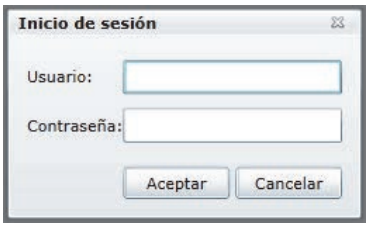
- **ProfilePropertyMetadata**. Proporciona una colección de valores que describe una propiedad de perfil.

- **KnownTypesProvider**. Esta API es compatible con la infraestructura de .NET Framework y no está diseñada para utilizarse directamente desde el código. Permite crear una matriz de objetos **Type** que contienen los tipos de propiedades de perfiles individuales.

Crear la estructura de la aplicación

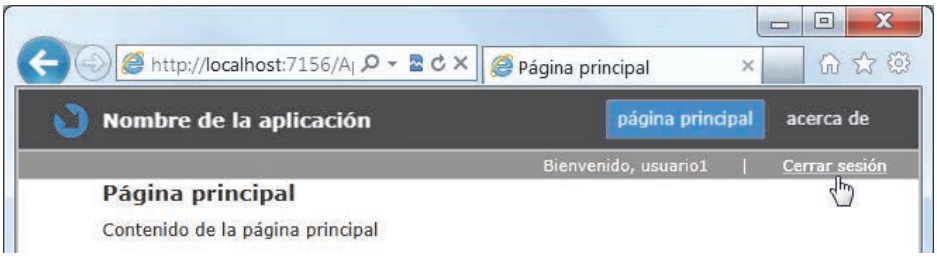
El primer paso para trabajar con los servicios de aplicación de ASP.NET desde una aplicación Silverlight es crear un servicio WCF que actúe de proveedor de estos servicios. Para verlo de una forma práctica, vamos a crear una nueva aplicación de navegación Silverlight denominada *ApNavSilverlightSvcAps*. La solución creada incluirá el proyecto de cliente *ApNavSilverlightSvcAps* y el proyecto de servidor (sitio web ASP.NET) *ApNavSilverlightSvcAps.Web*.

A continuación, igual que en el ejemplo anterior, añada a la carpeta *Views* del proyecto cliente un formulario de inicio de sesión:



Esta ventana será un objeto de la clase *Login* derivada de **ChildWindow**. La clase *Login* implementa dos propiedades, *Usuario* y *Contraseña*, para devolver los datos correspondientes introducidos por el usuario en las cajas de texto.

Esta ventana será mostrada desde un elemento "Iniciar sesión" de tipo **HyperlinkButton**, denominado *hlInicioSesion*, de la página principal, *MainPage*. Añadimos entonces este elemento, siguiendo los mismos pasos del ejemplo anterior, para que se muestre en la interfaz según muestra la figura siguiente:



Observe el borde relleno de gris; inicialmente sólo se mostrará un elemento "Iniciar sesión" a la derecha. Los otros elementos incluidos en el diseño de este borde gris, los que muestra la figura, aparecerán cuando el usuario se haya autenticado. Obsérvese en la figura que, una vez autenticado el usuario, se le da la bienvenida y el elemento "Iniciar sesión" pasa a contener "Cerrar sesión". El borde volverá a su estado inicial cuando se cierre la sesión.

A continuación, añadimos el controlador del evento **Click** del elemento *hlInicioSesion* desde el que mostraremos el formulario de inicio de sesión, para lo cual será necesario crear un objeto de la clase *Login* e invocar a su método **Show**. Una vez mostrada la ventana, el usuario introducirá el usuario y la contraseña y pulsará el botón *Aceptar*. Como ya sabemos, para obtener los datos introducidos, una vez cerrada la ventana hija, tendremos que hacerlo desde el controlador del evento **Closed** de la misma. Según lo expuesto, añada el código siguiente a la página *MainPage*.

```
Partial Public Class MainPage
  Inherits UserControl
```

```
  Private formLogin As New Login()

  Public Sub New()
    InitializeComponent()
    AddHandler formLogin.Closed, AddressOf formLogin_Closed
  End Sub

  ' ...

  Private Sub hlInicioSesion_Click(sender As System.Object,
  e As System.Windows.RoutedEventArgs)
    If hlInicioSesion.Content.ToString() = "Iniciar sesión" Then
      formLogin.Show()
    Else
      tbBienvenido.Text = ""
      tbSeparador.Visibility = Visibility.Collapsed
      hlInicioSesion.Content = "Iniciar sesión"
    End If
  End Sub

  Private Sub formLogin_Closed(sender As Object, e As EventArgs)
    If formLogin.DialogResult = True Then
      ' Aquí verificaríamos si las credenciales son válidas

      'formLogin.txtContraseña.Password
      'formLogin.txtUsuario.Text
    End If
  End Sub
End Class
```

Asignar y configurar servicios de aplicación

Ahora, para validar y exponer la información de los usuarios que crearemos más adelante y guardaremos en una base de datos SQL Server, utilizaremos los servicios de aplicación ASP.NET. Antes de poder tener acceso a la información del usuario en el cliente, hay que añadir los ficheros *.svc* (servicios webs WCF) que señalen a los servicios de aplicación, para que éstos estén disponibles para la aplicación Silverlight. También debe configurar el sitio web para que los servicios de aplicación se expongan en la red.

Pues bien, para añadir los ficheros *.svc* puede hacerlo manualmente o proceder así: agregue un servicio WCF (normal) al sitio web y denomínelo *AuthenticationService*. Después, abra el fichero *AuthenticationService.svc* y modifíquelo para que la clase del servicio sea **AuthenticationService** del espacio de nombres **System.Web.ApplicationServices**:

```
<%@ ServiceHost Language="VB" Debug="true"
    Service="System.Web.ApplicationServices.AuthenticationService"
 %>
```

Como vemos, este servicio no implementa un servicio de autenticación personalizado, sino que utiliza la clase **AuthenticationService** de la biblioteca .Net. Por lo tanto, elimine los ficheros de la carpeta *App_Code* relacionados con el servicio que añadimos.

El paso siguiente es configurar el servicio de aplicación **AuthenticationService**. Para ello, abra el archivo *web.config* y edítelo como se indica a continuación:

```
<configuration>
  <system.web>
    <authentication mode="Forms" />
    <compilation debug="true" targetFramework="4.0" />
  </system.web>

  <system.web.extensions>
    <scripting>
      <webServices>
        <authenticationService enabled="true" requireSSL = "false"/>
      </webServices>
    </scripting>
  </system.web.extensions>

  <system.serviceModel>
    <services>
      <service name="System.Web.ApplicationServices.AuthenticationService"
      behaviorConfiguration="ApplicationServiceTypeBehaviors">
        <endpoint contract="System.Web.ApplicationServices.AuthenticationService"
        binding="basicHttpBinding"
        bindingConfiguration="userHttp"
        bindingNamespace="http://asp.net/ApplicationServices/v200"/>
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name="userHttp">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <behaviors>
      <serviceBehaviors>
        <behavior name="ApplicationServiceTypeBehaviors">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
  </system.serviceModel>
</configuration>
```

Observe que hemos especificado que la autenticación será mediante formularios, hemos añadido el elemento **system.web.extensions** para habilitar el servicio

de autenticación, hemos añadido el elemento **system.serviceModel** para definir el *extremo* o punto final del servicio, el cual proporciona la única manera de comunicación con el mismo. Un *extremo* está compuesto de una *dirección*, un *enlace* y un *contrato* (para más detalles, véase el capítulo *Servicios WCF*). Observe también el elemento **<security>** que especifica el tipo de seguridad que se puede utilizar con **<basicHttpBinding>**; de forma predeterminada, el mensaje SOAP no está protegido durante la transferencia (valor *None*). También, para poder contar con muchas de las características de ASP.NET, hay que habilitar el modo de compatibilidad ASP.NET, atributo *aspNetCompatibilityEnabled*.

A continuación, vamos a repetir los pasos anteriores para añadir y configurar otros dos servicios definidos por las clases **RoleService** y **ProfileService** del espacio de nombres **System.Web.ApplicationServices**:

```
<%@ ServiceHost Language="VB" Debug="true"
    Service="System.Web.ApplicationServices.RoleService"
 %>

<%@ ServiceHost Language="VB" Debug="true"
    Service="System.Web.ApplicationServices.ProfileService"
 %>
```

La configuración sería análoga a la realizada para el servicio de autenticación:

```
<configuration>
  <system.web>
    <compilation debug="false" targetFramework="4.0" />
    <authentication mode="Forms" />
    <roleManager enabled="true" />
    <profile enabled="true" />
  </system.web>

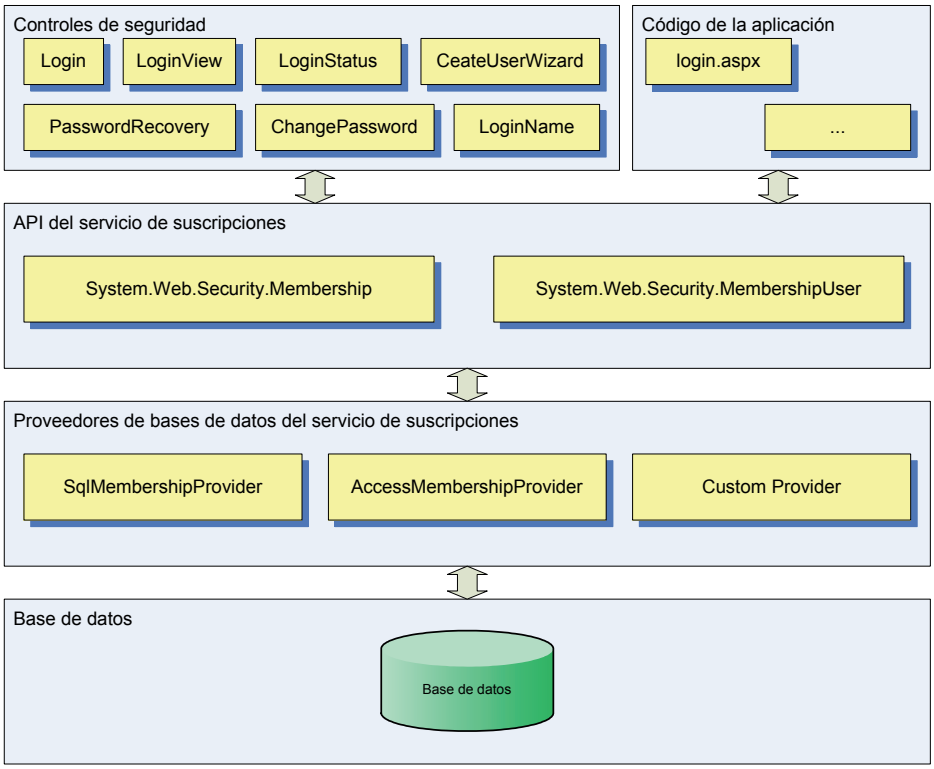
  <system.web.extensions>
    <scripting>
      <webServices>
        <authenticationService enabled="true" requireSSL="false"  />
        <roleService enabled="true" />
        <profileService enabled="true"/>
      </webServices>
    </scripting>
  </system.web.extensions>

  <system.serviceModel>
    ...
  </system.serviceModel>
  ...
<configuration>
```

Evidentemente, cualquiera de los servicios añadidospuede omitirse si la funcionalidad que proveen no va a ser utilizada por nuestra aplicación.

Crear usuarios

El siguiente paso es crear algunos usuarios para que podamos trabajar con nuestra aplicación. Para ello, ASP.NET proporciona lo que denominamos *servicio de suscripciones* (ASP.NET *membership*). Se trata de un servicio integrado que permite validar y almacenar las credenciales de un usuario (los controles de seguridad que vemos en la figura siguiente están disponibles en ASP.NET pero no en WPF/Silverlight, donde tendremos que construirlos). Este servicio, básicamente, proporciona soporte para crear nuevos usuarios y contraseñas, almacenar la información de cada usuario en una base de datos (SQL Server, Access, etc.), autenticar a los usuarios que visiten el sitio web y administrar las contraseñas. La figura siguiente muestra este servicio y su relación con el resto de los elementos:



De forma predeterminada este servicio de suscripciones integrado utiliza para almacenar los datos una base de datos SQL Server Express denominada ASPNETDB.MDF y que será creada y añadida automáticamente a la carpeta *App_Data* del proyecto web en cuanto añadamos el primer usuario.

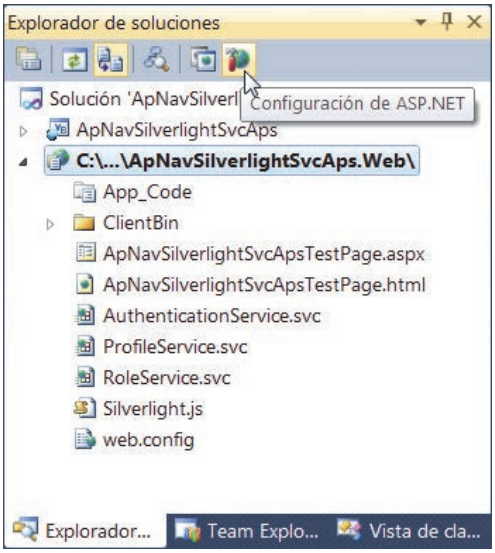
Para utilizar el servicio de suscripciones puede hacerlo con la configuración predeterminada o puede configurarlo para el sitio web; por ejemplo, suponga que

quiere modificar los requerimientos predeterminados exigidos para admitir una contraseña. Para ello, tiene que especificar las opciones del servicio de suscripciones como parte de la configuración del sitio web (fichero *web.config*, sección *system.web*, elemento *membership*). Por ejemplo:

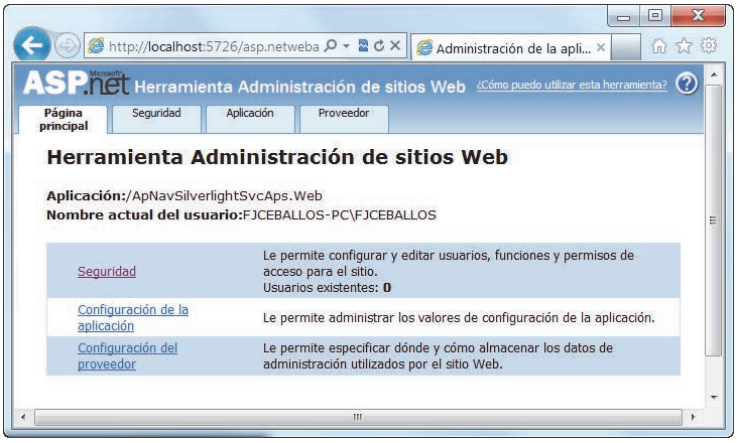
```
<system.web>
  <membership>
    <providers>
      <add name="AspNetSqlProvider"
      type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="LocalSqlServer"
      enablePasswordRetrieval="true"
      enablePasswordReset="true"
      requiresQuestionAndAnswer="false"
      requiresUniqueEmail="true"
      passwordFormat="Hashed"
      minRequiredNonalphanumericCharacters="0"
      minRequiredPasswordLength="6" />
    </providers>
  </membership>
</system.web>
```

De forma predeterminada, el servicio de suscripciones está activado y el proveedor de base de datos es **SqlMembershipProvider**.

Para poner en marcha este servicio, seleccione el proyecto de servidor (*ApNavSilverlightSvcAps.Web*) y haga clic en el botón *Configuración de ASP.NET* de la barra de herramientas del explorador de soluciones, según muestra la figura siguiente.



Se mostrará una página principal que pone a su disposición la *herramienta de administración de sitios webs*. Esta página también puede abrirla ejecutando la orden *Configuración de ASP.NET* del menú *Sitio web* de Visual Studio. Como se puede observar en la figura siguiente, esta página presenta tres opciones: *Seguridad* (editar usuarios y funciones), *Configuración de la aplicación* y *Configuración del proveedor* de suscripciones.



Para empezar, haga clic en *Seguridad*. Se le mostrará una página que le permitirá configurar usuarios, funciones y reglas de acceso. Quizás, llevar a cabo estas operaciones resulte más sencillo si hace clic en el enlace "Utilice el Asistente para la configuración de seguridad para configurar la seguridad paso a paso". Haga, por lo tanto, clic en este enlace, se muestra una página de bienvenida, y después en *Siguiente*.

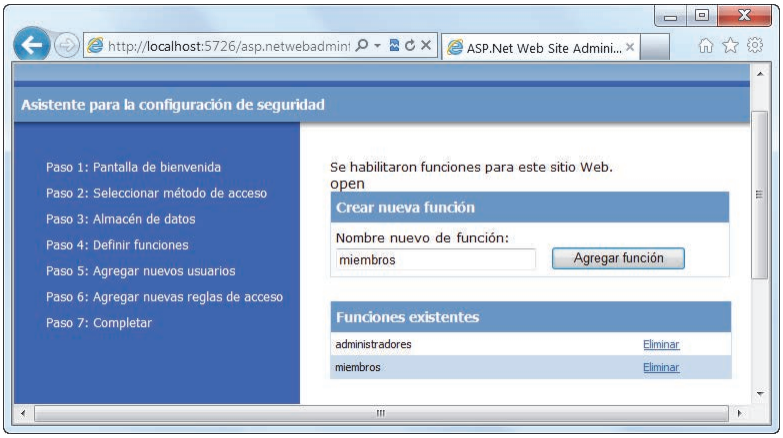
El primer paso que debe realizar para garantizar la seguridad del sitio es identificar a los usuarios (autenticación). El método para establecer la identidad de un usuario depende de la manera en que el usuario obtiene acceso al sitio.

Seleccionar el método de acceso *Desde Internet*. Haga clic en *Siguiente*.

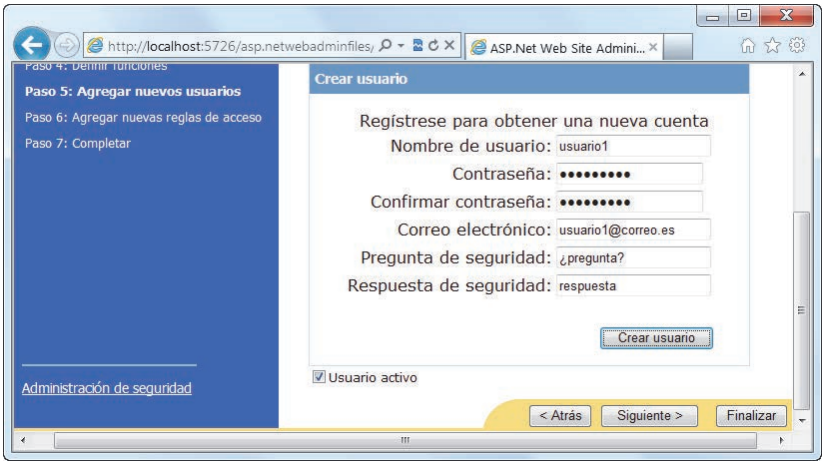
Almacén de datos. La aplicación está configurada actualmente para utilizar configuración de proveedores avanzada (en la pestaña *Proveedor* puede ver para qué proveedor está configurada la aplicación; por ejemplo, para **AspNetSqlProvider**). Si lo desea, puede cambiar la manera en que se almacenarán los datos de administración de sitios webs. Nosotros dejamos el tipo de almacén predeterminado. Haga clic en *Siguiente*.

Definir funciones. Opcionalmente, puede agregar funciones que le permitirán otorgar o denegar acceso a carpetas específicas del sitio web a grupos de usuarios. Por ejemplo, puede crear funciones como "administradores", "ventas" o "miem-

bros", cada una con diferente acceso a carpetas específicas. Después, puede agregar usuarios a estas funciones. Haga clic en *Siguiente*.



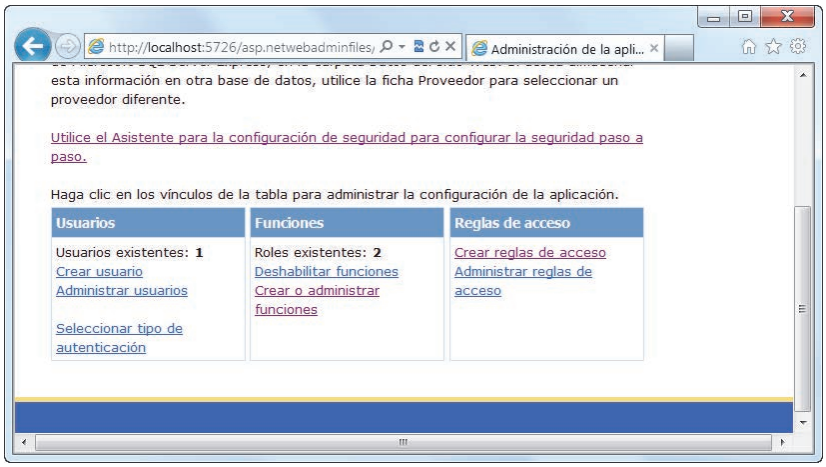
Agregar nuevos usuarios. Para agregar un usuario, escriba el nombre, la contraseña y la dirección de correo electrónico del usuario en esta página. Así mismo, puede especificar una pregunta con la respuesta que el usuario debe proporcionar para restablecer la contraseña o solicitar la contraseña olvidada. La contraseña debe contener letras, dígitos y caracteres especiales. Cuando haya completado el formulario, haga clic en *Crear usuario* y después en *Continuar*. Repita este proceso para añadir nuevos usuarios. Cuando termine haga clic en *Siguiente*.



Agregar nuevas reglas de acceso. Si se definen varias reglas, se aplicarán en el orden en el que aparecen en la tabla. Se aplicará la primera regla que coincida. Cuando termine haga clic en *Siguiente*.

El asistente para la configuración de seguridad se ha completado. Haga clic en *Finalizar*.

Más adelante podrá agregar, eliminar y modificar los usuarios, las funciones y las reglas, según puede observar en la página siguiente:



¿Dónde se han almacenado los datos? Quizás quiera echar una ojeada a la información que se ha almacenado. Para ello, diríjase a la base de datos ASPNETDB.MDF que fue almacenada en la carpeta *App_Data* del proyecto web (si no ve esta carpeta, actualice el explorador de soluciones desde su barra de herramientas), haga clic con el botón secundario del ratón y seleccione *Abrir*. El explorador de servidores le mostrará una conexión con la base de datos. Seleccione la tabla que desee y muestre sus datos.

Como siguiente paso vamos a completar el código del proyecto de cliente para que permita autenticar a los usuarios.

Autenticación

El servicio de autenticación WCF que hemos añadido al proyecto de servidor es útil siempre que sea necesario autenticar a los usuarios a través de los servicios de aplicación ASP.NET desde una aplicación que está fuera de la aplicación web, como sucede con la aplicación Silverlight de nuestro ejemplo. La aplicación cliente debe poder enviar y utilizar mensajes en formato SOAP.

La clase **AuthenticationService** contiene cuatro métodos a los que tendremos acceso a través del servicio WCF *AuthenticationService* que añadimos al proyecto de servidor: **IsLoggedIn** determina si el usuario actual está autenticado, **Login**

comprueba las credenciales del usuario y crea una cookie si son válidas, **Logout** borra la cookie en el explorador, y **ValidateUser** autentica las credenciales del usuario sin emitir una cookie. Utilice **ValidateUser** cuando un usuario haya iniciado sesión previamente y deba comprobar que las credenciales siguen siendo válidas al iniciar una nueva sesión de la aplicación.

Para llamar a estos métodos, hay que añadir a la aplicación cliente una referencia al servicio WCF que vamos a denominar *RefAuthenticationSvc*.

Una vez agregada en el cliente la referencia al servicio WCF, el siguiente paso es crear un objeto de la clase *RefAuthenticationSvc.AuthenticationServiceClient* del *proxy* para acceder a los métodos del servicio. Recuerde que todas las llamadas que se realicen al servicio web desde Silverlight son asíncronas, de ahí que tengamos que añadir los controladores de los eventos que se generan cada vez que se completa cada una de las operaciones solicitadas. También tendremos que garantizar que la URL actual del servicio sea la que tiene que ser, independientemente de dónde se instale dicho servicio. Esto lo haremos proporcionando la URL absoluta del servicio en el instante de crear el objeto de la clase *RefAuthenticationSvc.AuthenticationServiceClient*. Según lo expuesto, añada el siguiente código a la clase *MainPage* que da lugar a la página principal:

```
Imports System.Windows.Navigation
Imports ApNavSilverlightSvcAps.RefAuthenticationSvc

Partial Public Class MainPage
  Inherits UserControl

  Private formLogin As New Login()
  Private svcAuthentication As AuthenticationServiceClient = Nothing

  Public Sub New()
    InitializeComponent()
    AddHandler formLogin.Closed, AddressOf formLogin_Closed

    Dim uriSvcAut As New Uri(Application.Current.Host.Source,
                        "../AuthenticationService.svc")
    svc = New AuthenticationServiceClient(
      "BasicHttpBinding_AuthenticationService", uriSvcAut.AbsoluteUri)

    AddHandler svcAuthentication.LoginCompleted,
      AddressOf svcAuthentication_LoginCompleted

    AddHandler svcAuthentication.LogoutCompleted,
      AddressOf svcAuthentication_LogoutCompleted
  End Sub

  Private Sub ContentFrame_Navigated(sender As Object,
```

```
  e As NavigationEventArgs)
    ' ...
  End Sub

  Private Sub ContentFrame_NavigationFailed(sender As Object,
  e As NavigationFailedEventArgs)
    ' ...
  End Sub

  Private Sub hlInicioSesion_Click(sender As Object,
  e As RoutedEventArgs)
    ' ...
  End Sub

  Private Sub formLogin_Closed(sender As Object, e As EventArgs)
    ' ...
  End Sub

  Private Sub svcAuthentication_LoginCompleted(sender As Object,
  e As LoginCompletedEventArgs)

  End Sub

  Private Sub svcAuthentication_LogoutCompleted(sender As Object,
  e As AsyncCompletedEventArgs)

  End Sub
End Class
```

Entonces, para verificar las credenciales habrá que invocar al método *LoginAsync* del servicio, una vez se cierre el formulario de inicio de sesión. Este método tiene cuatro parámetros: el nombre de usuario que se va a validar, la contraseña correspondiente al usuario especificado, el valor o valores, si los hubiera, que se van a validar, además del nombre de usuario y la contraseña, y el valor que indica si la cookie mantiene su validez entre sesiones.

```
Private Sub formLogin_Closed(sender As Object, e As EventArgs)
  If formLogin.DialogResult = True Then
    svcAuthentication.LoginAsync(formLogin.txtUsuario.Text,
      formLogin.txtContraseña.Password, String.Empty, False)
  End If
End Sub
```

Y cuando la operación de verificación de las credenciales finalice, si son correctas, daremos la bienvenida al usuario y cambiaremos el título "Iniciar sesión" por "Cerrar sesión", y si no son correctas, mostraremos un mensaje de error al usuario. Para mostrar un mensaje de error podemos utilizar una ventana secundaria de la clase *ErrorWindow* proporcionada por la aplicación Silverlight.

```
Private Sub svcAuthentication_LoginCompleted(sender As Object,
e As LoginCompletedEventArgs)
  If e.Result Then
    tbSeparador.Visibility = Visibility.Visible
    tbBienvenido.Text = "Bienvenido, " + formLogin.txtUsuario.Text
    hlInicioSesion.Content = "Cerrar sesión"
  Else
    Dim errorWin As ChildWindow = New ErrorWindow("Error:",
      "Las credenciales no son correctas")
    errorWin.Show()
  End If
End Sub
```

Y para cerrar la sesión habrá que invocar al método *LogoutAsync* del servicio una vez el usuario haga clic en "Cerrar sesión":

```
Private Sub hlInicioSesion_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  If hlInicioSesion.Content.ToString() = "Iniciar sesión" Then
    formLogin.Show()
  Else
    tbBienvenido.Text = ""
    tbSeparador.Visibility = Visibility.Collapsed
    hlInicioSesion.Content = "Iniciar sesión"
    svcAuthentication.LogoutAsync()
  End If
End Sub
```

Y cuando la operación de cerrar sesión finalice, simplemente verificaremos si terminó satisfactoriamente o si ocurrió algún error. En este último caso, mostraremos un mensaje de error al usuario por medio de la ventana secundaria de la clase *ErrorWindow*.

```
Private Sub svcAuthentication_LogoutCompleted(sender As Object,
e As AsyncCompletedEventArgs)
  If e.Error IsNot Nothing Then
    Dim s As String = e.Error.Message
    Dim errorWin As ChildWindow = New ErrorWindow("Error:", s)
    errorWin.Show()
  End If
End Sub
```

El proceso de autenticación está finalizado. Pruebe la aplicación y observe los resultados. Después, puede publicarla y observar que todo sigue funcionando correctamente.

Funciones (roles)

Igual que ocurría con el servicio WCF de autenticación, el servicio WCF de funciones de los usuarios (roles) que hemos añadido al proyecto de servidor es útil siempre que tengamos que acceder a esta información desde una aplicación que está fuera de la aplicación web, como sucede con la aplicación Silverlight de nuestro ejemplo.

La clase **RoleService** expone dos métodos, a los que tendremos acceso a través de un servicio WCF, en nuestro caso a través del servicio *RoleService* que añadimos al proyecto de servidor: **GetRolesForCurrentUser**, que devuelve todas las funciones del usuario que ha iniciado sesión, e **IsCurrentUserInRole**, que verifica si el usuario pertenece a la función especificada.

Para llamar a estos métodos, hay que añadir a la aplicación cliente una referencia al servicio WCF, que vamos a denominar *RefRoleSvc*.

Una vez agregada en el cliente la referencia al servicio WCF, el siguiente paso es crear un objeto de la clase *RefRoleSvc.RoleServiceClient* del *proxy* para acceder a los métodos del servicio.

```
Imports ApNavSilverlightSvcAps.RefRoleSvc

Partial Public Class MainPage
  Inherits UserControl
  ' ...
  Private svcRole As RoleServiceClient = Nothing

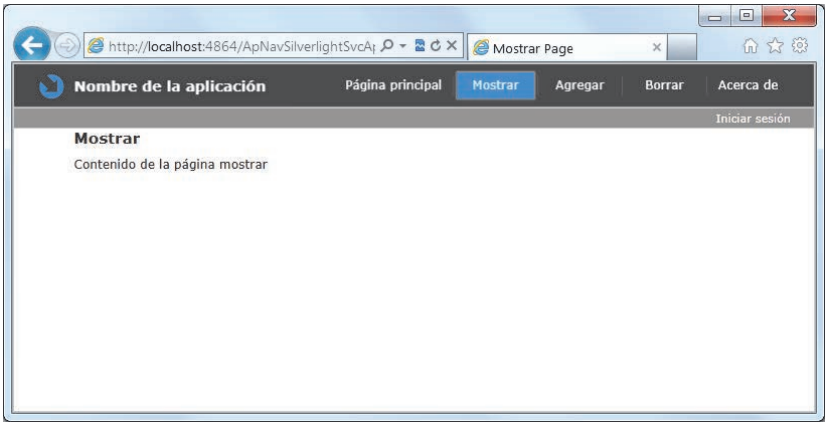
  Public Sub New()
    InitializeComponent()
    ' ...
    Dim uriSvcRol As New Uri(Application.Current.Host.Source,
                             "../RoleService.svc")
    svcRole = New RoleServiceClient("BasicHttpBinding_RoleService",
                                    uriSvcRol.AbsoluteUri)
  End Sub
  ' ...
End Class
```

Para acceder a las funciones del usuario, éste debe estar autenticado.

Después de que el usuario se haya autenticado, el siguiente paso es verificar a qué está autorizado, lo cual estará en consonancia con el papel que se le haya adjudicado a ese usuario al permitirle ejecutar la aplicación. Adjudicar a un usuario uno u otro papel es sinónimo de permitirle o no realizar determinadas operaciones cuando ejecute la aplicación. Por ejemplo, anteriormente, a través del servicio de

suscripciones, creamos para nuestra aplicación ejemplo dos funciones (papeles o roles): "administradores" y "miembros". Esto quiere decir que el administrador de la aplicación puede asignar un usuario que se haya registrado al grupo de "administradores", al grupo de "miembros" o no asignarle a ningún grupo.

¿Cómo administramos estas funciones? Para verlo vamos a añadir al proyecto de cliente, en su carpeta *Views*, tres páginas Silverlight: *Mostrar.xaml*, *Agregar.xaml* y *Borrar.xaml*. Después de añadirlas, las incluimos en el menú que hay implementado en la página *MainPage.xaml*.



Supongamos ahora que la política a seguir cuando un usuario solicite ejecutar la aplicación es que podrá acceder a unas u otras páginas en función de que el usuario esté o no autenticado y del papel que tenga asignado, según especifica la tabla siguiente:

	Principal	*Mostrar*	*Agregar*	*Borrar*	*Acerca de*
Sin autenticar	X				X
Autenticado	X	X			X
Miembro	X	X	X		X
Administrador	X	X	X	X	X

Para gestionar esta política de accesos necesitamos saber si el usuario está o no autenticado y si está autenticado, qué función tiene asignada. Según lo expuesto anteriormente, para saber si un usuario está autenticado, el servicio de autenticación proporciona el método **IsLoggedIn** que devuelve un valor **True**, está autenticado, o **False**, no está autenticado, y para conocer la función del usuario el servicio de funciones proporciona el método **GetRolesForCurrentUser** que devuelve la lista de funciones del usuario.

Según lo expuesto, vamos a añadir a la clase *MainPage*, según muestra el código siguiente, un atributo *usuarioAutenticado* de tipo **Boolean**, que indique si el usuario actual está o no autenticado, y otro *rolesUsuario* de tipo **List(Of String)** que indique cuáles son sus funciones. El valor para el primer atributo lo obtendremos cuando se complete la llamada al método *IsLoggedInAsync* que realizaremos siempre que un usuario haya completado un inicio o cierre de sesión, y el valor para el segundo lo obtendremos cuando se complete la llamada al método *GetRolesForCurrentUserAsync* que realizaremos siempre que un usuario haya completado un inicio de sesión.

```
Partial Public Class MainPage
  Inherits UserControl
  ' ...
  Private usuarioAutenticado As Boolean = False
  Private rolesUsuario As List(Of String) = Nothing

  Public Sub New()
    ' ...
    AddHandler svcAuthentication.IsLoggedInCompleted,
      AddressOf svcAuthentication_IsLoggedInCompleted
    ' ...
    AddHandler svcRole.GetRolesForCurrentUserCompleted,
      AddressOf svcRole_GetRolesForCurrentUserCompleted
  End Sub

  ' ...

  Private Sub svcAuthentication_LoginCompleted(sender As Object,
  e As LoginCompletedEventArgs)
    ' ...
    svcAuthentication.IsLoggedInAsync()
  End Sub

  Private Sub svcAuthentication_LogoutCompleted(sender As Object,
  e As AsyncCompletedEventArgs)
    ' ...
    svcAuthentication.IsLoggedInAsync()
  End Sub

  Private Sub svcAuthentication_IsLoggedInCompleted(
  sender As Object, e As IsLoggedInCompletedEventArgs)
    usuarioAutenticado = e.Result
    If usuarioAutenticado Then
      svcRole.GetRolesForCurrentUserAsync()
    End If
  End Sub

  Private Sub svcRole_GetRolesForCurrentUserCompleted(
  sender As Object, e As GetRolesForCurrentUserCompletedEventArgs)
```

```
    rolesUsuario = e.Result.ToList()
  End Sub

  Private Sub ContentFrame_Navigating(sender As Object,
  e As NavigatingCancelEventArgs)
  End Sub
End Class
```

Los valores obtenidos *usuarioAutenticado* y *rolesUsuario* los utilizaremos para permitir o denegar el acceso a las páginas *Home*, *About*, *Mostrar*, *Agregar* o *Borrar* de la aplicación Silverlight. Si el usuario tiene permiso para acceder a la página solicitada, se admitirá la navegación a esa página y si no, se denegará el acceso y se mostrará un mensaje que lo indique por medio de una ventana hija de la clase *ErrorWindow*. Cada vez que el usuario cambia de una página a otra, se genera, entre otros, el evento **Navigating** del elemento *ContentFrame* de tipo **Frame** de la página principal utilizado para controlar la navegación entre las páginas de la aplicación. Por eso, tales operaciones las realizaremos desde este controlador. Según esto, añada dicho controlador a la clase *MainPage* y complételo como se indica a continuación:

```
Private Sub ContentFrame_Navigating(sender As Object, e As Navi-
gatingCancelEventArgs)
  ' Páginas Home y About
  If e.Uri.OriginalString = "/Home" OrElse
     e.Uri.OriginalString = "/About" Then
    Return
  End If ' continuar
  ' Resto de páginas: Mostrar, Agregar y Borrar
  If Not usuarioAutenticado Then
    AccesoDenegado(e)
    ' Si está autenticado tiene acceso a Mostrar. Ahora:
  ElseIf e.Uri.OriginalString <> "/Mostrar" AndAlso
         rolesUsuario.Count = 0 Then
    AccesoDenegado(e)
  Else
    ' "miembros" o "administradores"
    For Each role As String In rolesUsuario
      ' Ambos pueden Agregar, pero Borrar sólo los administradores
      If e.Uri.OriginalString = "/Borrar" AndAlso
         role <> "administradores" Then
        AccesoDenegado(e)
      End If
    Next
  End If
End Sub

Private Sub AccesoDenegado(e As NavigatingCancelEventArgs)
  Dim window As New ErrorWindow("Acceso denegado",
    "No está autorizado para acceder a esta página")
```

```
  window.Show()
  e.Cancel = True
End Sub
```

Pruebe ahora la aplicación y observe los resultados.

Perfiles

Muchas aplicaciones requieren almacenar y utilizar información que es única de un usuario. De esta forma, cuando el usuario visita el sitio, se puede utilizar esa información para presentar al usuario una versión personalizada de la aplicación web. Pues bien, para facilitar estas tareas se puede utilizar la característica de perfiles de ASP.NET, ya que las realiza de forma automática. Ahora bien, cuando tengamos que acceder a esa información desde una aplicación que está fuera de la aplicación web, como sucede en el ejemplo que estamos realizando, añadiremos al proyecto de servidor el servicio WCF de perfiles, según vimos anteriormente.

La clase **ProfileService** permite obtener y establecer las propiedades de un perfil de usuario a través de un servicio WCF. Esta clase contiene los métodos siguientes: **GetPropertiesForCurrentUser**, que recupera las propiedades y los valores de perfil especificados para el usuario actual, **GetAllPropertiesForCurrentUser**, que recupera todas las propiedades y valores de perfil disponibles para el usuario actual, **SetPropertiesForCurrentUser**, que establece los valores de las propiedades especificadas en el perfil de usuario, y **GetPropertiesMetadata**, que recupera información sobre las propiedades de perfil que están disponibles a través del servicio de perfil.

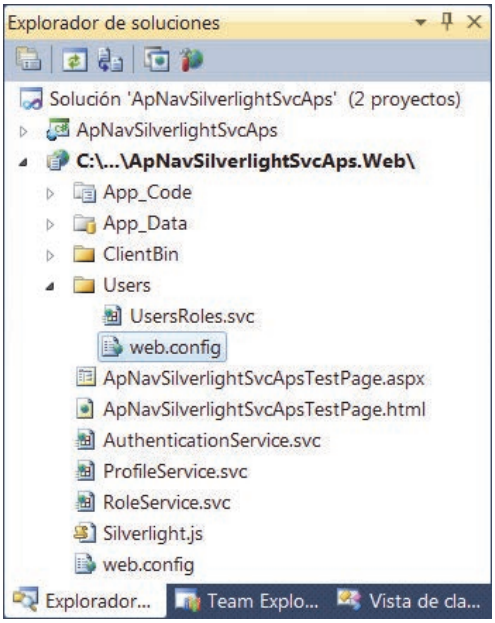
Para llamar a estos métodos, proceda de forma análoga a lo explicado para los servicios WCF de autenticación y de funciones.

Autorización de ASP.NET

Una vez autenticados los usuarios, puede ser importante asegurar que sólo ellos pueden intentar acceder a unos u otros servicios. Los servicios de WCF se representan mediante un fichero *.svc* y, como hemos visto, pueden ser hospedados en IIS vía ASP.NET, lo cual hace posible utilizar carpetas para proteger el acceso a los mismos. Ahora bien, para poder contar con muchas de las características de ASP.NET, hay que habilitar el modo de compatibilidad ASP.NET:

```
<system.serviceModel>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
</system.serviceModel>
```

Los servicios WCF en el modo de compatibilidad de ASP.NET participan en la canalización HTTP de ASP.NET de una manera similar a los servicios ASMX; esto significa que las características de ASP.NET como autorización de archivo, autorización URL y estado de sesión HTTP son aplicables a los servicios que se ejecutan en este modo. Por ejemplo, con la compatibilidad con ASP.NET habilitada, es posible impedir el acceso a uno o más servicios a determinados usuarios mediante la sección de autorización de un fichero *web.config*, incluido en la carpeta donde se guardan los servicios.

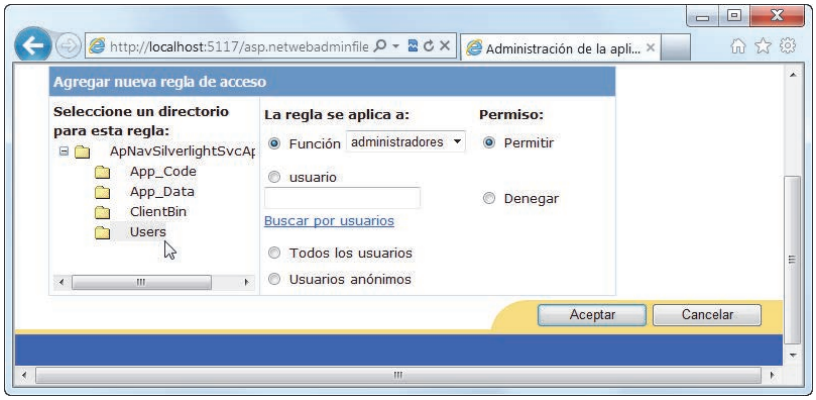


Tenga en cuenta que la raíz de la aplicación debe tener acceso anónimo permitido, de lo contrario los usuarios no podrán acceder al sitio.

Por ejemplo, en la figura anterior se puede ver en el proyecto de servidor una carpeta *Users* con un servicio WCF *UsersRoles* y un fichero de configuración *web.config* que tiene como contenido el código siguiente:

```
<configuration>
    <system.web>
        <authorization>
            <allow roles="administradores" />
            <deny users="*" />
        </authorization>
    </system.web>
</configuration>
```

Este código indica que a la carpeta *Users* sólo pueden acceder los usuarios con la función de "administradores", denegando el acceso a todos los demás usuarios. Este fichero se puede construir desde el asistente del servicio de suscripciones, según muestra la figura siguiente. Primero se selecciona la carpeta para la cual se quieren editar los permisos y después se añaden las reglas una a una. En nuestro ejemplo, según se ve en el código anterior, hemos añadido dos reglas. Cuando se definen varias reglas, se aplicarán en el orden en el que aparecen; se aplicará la primera regla que coincida.



Como ejemplo, vamos a añadir al proyecto de servidor de nuestra aplicación una carpeta *Users*. A continuación, añadimos a esta carpeta un servicio WCF habilitado para Silverlight denominado *UsersRoles*.

Este servicio expondrá un método *Users* que devolverá una lista, obtenida de la base de datos del servicio de suscripciones, con los usuarios y sus funciones, según muestra el código siguiente:

```
<ServiceContract(Namespace:="")>
<SilverLightFaultBehavior()>
<AspNetCompatibilityRequire-
ments(RequirementsMode:=AspNetCompatibilityRequirementsMode.Allowed
)>
Public Class UsersRoles
  <OperationContract()> _
  Public Function Users() As List(Of String)
    Dim ls As List(Of String) = Nothing
    Dim strConexion As String = ConfigurationManager.
      ConnectionStrings("ApplicationServices").ConnectionString
    Dim ConexionConBD As New SqlConnection(strConexion)
    Using ConexionConBD
      ' Crear una consulta
      Dim Consulta As String =
        "SELECT aspnet_Users.UserName, aspnet_Roles.RoleName " &
```

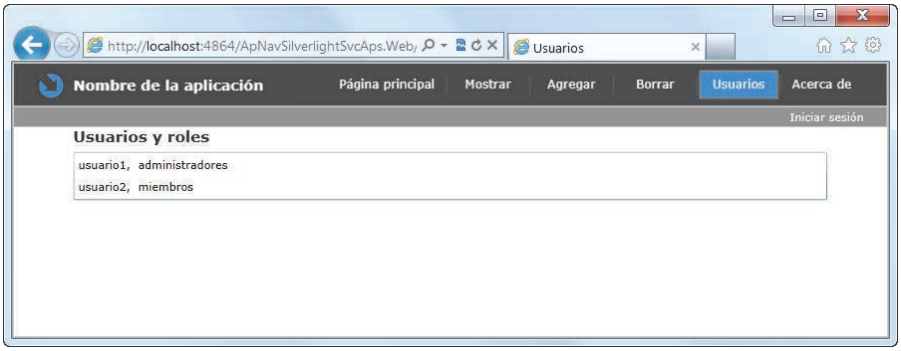
```
        "FROM aspnet_Roles INNER JOIN " &
        "aspnet_UsersInRoles ON aspnet_Roles.RoleId = " &
        "aspnet_UsersInRoles.RoleId INNER JOIN " &
        "aspnet_Users ON aspnet_UsersInRoles.UserId = " &
        "aspnet_Users.UserId"
      Dim OrdenSql As New SqlCommand(Consulta, ConexionConBD)
      ' Abrir la conexión con la base de datos
      ConexionConBD.Open()
      ' ExecuteReader hace la consulta y devuelve un SqlDataReader
      Dim Lector As SqlDataReader = OrdenSql.ExecuteReader()
      ' Llamar siempre a Read antes de acceder a los datos
      ls = New List(Of String)()
      While Lector.Read() ' siguiente registro
        ls.Add(Lector("UserName") + ",  " + Lector("RoleName"))
      End While
      ' Llamar siempre a Close una vez finalizada la lectura
      Lector.Close()
    End Using
    Return ls
  End Function
End Class
```

La consulta realizada no muestra los usuarios que no tienen asignada una función. Esto queda como ejercicio para el lector.

La cadena de conexión *ApplicationServices* para acceder a la base de datos se obtiene del fichero de configuración de la carpeta *Users* utilizando la funcionalidad proporcionada por la clase **ConfigurationManager**.

Una vez finalizado el servicio, añada al proyecto de cliente una referencia al mismo, por ejemplo *RefUsersRolesSvc*, para que sus métodos queden expuestos a la aplicación Silverlight.

Añada una página *Users* al cliente y diséñela como muestra la figura siguiente (muestra una etiqueta y una lista). Añádala al menú. Dé autorización a todos los usuarios para que puedan acceder a esta página, igual que hicimos con las páginas *Home* y *About*. Estamos suponiendo que aún no hemos añadido el fichero de configuración a la carpeta *Users*.



A continuación escriba el siguiente código para esta página:

```
Imports ApNavSilverlightSvcAps.RefUsersRolesSvc

Partial Public Class Users
  Inherits Page

  Public Sub New()
    InitializeComponent()
  End Sub

  ' Se ejecuta cuando el usuario navega a esta página.
  Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
    Dim uriSvcUsersRoles As New Uri(Application.Current.Host.Source,
                                    "../Users/UsersRoles.svc")
    Dim svcUsersRoles As New UsersRolesClient(
      "CustomBinding_UsersRoles", uriSvcUsersRoles.AbsoluteUri)
    AddHandler svcUsersRoles.UsersCompleted,
      AddressOf svcUsersRoles_UsersCompleted
    svcUsersRoles.UsersAsync()
  End Sub

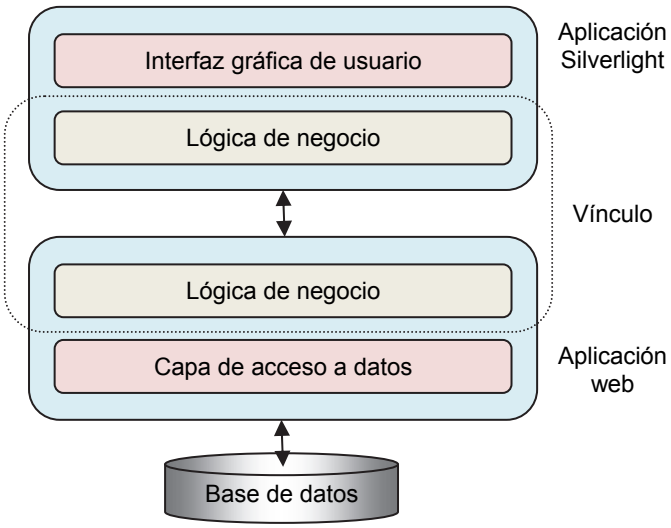
  Private Sub svcUsersRoles_UsersCompleted(sender As Object,
  e As UsersCompletedEventArgs)
    Dim ls As List(Of String) = e.Result.ToList()
    lstUsers.ItemsSource = ls
  End Sub
End Class
```

Si ahora ejecuta la aplicación, observará que cualquier usuario autenticado o no tiene acceso a la página *Users*. Para que sólo tengan acceso los usuarios con la función de "administradores", edite en la carpeta *Users* el fichero de configuración *web.config* indicado anteriormente. Una vez editado, si ejecuta de nuevo la aplicación y un usuario que no tiene la función de "administradores" intenta acceder a la página *Users*, obtendrá un error.

Resumiendo, ésta es otra forma de proteger el acceso a un servicio WCF. Quizás prefiera utilizar la forma explicada en el apartado *Funciones (roles)*, donde lo que hacíamos era denegar el acceso a la página en función de si el usuario estaba o no autenticado y de la función que tenía asignada.

SIMPLIFICAR EL DESARROLLO DE APLICACIONES

Si analizamos la arquitectura de la solución que hemos desarrollado en el apartado anterior, llegaremos a la conclusión de que, en términos generales, está compuesta por los siguientes bloques:

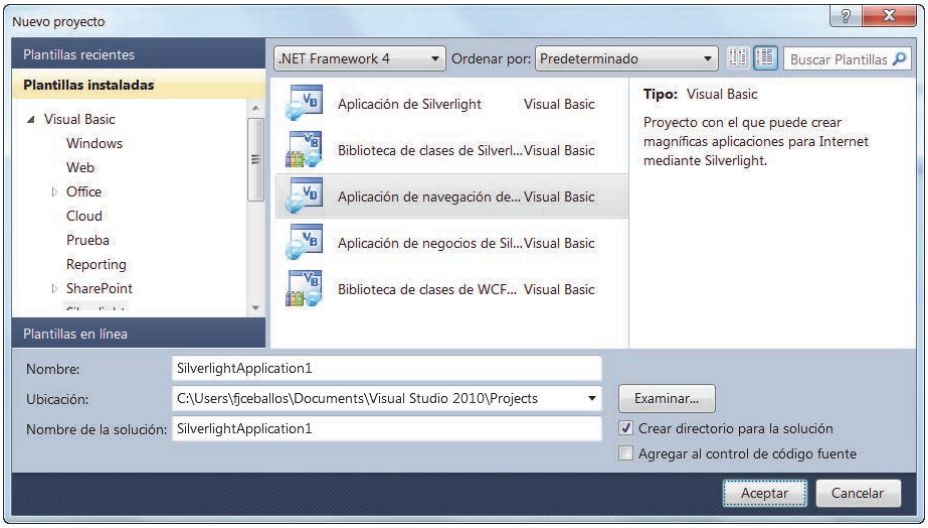


Como vemos, la solución que construimos está formada por una aplicación cliente, la aplicación Silverlight y una aplicación web, y el vínculo entre ambas aplicaciones lo formaba la lógica de negocio desarrollada en ambas partes: los servicios WCF del lado del servidor y el *proxy* creado en el lado del cliente para comunicar con los servicios. Está claro que esta estructura nos obliga a mantener la lógica de negocio que reside en el servidor y la lógica de negocio que reside en el cliente, específica para comunicarse con la lógica de negocio del servidor.

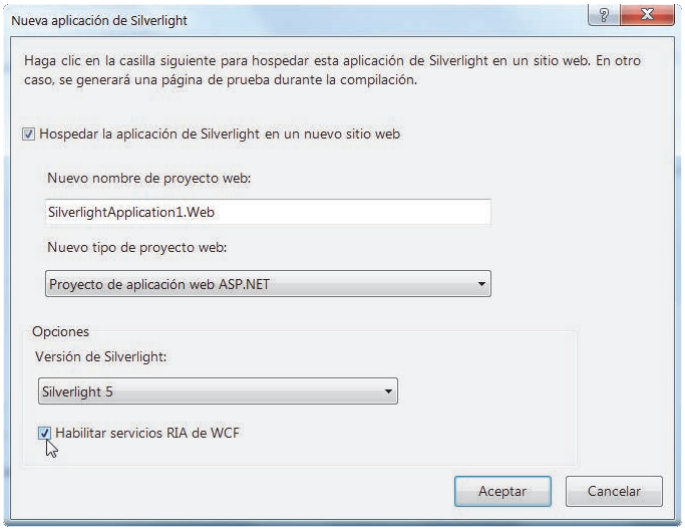
Para dar solución a este problema y simplificar el desarrollo de aplicaciones de Internet enriquecidas (RIA) de N niveles, como son las aplicaciones de Silverlight, ASP.NET pone a nuestra disposición *WCF RIA Services*. Se trata de un marco de trabajo formado por componentes, herramientas y servicios que pone la lógica de negocio del servidor a disposición del cliente de *RIA Services* sin que sea necesario duplicar manualmente tal lógica de negocio, permitiendo que el cliente se actualice automáticamente con la última lógica de nivel intermedio cada vez que se recompila la solución. Esto es, *RIA Services* establece un vínculo (co-

nocido como vínculo a *RIA Services*) entre la aplicación Silverlight y la aplicación web facilitando la generación de código de nivel de presentación a partir del código de nivel intermedio.

¿Cómo se crea un *vínculo a RIA Services*? Cuando se crea una aplicación Silverlight,



se puede crear un vínculo a *RIA Services* activando la casilla *Habilitar WCF RIA Services*, según muestra la figura siguiente. Esta opción sólo está disponible cuando la aplicación web es de tipo *proyecto de aplicación web ASP.NET*.



Recuerde que se pueden crear *proyectos de aplicación web ASP.NET* o *sitios webs ASP.NET*. Cada tipo de proyecto tiene ventajas y desventajas; por ejemplo, en el proyecto, el código fuente se compila explícitamente en el equipo que se usa para el desarrollo generando un único ensamblado, y en el sitio, normalmente, ASP.NET compila el código fuente de forma dinámica en el servidor la primera vez que se recibe una solicitud y la compilación genera múltiples ensamblados. No obstante los comentarios acerca de estas diferencias se salen fuera de los objetivos de este capítulo.

También es posible crear el vínculo a *RIA Services* entre proyectos de biblioteca de clases.

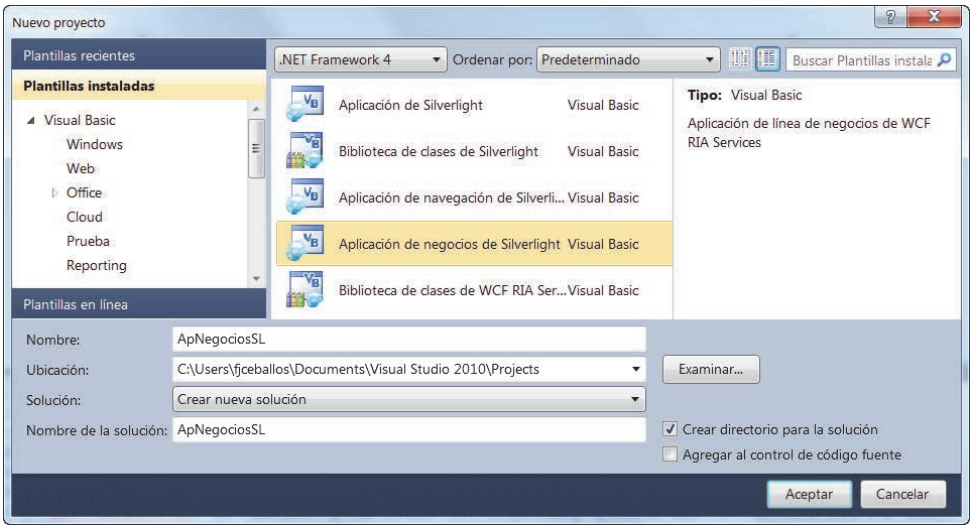
A modo de ejemplo, después de crear la estructura de la aplicación con un vínculo a *RIA Services* podría añadir al proyecto de servidor las clases de ADO.NET Entity para acceder a una base de datos, así como un servicio de dominio que exponga la lógica de negocio al cliente (un servicio de dominio expone las entidades y operaciones con los datos del proyecto de servidor en el proyecto de cliente) o podría agregar lógica de negocio al servicio de dominio para administrar cómo el cliente interactúa con los datos, etc.

Después del trabajo anterior, en el momento de generar la solución se crearán las clases que forman el *proxy* del cliente. Precisamente, esta generación de código es debida al vínculo a *RIA Services* que se estableció entre el proyecto de cliente y el proyecto de servidor.

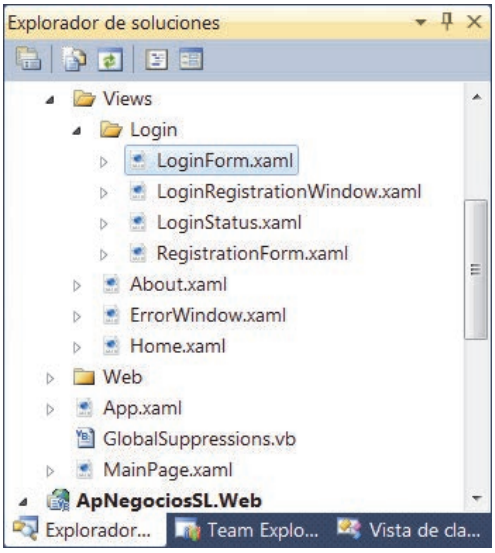
No obstante, todo este trabajo se simplificará si utiliza una plantilla proporcionada por *RIA Services* denominada *Aplicación de negocios de Silverlight*. La plantilla da lugar a una aplicación de navegación de Silverlight y utiliza *RIA Services* para admitir el registro de autenticación y de usuario.

Plantilla aplicación de negocios Silverlight

La plantilla *Aplicación de negocios de Silverlight* crea un proyecto que incluye automáticamente las características estudiadas en los apartados anteriores como autenticación, funciones y perfiles, así como controles para el inicio de sesión de los usuarios y para el registro de nuevos usuarios, que normalmente se desean en una aplicación de negocios. El proyecto creado por esta plantilla también se configura para utilizar la navegación de Silverlight, lo que permite agregar nuevas páginas de Silverlight para añadir a la aplicación funcionalidad adicional. El modo de autenticación predeterminado utilizado por la plantilla es la autenticación mediante formularios, pero se puede cambiar al modo de autenticación de Windows cambiando simplemente un par de valores de configuración. Para verificar lo expuesto, vamos a generar una solución *ApNegociosSL*:



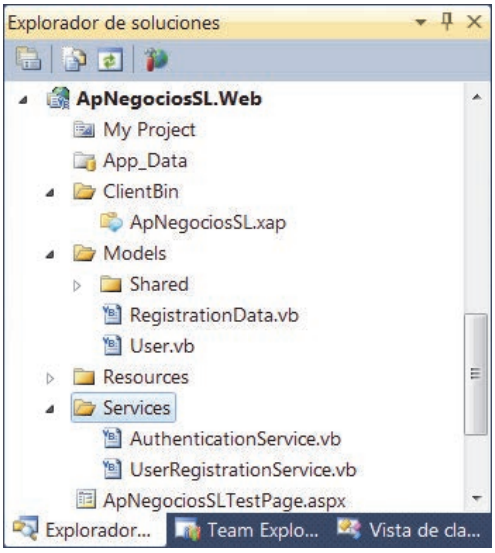
Cuando creamos un proyecto utilizando esta plantilla, *RIA Services* creará la solución con una estructura predeterminada, incluyendo una ventana de inicio de sesión, una ventana de registro y la navegación de Silverlight.



El modo de autenticación predeterminado es la autenticación mediante formularios. Para utilizar la autenticación de Windows, sólo tiene que cambiar el valor del elemento de autenticación en el fichero *web.config* (*<authentication mode="Windows"/>*) y cambiar el valor de la propiedad **Authentication** de la clase **WebContext** generada, derivada de la clase **WebContextBase** del espacio de nombres **System.ServiceModel.DomainServices.Client.ApplicationServices**, en

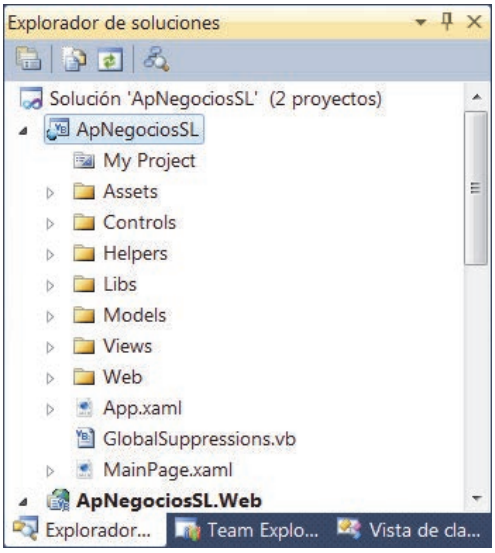
la clase *App* del proyecto de cliente (*webContext.Authentication = new FormsAuthentication()*).

La carpeta *Services* del proyecto de servidor contiene los servicios de dominio que exponen la autenticación y el registro de usuario. El servicio de registro de usuario utiliza los servicios de aplicación de ASP.NET para crear nuevos usuarios.

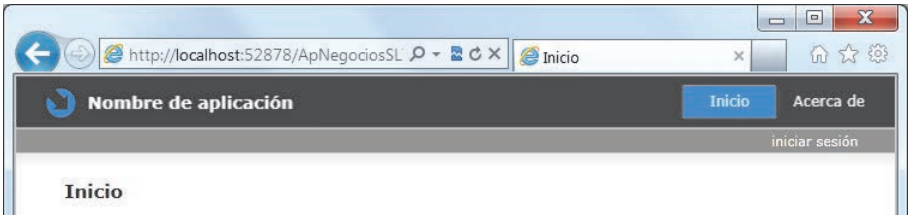


Y la carpeta *Models* contiene las clases para definir propiedades para los datos de registro y de usuario. Puede agregar propiedades a estas clases para personalizar los datos de usuario con los que trabajará la aplicación.

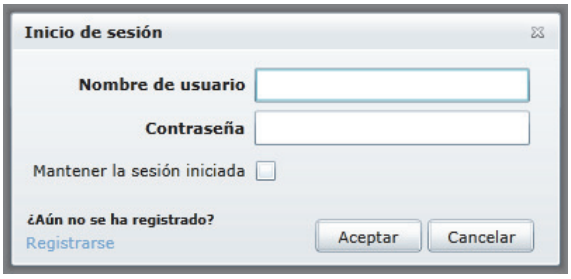
En el proyecto de cliente, la plantilla generó código para simplificar el desarrollo de la capa de presentación. Las carpetas *Controls* y *Libs*, ubicadas dentro de la carpeta *Models*, contienen los ensamblados y controles utilizados por esta plantilla. La carpeta *Helpers* contiene métodos de extensión y convertidores. La carpeta *Login*, ubicada dentro de la carpeta *Views*, contiene los controles de autenticación y de registro de usuario; estos controles se habilitan automáticamente en el proyecto. La carpeta *Resources*, ubicada en la carpeta *Assets*, contiene ficheros de recursos, como cadenas de texto, utilizados en el proyecto; esto es, cuando sea necesario agregar recursos a la aplicación, se puede hacer en estos ficheros.



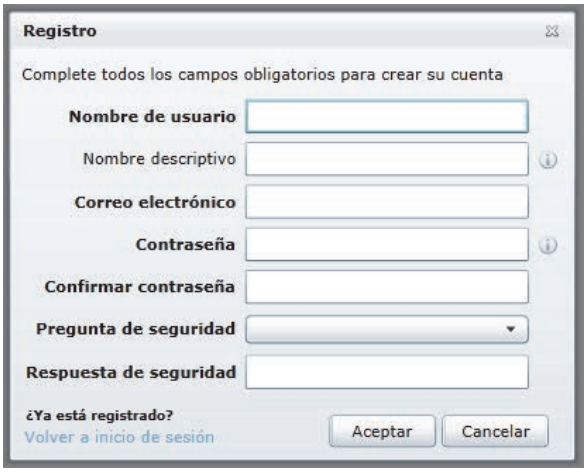
Si ejecuta ahora la aplicación, observará una aplicación de navegación Silverlight con una barra que tiene el elemento de inicio de sesión, construcción análoga a la que hicimos en nuestra anterior aplicación.



Ahora, si hace clic en el elemento "iniciar sesión" observará que se muestra la ventana *Inicio de sesión* que muestra la figura siguiente. Si introduce un nombre de un usuario y su contraseña que no se corresponden con alguno de los registros realizados, la ventana le mostrará un mensaje de error.



La ventana anterior tiene en su parte inferior izquierda un elemento que permitirá a un usuario registrarse, según muestra la figura siguiente. Los usuarios registrados por este medio tienen de forma predeterminada la función *Registered User*, que puede modificar si lo desea.



También puede configurar y editar usuarios, funciones y permisos de acceso para el sitio y administrar los valores de configuración de la aplicación desde el servicio de suscripciones de ASP.NET que expusimos anteriormente en el apartado *Crear usuarios* de este mismo capítulo.

Continuando con el desarrollo de nuestra solución *ApNegociosSL*, el objetivo es reproducir la misma solución que desarrollamos anteriormente en el apartado *Servicios de aplicación de ASP.NET* para poder comparar las ventajas que reporta la utilización de esta plantilla. Después añadiremos el acceso a datos.

Autenticación, funciones y perfiles

Cuando se generó el proyecto de servidor, *ApNegociosSL.Web*, la plantilla que utilizamos añadió un servicio de dominio de autenticación que habilita al proyecto cliente para realizar operaciones con los usuarios que se autentican.

Crear un servicio de dominio de autenticación supone añadir al proyecto de servidor un nuevo elemento de tipo *Servicio de dominio de autenticación* (*DomainService*). El resultado es que se añaden automáticamente dos clases al proyecto de servidor; en nuestro ejemplo, estas clases, *AuthenticationService* y *UserRegistrationService*, fueron añadidas en la carpeta *Services*. La primera representa el servicio de autenticación y se deriva de la clase **AuthenticationBase** y

la segunda representa al usuario y se deriva de la clase **UserBase**. La clase de usuario contiene las propiedades de perfil para un usuario autenticado.

Así mismo, cuando se genera la solución, *RIA Services* genera automáticamente la clase *WebContext*, que se deriva de **WebContextBase**, en el proyecto de cliente para que éste pueda acceder al servicio de dominio de autenticación y al usuario. Esta clase proporciona dos propiedades: *Current* y *User*. La primera es una propiedad **Shared** que representa el objeto actual de *WebContext* (el contexto web) y la segunda no es **Shared** y representa el usuario actualmente autenticado. Por ejemplo, las dos líneas de código siguientes permiten obtener, la primera, el contexto web y, la segunda, el usuario actual:

```
Dim contextoWeb As WebContext = WebContext.Current
Dim usuarioActual As Web.User = contextoWeb.User
```

Normalmente, para obtener el usuario actual utilizaremos la expresión:

```
WebContext.Current.User
```

Además de esas dos propiedades, el contexto web proporciona otra propiedad *Authentication* que permite acceder al contexto de autenticación de la aplicación.

Y la clase *User* proporciona las propiedades *IsAuthenticated* de tipo **Boolean** que indica si el usuario está autenticado, *Roles* que permite acceder a las funciones de las que es miembro el usuario, *Name* que especifica el nombre del usuario, y el método, entre otros, *IsInRole* que devuelve un valor de tipo **Boolean** indicando si el usuario actual pertenece a la función especificada como argumento. Por ejemplo, la línea de código siguiente devuelve **True** si el usuario autenticado tiene la función (el rol) de "administradores":

```
WebContext.Current.User.IsInRole("administradores")
```

También es posible especificar que una operación de dominio determinada solo esté disponible para los miembros de ese rol. La restricción de acceso a una operación de dominio se obtiene aplicando el atributo **RequiresRoleAttribute** a la operación de dominio. Por ejemplo:

```
<RequiresRole("administradores")> _
Public Function ObtenerAsignaturas() As IQueryable(Of asignatura)
    Return Me.ObjectContext.asignaturas
End Function
```

Análogamente, el atributo **RequiresAuthenticationAttribute** especifica que sólo los usuarios con credenciales de autenticación válidas pueden tener acceso a una determinada operación de dominio.

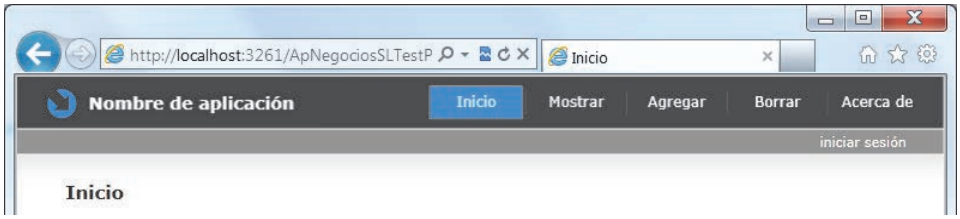
Anteriormente dijimos que la clase de usuario contiene las propiedades de perfil para un usuario autenticado. Dicha clase puede ser personalizada para añadir todas las propiedades que deseemos configuren el perfil del usuario, perfil que puede ser utilizado después para personalizar la aplicación para cada usuario. Para obtener acceso al objeto y a las propiedades que hayamos agregado a la clase *User*, procederemos así:

```
WebContext.Current.User.UnaPropiedad
```

Para trabajar con perfiles, disponemos también de los métodos **LoadUser** y **SaveUser**. El primero permite actualizar el estado del usuario autenticado y el segundo, guardar los cambios en el estado del usuario (por ejemplo, después de establecer el valor de algunas propiedades de su perfil). **SaveUser** tiene un parámetro de tipo **Boolean** para especificar si un error no controlado debe generar una excepción (**True**) o no (**False**). Las siguientes líneas de código indican cómo puede invocar a estos métodos:

```
WebContext.Current.Authentication.LoadUser()
WebContext.Current.Authentication.SaveUser(True)
```

Pues bien, continuando con nuestra aplicación, vamos a añadir al proyecto de cliente, en su carpeta *Views*, tres páginas Silverlight: *Mostrar.xaml*, *Agregar.xaml* y *Borrar.xaml*. Después de añadirlas, las incluimos en el menú que hay implementado en la página *MainPage.xaml*.



Supongamos ahora que la política a seguir cuando un usuario solicite ejecutar la aplicación es que podrá acceder a unas u otras páginas en función de que el usuario esté o no autenticado y del papel que tenga asignado, según especifica la tabla siguiente:

	Principal	*Mostrar*	*Agregar*	*Borrar*	*Acerca de*
Sin autenticar	X				X
Registered User	X	X			X
Miembro	X	X	X		X
Administrador	X	X	X	X	X

Para gestionar esta política de accesos necesitamos saber si el usuario está o no autenticado y si está autenticado, qué función tiene asignada. Según lo expuesto anteriormente, para saber si un usuario está autenticado, la clase *User* proporciona la propiedad *IsAuthenticated* que devuelve un valor **True** si el usuario está autenticado o **False** si no lo está, y para conocer la función del usuario la clase *User* proporciona el método *IsInRole(función)* que devuelve un valor **True** si tiene la función especificada o **False** si no la tiene. Estos valores los utilizaremos para permitir o denegar el acceso a las páginas *Home*, *About*, *Mostrar*, *Agregar* o *Borrar* de la aplicación Silverlight. Si el usuario tiene permiso para acceder a la página solicitada, se admitirá la navegación a esa página y si no, se denegará el acceso y se mostrará un mensaje que lo indique por medio de una ventana hija de la clase *ErrorWindow*. Cada vez que el usuario cambia de una página a otra, se genera, entre otros, el evento **Navigating** del elemento *ContentFrame* de tipo **Frame** de la página principal utilizado para controlar la navegación entre las páginas de la aplicación. Por eso, la política de acceso a las páginas mencionadas la implementaremos desde este controlador. Según esto, añada dicho controlador a la clase *MainPage* y complételo como se indica a continuación:

```
Private Sub ContentFrame_Navigating(sender As Object,
e As NavigatingCancelEventArgs)
  ' Páginas Home y About
  If e.Uri.OriginalString = "/Home" OrElse
     e.Uri.OriginalString = "/About" Then
    Return
  End If ' continuar
  ' Resto de páginas: Mostrar, Agregar y Borrar
  If Not WebContext.Current.User.IsAuthenticated Then
    AccesoDenegado(e)
    ' Si está autenticado tiene acceso a Mostrar. Ahora:
  ElseIf e.Uri.OriginalString <> "/Mostrar" AndAlso
         WebContext.Current.User.IsInRole("Registered Users") Then
    AccesoDenegado(e)
  Else
    ' "miembros" o "administradores"
    ' Ambos pueden Agregar, pero Borrar sólo los administradores
    If e.Uri.OriginalString = "/Borrar" AndAlso
       Not WebContext.Current.User.IsInRole("administradores") Then
      AccesoDenegado(e)
    End If
  End If
End Sub

  Private Sub AccesoDenegado(e As NavigatingCancelEventArgs)
    Dim window As New ErrorWindow("Acceso denegado",
      "No está autorizado para acceder a esta página")
    window.Show()
    e.Cancel = True
  End Sub
```

Pruebe ahora la aplicación y observe los resultados. Comparando el código que hemos necesitado escribir en esta segunda versión de la aplicación con la primera versión que realizamos en el apartado *Servicios de aplicación de ASP.NET*, es evidente que todo se ha reducido a escribir este último método.

RESUMEN

En este capítulo se ha expuesto cómo implementar en una aplicación Silverlight la autenticación de usuarios y la autorización para permitir a un usuario autenticado acceder a un recurso específico.

Para implementar estas características es preciso recordar que Silverlight es una tecnología del lado del cliente, lo cual significa que el cliente donde se ejecuta la aplicación Silverlight se tendrá que comunicar con el servidor para llevar a cabo las operaciones de autenticación y autorización. Por eso, en una primera aproximación creamos en el lado del servidor un servicio que expusiera los métodos de autenticación, apoyándonos en la clase **FormsAuthentication**, y en una segunda mejor aproximación, aprovechando que la ejecución de una aplicación Silverlight se lanza a través de una aplicación ASP.NET, tomamos ventaja al utilizar los servicios de aplicación de ASP.NET: autenticación, función (rol) y perfil del usuario. Cualquier aplicación cliente que pueda comunicarse usando el protocolo SOAP, independientemente del lenguaje y del sistema operativo utilizado, puede tener acceso a estos servicios.

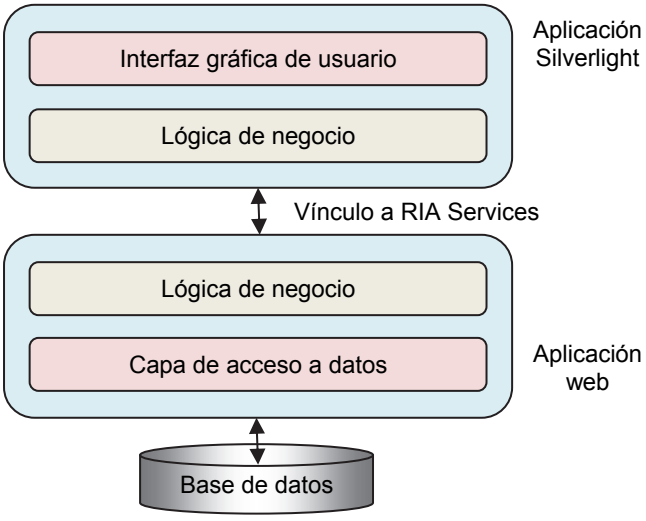
Finalmente, vimos que añadir a una aplicación las características anteriores resulta sencillo si se utiliza la plantilla *Aplicación de negocios de Silverlight*. Esta plantilla crea un proyecto que incluye automáticamente las características mencionadas como autenticación, funciones y perfiles, así como controles para el inicio de sesión de los usuarios y para el registro de nuevos usuarios, que normalmente se desean en una aplicación de negocios.

CAPÍTULO 12

© F.J.Ceballos/RA-MA

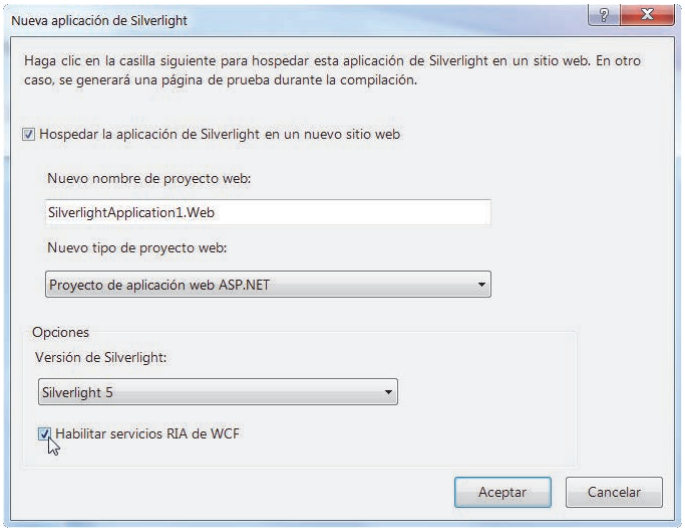
ACCESO A DATOS UTILIZANDO RIA SERVICES

WCF RIA Services es un marco de trabajo formado por componentes, herramientas y servicios que pone la lógica de negocio del servidor a disposición del cliente de *RIA Services* sin que sea necesario duplicar manualmente tal lógica de negocio, permitiendo que el cliente se actualice automáticamente con la última lógica de nivel intermedio cada vez que se recompila la solución. Esto es, *RIA Services* establece un vínculo, conocido como vínculo a *RIA Services*, entre la aplicación Silverlight y la aplicación web facilitando la generación de código de nivel de presentación a partir del código de nivel intermedio.



¿Cómo se crea un *vínculo a RIA Services*? Según vimos en el capítulo *Autenticación y Autorización*, cuando se crea una aplicación Silverlight, se puede crear

un vínculo a *RIA Services* activando la casilla *Habilitar WCF RIA Services*, según muestra la figura siguiente. Esta opción sólo está disponible cuando la aplicación web es de tipo *proyecto de aplicación web ASP.NET*.



También es posible crear el vínculo a *RIA Services* entre proyectos de biblioteca de clases.

ACCESO A DATOS

En este apartado se describe cómo WCF RIA Services trata los escenarios relacionados con el acceso a datos almacenados en una base de datos. Normalmente, para modelar los datos que existen en una base de datos relacional utilizaremos *Entity Data Model*, técnica que ya explicamos y utilizamos en capítulos anteriores. De esta forma, el código que escribamos en el proyecto de cliente para facilitar las operaciones de acceso a datos será indiferente al origen de datos en el sentido de que, realmente, desconoce la tecnología de acceso a datos o el esquema utilizado por el nivel intermedio.

También decir que WCF RIA Services admite la simultaneidad optimista, de la cual ya hablamos en el capítulo *LINQ*, para garantizar la coherencia de los datos. Este modelo de concurrencia se basa en la capacidad de los desarrolladores para proporcionar la lógica necesaria para controlar los conflictos que se puedan producir al actualizar un origen de datos; esto es, si se permite que los usuarios actualicen o eliminen datos, debe asegurarse de que otro proceso no haya cambiado los datos del origen de datos.

También, con las clases de datos, o con sus miembros, presentes en una aplicación de RIA Services, se pueden aplicar los atributos que especifican las reglas de validación, según vimos en el capítulo *Silverlight*.

Para ver todo esto de una forma práctica, vamos a crear una aplicación de RIA Services que acceda a los datos de una base de datos. Para ello vamos a utilizar dos técnicas: una se apoyará en la clase genérica **LoadOperation** y la otra se apoyará en el control **DomainDataSource**, ya que este control permite especificar con muy poco esfuerzo ordenación, filtrado y paginación para el control de presentación de datos.

Crear y configurar la solución

Para crear una nueva aplicación de WCF RIA Services, cree un nuevo proyecto Silverlight utilizando la plantilla *Aplicación de negocios de Silverlight* y asigne a la aplicación, por ejemplo, el nombre *ApRiaServices*. Observe la estructura de la solución que se crea. Tiene dos proyectos: un proyecto de cliente de Silverlight denominado *ApRiaServices* y un proyecto de servidor de aplicación web ASP.NET denominado *ApRiaServices.Web*.

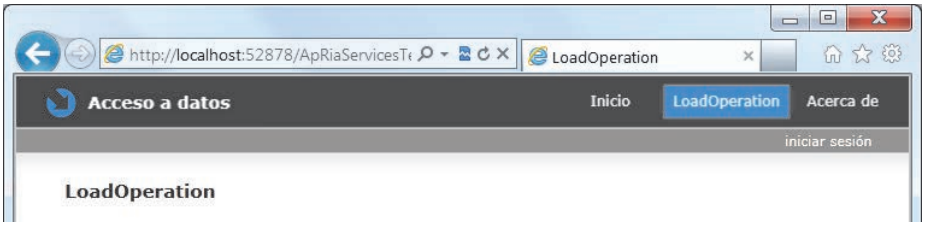
Si ahora explora la solución que se acaba de generar, la compila y la ejecuta, observará que contiene muchas características implementadas automáticamente, como navegación, inicio de sesión, cierre de sesión y registro de nuevos usuarios, según estudiamos en el apartado anterior.

Hagamos ahora alguna operación de configuración. Por ejemplo, vamos a cambiar el título de la aplicación. Dicho título se corresponde con el contenido del **TextBlock** de la página *MainPage.xaml* indicado a continuación:

```
<TextBlock x:Name="ApplicationNameTextBlock"
           Style="{StaticResource ApplicationNameStyle}"
           Text="{Binding Strings.ApplicationName,
           Source={StaticResource ApplicationResources}}"/>
```

Observe que el nombre de aplicación se recupera del recurso *ApplicationResources*. Por lo tanto, para cambiar el título tendremos que modificar este recurso, para lo cual, editaremos el fichero *Assets/Resources/ApplicationStrings.resx* del proyecto de cliente. Abra este fichero y cambie el valor del recurso *ApplicationName* a "Acceso a datos".

A continuación, vamos a añadir al proyecto de cliente, en su carpeta *Views*, una página Silverlight denominada *LoadOperation.xaml*. Después de añadirla, la incluimos en el menú que hay implementado en la página *MainPage.xaml* y la configuramos para que tome el aspecto siguiente:



Mostrar datos utilizando la clase LoadOperation

La aplicación que estamos realizando tiene que acceder a la base de datos SQL Server *bd_notasAlumnos* que utilizamos en el capítulo *LINQ* (notas de los alumnos matriculados en un determinado centro de unas determinadas asignaturas).



Generar el modelo de entidades

Siguiendo el procedimiento explicado en el apartado *Acceso a una base de datos* del capítulo *LINQ*, vamos a añadir al proyecto de servidor, *ApRiaServices.Web*, un nuevo elemento de tipo *ADO.NET Entity Data Model*; lo vamos a denominar *MDEBDNotasAlums.edmx* y denominaremos *ModeloBDNotasAlumnos* al espacio de nombres de las clases que se generen. El resultado será el siguiente:



Una vez añadido el modelo de entidades, genere la solución.

Agregar un servicio de dominio

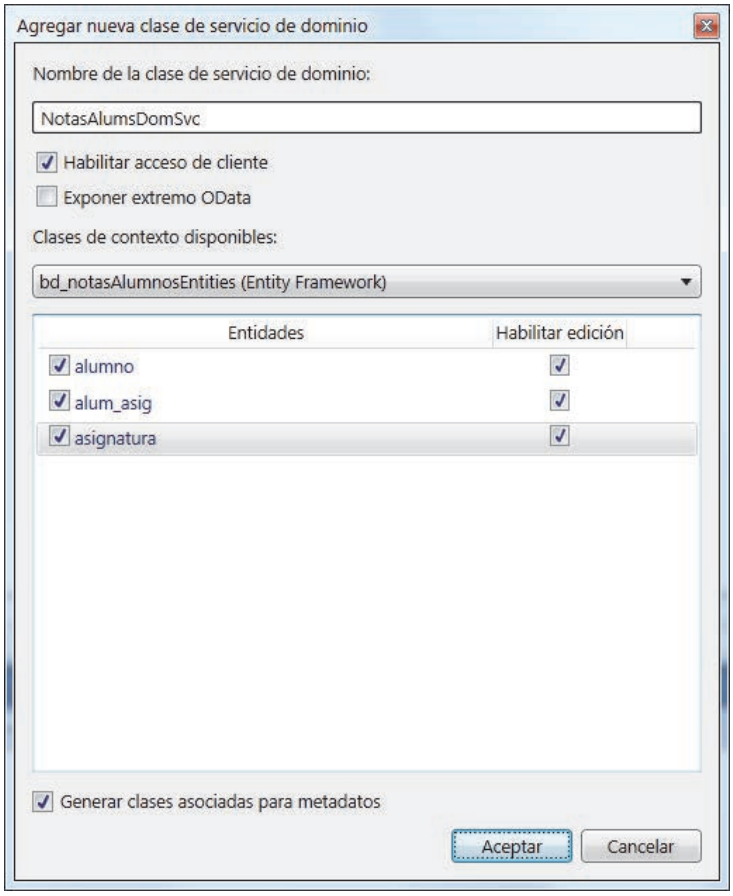
Para consultar los datos desde el cliente vamos a agregar un servicio de dominio que exponga las entidades y muestre esos datos en el proyecto de cliente.

Un servicio de dominio es un servicio WCF que encapsula lógica de negocio de una aplicación que hace uso de RIA Services. Cuando se diseña un servicio de dominio se debe pensar en él como en un conjunto de tareas relacionadas que se espera que los usuarios realicen en la aplicación. Normalmente, dichas tareas afectan a un grupo pequeño de entidades estrechamente relacionadas. Por ejemplo, tareas que afecten a las entidades *alumno*, *asignatura* y *alum_asig*, o conjuntos de ellas, de nuestro modelo de entidades.

Para crear un servicio de dominio que enlaza a un objeto de datos personalizado, debe crear una clase que derive directamente de **DomainService**. Pero si tiene un servicio de dominio que enlaza con un *ADO.NET Entity Data Model*, como es nuestro caso, o que expone un vínculo a una base de datos SQL, puede utilizar en su lugar clases abstractas especiales que derivan de **DomainService**.

En nuestro caso, tenemos que añadir al proyecto de servidor un servicio de dominio que enlace con el modelo *MDEBDNotasAlums* que creamos anteriormente. Este servicio derivará de la clase **LinqToEntitiesDomainService** que proporciona RIA Services. Para ello, haga clic con el botón secundario en el proyecto *ApRiaServices.Web* y seleccione *Agregar > Nuevo elemento > Clase de servicio de dominio* y añada un servicio denominado *NotasAlumsDomSvc*. Cuando haga clic en el botón *Agregar*, se visualizará un diálogo *Agregar nueva clase de servicio de dominio*. Seleccione en este diálogo las entidades *alumno*, *alum_asig* y *asignatura* y habilite la edición para todas. Asegúrese también de que las casillas *Habilitar acceso de cliente* y *Generar clases asociadas para metadatos* están activadas, según muestra la figura siguiente.

Cuando haga clic en el botón *Aceptar* del diálogo *Agregar nueva clase de servicio de dominio*, se añadirán al proyecto de servidor dos ficheros: *NotasAlumsDomSvc.vb* y *NotasAlumsDomSvc.metadata.vb*. El primero implementa la lógica de la aplicación mediante el contexto *bd_notasAlumnosEntities* y el segundo permite adjuntar atributos personalizados a las propiedades de la clase *alum_asig*.



Puede abrir el fichero *NotasAlumsDomSvc.vb* y observar que se han creado los métodos de consulta, inserción, actualización y eliminación para cada entidad. El método de consulta se crea siempre, pero los métodos de inserción, actualización y eliminación se crean sólo si la casilla *Habilitar edición* está seleccionada. No obstante, piense que el código generado por el asistente sólo está previsto para ayudarle a empezar a escribir la lógica de aplicación.

Ahora puede personalizar los métodos existentes, o bien agregar métodos nuevos al servicio de dominio que realicen la operación de datos que desee exponer, además de los ya implementados. Por ejemplo, puede agregar métodos que realicen las operaciones: **Query**, **Update**, **Insert** y **Delete**, o incluso, puede agregar operaciones más complicadas. Por ejemplo, si echa una ojeada al fichero *NotasAlumsDomSvc.vb*, observará que el servicio de dominio ya expone el método siguiente:

```
Public Function GetAlumnos() As IQueryable(Of alumno)
  Return Me.ObjectContext.alumnos
```

```
End Function
```

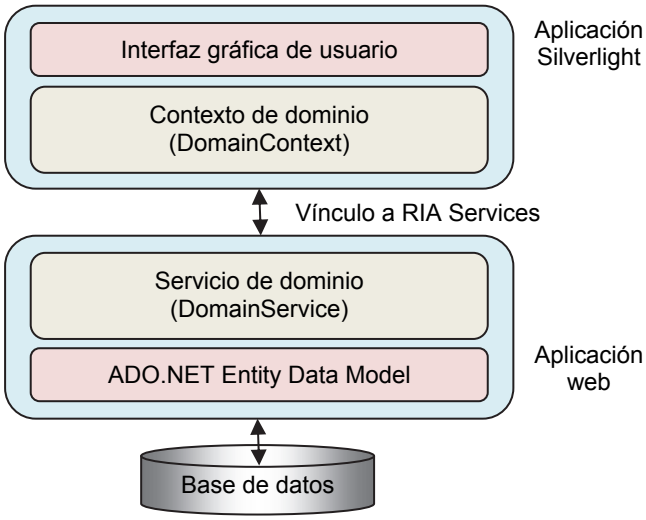
pero nosotros podemos añadir otro (o modificar el anterior) que devuelva la lista de alumnos ordenada:

```
Public Function GetAlumnosOrd() As IQueryable(Of alumno)
  Return Me.ObjectContext.alumnos.OrderBy(Function(e) e.nombre)
End Function
```

El método agregado debe ser público, coincidir con la firma esperada para esa operación, y sus parámetros y el valor devuelto deben ser de tipos seriables. Así mismo, se debe especificar un nombre único para cada método que se pueda invocar desde el proyecto de cliente; esto es, no se puede sobrecargar (para más detalles, vea la ayuda proporcionada por MSDN).

También es posible evitar que un método se exponga agregando al mismo el atributo **IgnoreAttribute**.

Cuando se genere de nuevo la solución, se generarán el contexto de dominio y las entidades en el proyecto de cliente. Desde un punto de vista del código generado, podemos decir que el contexto de dominio es al cliente como el servicio de dominio es al servidor.



Por ejemplo, puede echar una ojeada al fichero *ApRiaServices\Generated_Code\ApRiaServices.Web.g.vb* (no está incluido en el proyecto; utilice el botón *Mostrar todos los archivos* del explorador de soluciones para acceder al mismo) y observar el código generado:

```
' Clase de entidad 'alumno'.
```

```
<DataContract([Namespace]:="...")>  _
Partial Public NotInheritable Class alumno
    Inherits Entity
'  ...
End Class

'  ...

' Contexto de dominio correspondiente al servicio de domino
' 'NotasAlumsDomSvc'.
Partial Public NotInheritable Class NotasAlumsDomSvc
    Inherits DomainContext
  '  ...
  Public ReadOnly Property alumnos() As EntitySet(Of alumno)
    Get
      Return MyBase.EntityContainer.GetEntitySet(Of alumno)
    End Get
  End Property
  '  ...
End Class
```

El paso siguiente es ver cómo utilizamos la infraestructura actual para mostrar datos en la interfaz gráfica que crearemos sobre la página *LoadOperation.xaml*.

LoadOperation

Recuerde que todas las llamadas que se realicen al servicio web desde Silverlight tienen que ser asíncronas. WCF RIA Services soluciona este requisito proporcionando la clase genérica **LoadOperation(Of** *TEntity***)**. Entonces, lo que tenemos que saber es que un objeto de esta clase representa una operación asincrónica de carga y que el método **Load** de un contexto de dominio (objeto derivado de **DomainContext**) inicia una operación de carga para la consulta especificada como argumento y devuelve un objeto **LoadOperation(Of** *TEntity***)**.

Por ejemplo, el siguiente código muestra un objeto **LoadOperation** utilizado para obtener el conjunto de entidades que es asignado al contexto de datos de un control: en un caso a un **DataGrid** y en otro a un **TextBlock** (para entender el código, asuma que el resultado de la consulta es una sola entidad *alum_asig*):

```
Imports ApRiaServices.Web
Imports System.ServiceModel.DomainServices.Client

Private Sub dgAsignaturas_SelectionChanged(sender As Object,
e As SelectionChangedEventArgs)
  '  ...
  Dim contextoDom As New NotasAlumsDomSvc()
  Dim opDeCarga As LoadOperation(Of alum_asig) = contextoDom.Load(
    contextoDom.GetNotaByIDQuery(idAlumnoActual, asig.id_asignatura))
```

```
  AddHandler opDeCarga.Completed, AddressOf opDeCarga_Completed
  Me.dataGrid1.ItemsSource = opDeCarga.Entities
End Sub

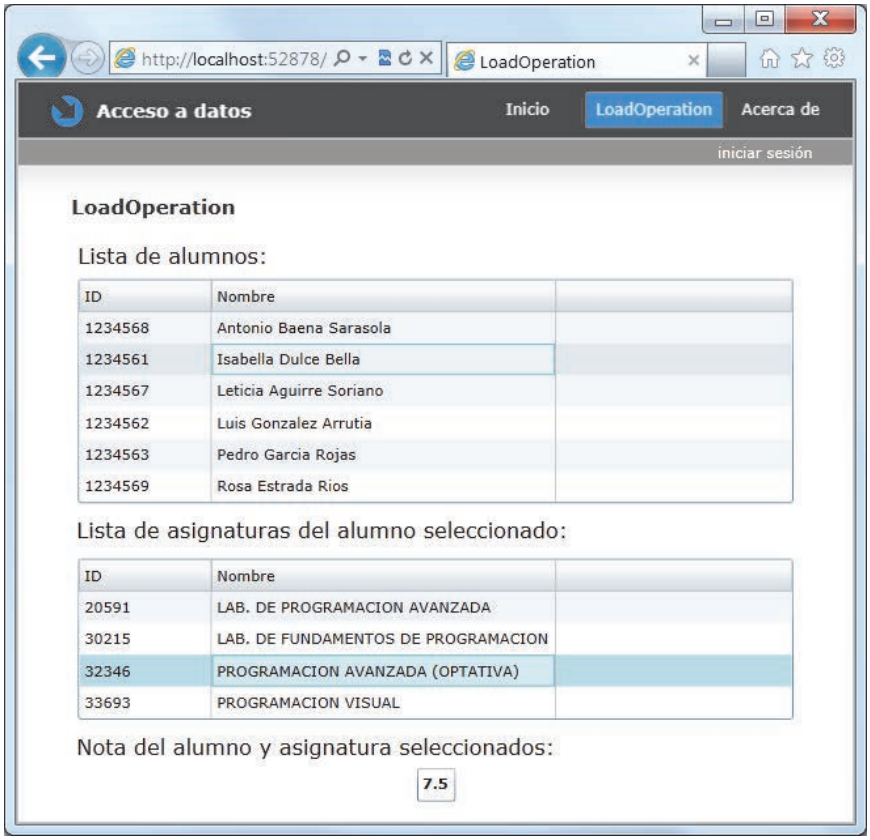
Private Sub opDeCarga_Completed(sender As Object, e As EventArgs)
   Me.textBlock1.DataContext = opDeCarga.Entities.First()
End Sub
```

En el código anterior primero se crea un objeto, *contextoDom*, del contexto de dominio correspondiente al servicio de dominio *NotasAlumsDomSvc* para después invocar a su método **Load** e iniciar la operación de carga para la consulta *GetNotaByIDQuery* especificada, mediante una llamada asíncrona. El resultado es el objeto *opDeCarga* de la clase *LoadOperation(Of alums_asigs)*; su propiedad **Entities** hace referencia al conjunto de entidades de tipo *alum_asig* obtenido, en este caso una, la correspondiente a los identificadores pasados como argumento cuando se invocó a *GetNotaByIDQuery*. Cuando la operación se completa, se genera el evento **Completed** de **LoadOperation**. En el ejemplo hemos utilizado el controlador de este evento para obtener la primera entidad (y única, en este caso) del conjunto de entidades *opDeCarga.Entities*. Pruebe a obtener esta entidad inmediatamente después de la llamada a **Load** y comprobará que el conjunto de entidades aún está vacío porque la llamada asíncrona aún no se ha completado.

Una vez que sabemos cómo tenemos que actuar para acceder a los datos, vamos a diseñar la página *LoadOperation.xaml* para que presente el aspecto que muestra la figura que se muestra un poco más adelante.

La idea es mostrar una interfaz que permita a un alumno acceder a la base de datos *bd_notasAlumnos* para conocer la nota obtenida en cualquiera de las asignaturas en las que está matriculado.

Para diseñar la interfaz gráfica puede echar una ojeada a la aplicación análoga que diseñamos en el apartado *Una aplicación con interfaz gráfica* del capítulo *LINQ*.



Básicamente esta interfaz tiene dos **DataGrid** para mostrar las listas, *dgAlumnos* y *dgAsignaturas*, y un **TextBlock**, *tbNota*, para mostrar la nota. Por ejemplo, el código XAML que da lugar a la primera lista es el siguiente. Observe que las columnas ya están enlazadas con las propiedades respectivas de las entidades que van a mostrar, en este caso, entidades de la clase *alumno*.

```
<!-- Lista de alumnos -->
<TextBlock FontSize="16" Margin="5,10,0,0"
           Text="Lista de alumnos:"/>
<sdk:DataGrid Name="dgAlumnos" Margin="5" Height="Auto"
    AutoGenerateColumns="False" IsReadOnly="True"
    SelectionChanged="dgAlumnos_SelectionChanged">
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn Header="ID"
    Binding="{Binding Path=id_alumno}" MinWidth="100"/>
    <sdk:DataGridTextColumn Header="Nombre"
    Binding="{Binding Path=nombre}" MinWidth="260"/>
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

La segunda lista estará vinculada con entidades de tipo *asignatura* y el control **TextBlock** con una entidad de la clase *alum_asig*.

A la vista de la interfaz anterior, el alumno seleccionará su nombre en la primera lista, lo que hará que se muestren en la otra lista las asignaturas de las que está matriculado. Después, seleccionará la asignatura de la cual quiere conocer la nota y ésta le será mostrada en el bloque de texto.

Ya tenemos generado el modelo de objetos representación de la base de datos, esto es, la capa de acceso a datos (DAL), la lógica de negocio basada en el servicio de dominio que añadimos en el lado del servidor (para interaccionar con la capa de acceso a datos), el contexto de dominio (que generó RIA Services en el lado del cliente a partir del servicio de dominio) para interaccionar con el servicio de dominio, y diseñada la interfaz gráfica. Ahora, partiendo del contexto de dominio (objeto de la clase *NotasAlumsDomSvc* derivada de **DomainContext** en el lado del cliente), escribiremos el código necesario en *LoadOperation.xaml.vb* para que la interfaz gráfica se comporte como hemos pensado.

Continuando con el desarrollo de la aplicación, cuando el usuario navegue a la página *LoadOperation.xaml* la primera lista tendrá que mostrar los identificadores y nombres de los alumnos. Para ello, complete el método **OnNavigatedTo**, que se ejecuta cada vez que el usuario navega a esta página, como se indica a continuación:

```
Protected Overrides Sub OnNavigatedTo(ByVal e As
System.Windows.Navigation.NavigationEventArgs)
  Dim contextoDom As New NotasAlumsDomSvc()
  Dim opDeCarga As LoadOperation(Of alumno) =
    contextoDom.Load(contextoDom.GetAlumnosOrdQuery())
  Me.dgAlumnos.ItemsSource = opDeCarga.Entities
End Sub
```

Analicemos el código para entender mejor lo que estamos haciendo. La llamada *contextoDom.GetAlumnosOrdQuery()* es la consulta que se desea ejecutar; se trata de un objeto de tipo **EntityQuery(Of** *alumno***)**. Esta consulta se pasa como argumento en la llamada al método **Load** para ejecutar la consulta y obtener los resultados. **Load** devuelve un objeto de tipo **LoadOperation(Of** *alumno***)** cuya propiedad **Entities** obtiene todas las entidades que se cargaron cuando se completó la consulta. Este conjunto de entidades obtenido será el origen de datos del **DataGrid** que hemos denominado *dgAlumnos*. Según lo expuesto, el código anterior sería equivalente al siguiente:

```
Protected Overrides Sub OnNavigatedTo(ByVal e As
System.Windows.Navigation.NavigationEventArgs)
    Dim contextoDom As New NotasAlumsDomSvc()
```

```
    Dim consulta As EntityQuery(Of alumno) =
      From alum In contextoDom.GetAlumnosQuery()
      Order By alum.nombre
      Select alum
    contextoDom.Load(consulta)
    Me.dgAlumnos.ItemsSource = contextoDom.alumnos
End Sub
```

Para entender esta versión, además de echar una ojeada a los métodos *GetAlumnosOrd* y *GetAlumnos* del servicio de dominio, piense dónde se almacena el resultado de la consulta que inicia el método **Load**. Pues este resultado, dependiendo del tipo de entidades a obtener en la consulta, poblará el conjunto de entidades, *alumnos*, *asignaturas* o *alums_asigs*, correspondiente del contexto de dominio. De ahí que hayamos asignado el conjunto de entidades *contextoDom.alumnos* a la propiedad **ItemsSource** del **DataGrid**. Según esto, otra versión del método anterior puede ser la siguiente:

```
Protected Overrides Sub OnNavigatedTo(ByVal e As
System.Windows.Navigation.NavigationEventArgs)
    Dim contextoDom As New NotasAlumsDomSvc()
    contextoDom.Load(contextoDom.GetAlumnosOrdQuery())
    Me.dgAlumnos.ItemsSource = contextoDom.alumnos
End Sub
```

Si ejecuta ahora la aplicación, observará cómo la ventana muestra ya la lista de alumnos.

Continuemos con la aplicación. Cuando el usuario seleccione un nombre en la primera lista, en la segunda lista deben mostrarse las asignaturas de la tabla *asignaturas* de las que ese alumno se ha matriculado. ¿Cómo obtenemos esas asignaturas y las notas correspondientes? Pues, igual que hicimos en el capítulo *LINQ*, a través de la siguiente expresión de consulta:

```
Dim alumAsigs =
  From al_as In ObjectContext.alums_asigs
  Where al_as.id_alumno = idAlumno
  Select New With {
    al_as.asignatura.id_asignatura,
    al_as.asignatura.nombre,
    al_as.nota}
```

Pero esta consulta presenta un problema y es que genera un tipo anónimo de entidades con tres propiedades: *id_asignatura*, *nombre* y *nota* y no hay ninguna posibilidad de convertir este tipo en alguno de los definidos en el modelo de objetos o en el contexto de dominio. Lo que vamos a hacer es dividir esa consulta en dos: la primera obtendrá las asignaturas (entidades *asignatura*) de las que está ma-

triculado el alumno *idAlumno* y la segunda, la entidad *alum_asig* correspondiente al alumno *idAlumno* y a la asignatura *idAsignatura*.

Según lo expuesto, el origen de datos para la segunda lista será el conjunto de entidades de tipo *asignatura* que se obtiene al realizar la consulta siguiente:

```
Public Function GetAsignaturasAlum(idAlumno As Integer) As _
IQueryable(Of asignatura)
  Dim alumAsigs =
    From al_as In ObjectContext.alums_asigs
    From asig In ObjectContext.asignaturas
    Where al_as.id_alumno = idAlumno AndAlso
          al_as.id_asignatura = asig.id_asignatura
    Select asig

  Return alumAsigs
End Function
```

Y el origen de datos para el **TextBlock** será el conjunto de entidades de tipo *alum_asig* (en este caso el conjunto tiene una sola entidad) que se obtiene al realizar la consulta siguiente:

```
Public Function GetNotaAlumAsig(idAlumno As Integer,
idAsignatura As Integer) As IQueryable(Of alum_asig)
  Dim notaAsig =
    From al_as In ObjectContext.alums_asigs
    Where al_as.id_alumno = idAlumno AndAlso
    al_as.id_asignatura = idAsignatura
    Select al_as

  Return notaAsig
End Function
```

Añada estos dos métodos al servicio de dominio *NotasAlumsDomSvc* implementado en el lado del servidor. Después, compile la solución para que RIA Services genere de nuevo el contexto de dominio en el lado del cliente.

Entonces, para que el **DataGrid** *dgAsignaturas* muestre los datos *id_asignatura* y *nombre* de cada una de las entidades del conjunto de entidades obtenido por medio del método *GetAsignaturasAlum* del servicio de dominio, tenemos que asignar dicho conjunto a la propiedad **ItemsSource** del **DataGrid**, y los atributos *id_ asignatura* y *nombre* de cada entidad a sus respectivas columnas en el **DataGrid**.

```
<sdk:DataGrid Name="dgAsignaturas" Margin="5" Height="Auto"
    AutoGenerateColumns="False" IsReadOnly="True"
    SelectionChanged="dgAsignaturas_SelectionChanged" >
```

```
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn Header="ID"
      Binding="{Binding Path=id_asignatura}" MinWidth="100"/>
    <sdk:DataGridTextColumn Header="Nombre"
      Binding="{Binding Path=nombre}" MinWidth="260"/>
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

Observe también que en la consulta *GetAsignaturasAlum*, hay un parámetro *idAlumno* que se corresponde con el identificador del alumno seleccionado y que será obtenido de la fila seleccionada en el **DataGrid** *dgAlumnos*.

Ahora bien, el conjunto de entidades al que nos hemos referido en el párrafo anterior cambia para cada alumno seleccionado. Según esto, ¿cuándo asignamos ese conjunto a la propiedad **ItemsSource** de *dgAsignaturas*? Pues cada vez que el usuario seleccione un nuevo nombre en el **DataGrid** *dgAlumnos*. ¿Y dónde hacemos esa asignación? Pues en el controlador del evento **SelectedChanged** del **DataGrid** *dgAlumnos*. Por lo tanto, añada este método en la página *LoadOperation* y complételo así:

```
Private Sub dgAlumnos_SelectionChanged(sender As System.Object, e
As System.Windows.Controls.SelectionChangedEventArgs)
  If e.AddedItems.Count = 0 Then Return
  Dim alum As alumno = TryCast(e.AddedItems(0), alumno)
  If alum Is Nothing Then Return

  Dim contextoDom As New NotasAlumsDomSvc()
  contextoDom.Load(
    contextoDom.GetAsignaturasAlumQuery(alum.id_alumno))
  Me.dgAsignaturas.ItemsSource = contextoDom.asignaturas

  idAlumnoActual = alum.id_alumno
End Sub
```

Este código requiere que defina *idAlumnoActual* de tipo **Integer** como atributo de la clase *LoadOperation*.

Ahora el **DataGrid** *dgAsignaturas* ya muestra el nombre de las asignaturas en las que se ha matriculado el alumno seleccionado. Pues bien, a continuación, cuando el usuario seleccione una de las asignaturas mostradas, se tiene que visualizar la nota correspondiente a la misma en el **TextBlock** *tbNota* que hemos colocado al final de la página. La acción de seleccionar un elemento en el **DataGrid** *dgAsignaturas* hace que se genere el evento **SelectedChanged**. Entonces, vamos a utilizar el controlador de este evento para obtener la entidad *alum_asig*, correspondiente al alumno y asignatura seleccionados, por medio del método *GetNotaAlumAsig* del servicio de dominio. El identificador del alumno seleccionado lo

proporciona el atributo *idAlumnoActual* de la clase *LoadOperation* y el identificador de la asignatura seleccionada lo obtenemos de la fila seleccionada en la lista de asignaturas. Por lo tanto, añada este método en la página *LoadOperation* y complételo así:

```
Private Sub dgAsignaturas_SelectionChanged(sender As System.Object,
e As System.Windows.Controls.SelectionChangedEventArgs)
  If e.AddedItems.Count = 0 Then Return
  Dim asig As asignatura = TryCast(e.AddedItems(0), asignatura)
  If asig Is Nothing Then Return

  Dim contextoDom As New NotasAlumsDomSvc()
  Dim opDeCarga As LoadOperation(Of alum_asig) =
    contextoDom.Load(
      contextoDom.GetNotaAlumAsigQuery(idAlumnoActual,
                                       asig.id_asignatura))
  AddHandler opDeCarga.Completed, AddressOf opDeCarga_Completed
End Sub
```

Este método, primero realiza la consulta *GetNotaAlumAsigQuery* que obtendrá el conjunto de entidades *opDeCarga.Entities*; este conjunto de entidades tiene una sola entidad *alum_asig*, la correspondiente a los identificadores de alumno y asignatura pasados como argumento en la consulta. Ahora bien, según explicamos anteriormente, esa entidad no está presente en el conjunto mientras no se complete la consulta, cosa que ocurrirá cuando *opDeCarga* genere su evento **Completed**. De ahí que hayamos utilizado el controlador de este evento para obtener dicha entidad y asignarla al contexto de datos del bloque de texto, según muestra el método siguiente:

```
Private Sub opDeCarga_Completed(sender As Object, e As EventArgs)
  Dim opDeCarga As LoadOperation(Of alum_asig) =
    TryCast(sender, LoadOperation(Of alum_asig))
  Me.tbNota.DataContext = opDeCarga.Entities.First()
End Sub
```

Entonces, para que el control *tbNota* muestre la nota proporcionada por la entidad *alum_asig* asignada a su contexto de datos, tenemos que asignar la propiedad *nota* de esa entidad con la propiedad **Text** de este control.

```
<TextBlock Name="tbNota"
  Text="{Binding Path=nota}"
  HorizontalAlignment="Center" FontWeight="Bold" />
```

DomainDataSource

Para simplificar la interacción entre la interfaz de usuario y los datos accedidos desde un contexto de dominio, WCF RIA Services proporciona el control **Do-**

mainDataSource. Este control permite obtener datos, modificarlos y darlos forma, utilizando solamente código XAML.

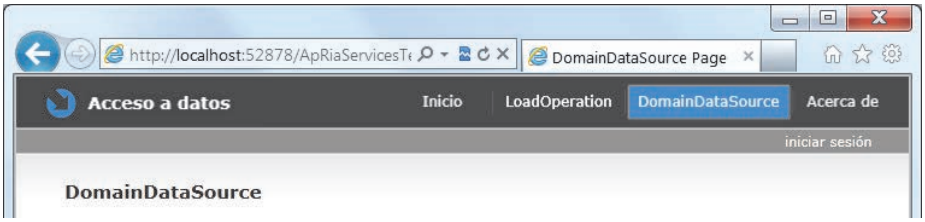
Básicamente, un control **DomainDataSource** debe especificar el contexto de dominio que se va a utilizar y la operación del contexto que se va a ejecutar. En el ejemplo siguiente podemos ver cómo la propiedad **DomainContext** del control ha sido iniciada con el contexto de dominio *NotasAlumsDomSvc* y la propiedad **QueryName** con la consulta *GetAlumnosQuery* a realizar.

```
<riaControls:DomainDataSource Name="ddsAlumnos" AutoLoad="True"
                              QueryName="GetAlumnosQuery" >
  <riaControls:DomainDataSource.DomainContext>
    <ds:NotasAlumsDomSvc />
  </riaControls:DomainDataSource.DomainContext>
</riaControls:DomainDataSource>
```

El código anterior requiere añadir el espacio de nombres XML, denominado en este ejemplo *ds*, que haga referencia al espacio de nombres del CLR al que pertenece la clase que define el contexto de dominio.

```
xmlns:ds="clr-namespace:ApRiaServices"
```

Para ver cómo trabaja este control, vamos a añadir al proyecto de cliente, en su carpeta *Views*, otra página Silverlight denominada *DomainDataSource.xaml*. Después de añadirla, la incluimos en el menú que hay implementado en la página *MainPage.xaml* y la configuramos para que tome el aspecto siguiente:



A continuación, sobre esta página, vamos a reproducir la interfaz gráfica de la página *LoadOperation*, pero utilizando un objeto **DomainDataSource**.

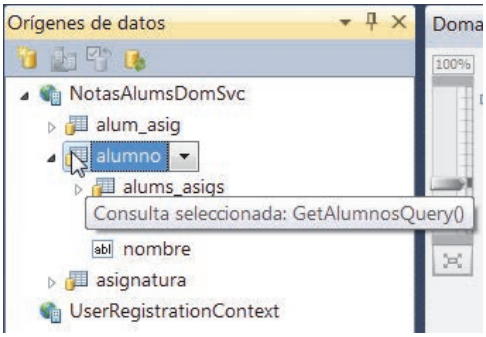
Una vez que hayamos copiado el código XAML y el código subyacente de una página a otra, vamos a añadir a la página *DomainDataSource* un control **DomainDataSource**. Arrastre desde la caja de herramientas un control de este tipo, lo llamaremos *ddsAlumnos*, y configúrelo según muestra el código anterior. Si el control no está en la caja de herramientas, haga clic con el botón secundario del ratón en la cabecera de alguno de los paneles de componentes Silverlight y añádalo. El hecho de arrastrar el control sobre la página ha agregado al proyecto Silver-

light una referencia al ensamblado **System.Windows.Controls.DomainServices** que define el control.

Este control será ahora el proveedor de datos para la propiedad **ItemsSource** del **DataGrid** *dgAlumnos*, con lo que podemos prescindir del código que implementamos en el método *OnNavigatedTo*. Por lo tanto, configure este **DataGrid** para que su origen de datos sea el control *ddsAlumnos*, según muestra el código siguiente:

```
<TextBlock FontSize="16" Margin="5,10,0,0"
           Text="Lista de alumnos:"/>
<sdk:DataGrid Name="dgAlumnos" Margin="5" Height="Auto"
    AutoGenerateColumns="False" IsReadOnly="True"
    ItemsSource="{Binding Data, ElementName=ddsAlumnos}"
    SelectionChanged="dgAlumnos_SelectionChanged">
  <sdk:DataGrid.Columns>
    ...
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

El trabajo realizado puede automatizarlo si desde el panel *Orígenes de datos* que muestra la figura siguiente, arrastra sobre la página el nodo *alumno*.



El resto del código, XAML y código C# subyacente, lo dejamos como está, puesto que el resto de los controles cambian dinámicamente en función de la selección que realice el usuario en las listas.

Parámetros de consulta

Según hemos visto, un método de consulta puede tener parámetros, por ejemplo, para filtrar los datos devueltos. Estos parámetros pueden ser proporcionados al control **DomainDataSource** a través de su propiedad **QueryParameters**. Por ejemplo:

```
<riaControls:DomainDataSource.QueryParameters>
```

```
  <riaControls:Parameter ParameterName="id"
                         Value="{Binding ElementName=comboBox1,
                                   Path=SelectedItem.Content}" />
</riaControls:DomainDataSource.QueryParameters>
```

Ordenar, filtrar y agrupar

El control **DomainDataSource** proporciona la colección **SortDescriptors** para facilitar la ordenación de los datos. Para ello, hay que agregar a dicha colección los elementos **SortDescriptor** que describan los valores que se utilizarán para la ordenación de la misma. Por ejemplo:

```
<riaControls:DomainDataSource.SortDescriptors>
  <riaControls:SortDescriptor PropertyPath="nombre"
                              Direction="Ascending" />
</riaControls:DomainDataSource.SortDescriptors>
```

También proporciona la colección **FilterDescriptors** que permite filtrar los datos que devuelve la consulta. Por ejemplo:

```
<riaControls:DomainDataSource.FilterDescriptors>
  <riaControls:FilterDescriptor PropertyPath="nota"
        Operator="IsGreaterThanOrEqualTo" Value="5" />
</riaControls:DomainDataSource.FilterDescriptors>
```

La colección **GroupDescriptors** facilita la agrupación de los datos por valores de propiedad. Para ello, hay que añadir a dicha colección elementos **GroupDescriptor** que definan los valores utilizados para realizar la agrupación. Por ejemplo:

```
<riaControls:DomainDataSource.GroupDescriptors>
    <riaControls:GroupDescriptor PropertyPath="valor" />
</riaControls:DomainDataSource.GroupDescriptors>
```

Paginación

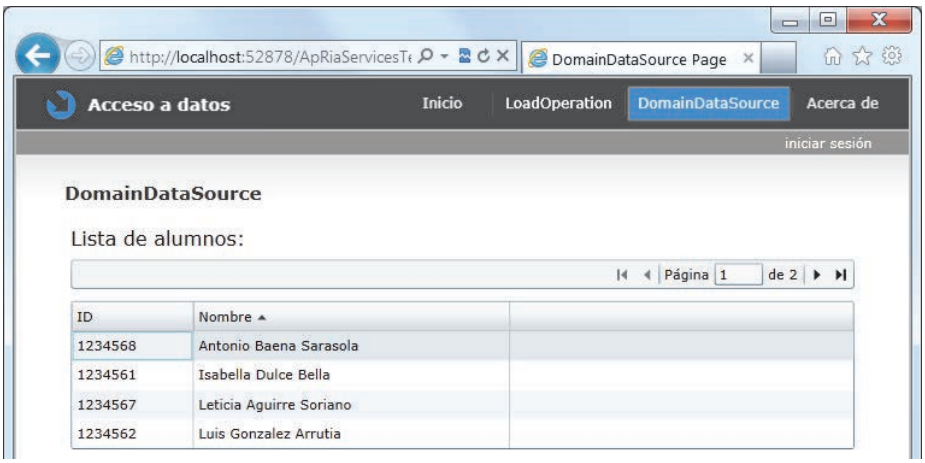
Cuando trabajemos con un gran número de entidades es posible que deseemos mostrarlas en un número de ellas; es lo que se conoce como paginación. Pues bien, el control **DomainDataSource** proporciona la propiedad **LoadSize** para especificar el número de entidades que se desea cargar cada vez que se ejecuta **Load** y la propiedad **PageSize** para especificar el número de entidades que se desea mostrar en la página. Una vez establecidas estas propiedades, hay que utilizar un control **DataPager** para implementar la interfaz de paginación.

Como ejemplo, el siguiente código permitirá mostrar la lista de alumnos de 4 en 4 utilizando un control **DataPager**.

```
<riaControls:DomainDataSource Name="ddsAlumnos "AutoLoad="True"
                              QueryName="GetAlumnosQuery"
                              LoadSize="16"
                              PageSize="4">
  ...
</riaControls:DomainDataSource>

<!-- Lista de alumnos -->
<TextBlock FontSize="16" Margin="5,10,0,0"
           Text="Lista de alumnos:"/>
<sdk:DataPager Margin="5"
               Source="{Binding Data, ElementName=ddsAlumnos}" />
<sdk:DataGrid Name="dgAlumnos" Margin="5" Height="Auto"
  ...
</sdk:DataGrid>
```

El resultado de aplicar el código anterior sería el siguiente:

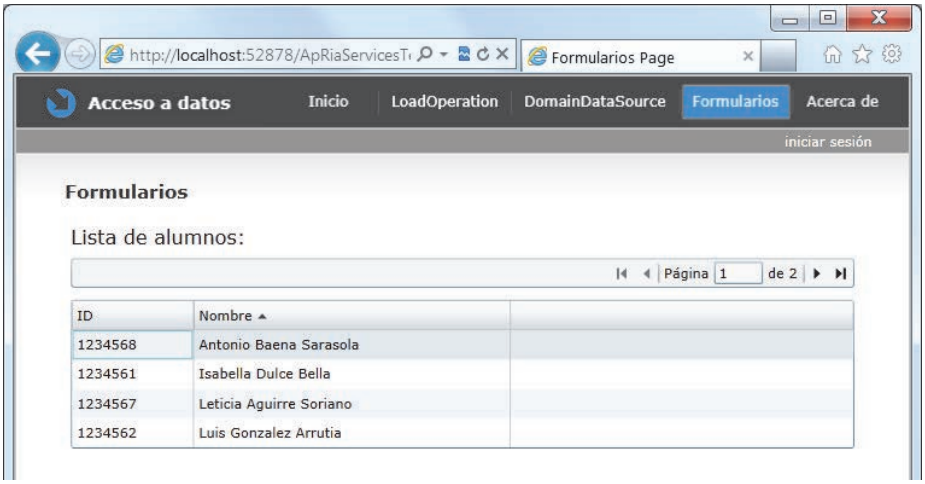


Actualizar la base de datos

Permitir a un usuario trabajar con grupos de datos relacionados es un requisito de casi todas las aplicaciones; por ejemplo, mostrar la lista de alumnos o editar los detalles de un alumno. Los grupos de datos generalmente se presentan mediante formularios, y **DataForm** de Silverlight, del kit de herramientas de Silverlight, es justamente eso, un formulario (de forma predeterminada, la plantilla de proyecto *Aplicación de negocios de Silverlight* contiene la biblioteca *System.Windows.Controls.Data.DataForm.Toolkit.dll* en la carpeta *Libs*).

Para ver cómo trabaja este control, vamos a añadir al proyecto de cliente, en su carpeta *Views*, otra página Silverlight denominada *Formularios.xaml*. Después de añadirla, la incluimos en el menú que hay implementado en la página *MainPa-*

ge.xaml y la configuramos para que tome el aspecto siguiente (sólo incluimos el **DataGrid** *dgAlumnos* con paginación para hacer la explicación más sencilla):



Para agregar el control **DataForm** abrimos el fichero *Formularios.xaml* y lo primero que hacemos es añadir el espacio de nombres XML, que denominaremos *df*, que haga referencia al espacio de nombres del CLR *System.Windows.Controls.Data.DataForm.Toolkit*.

```
xmlns:df="clr-namespace:System.Windows.Controls;
          assembly=System.Windows.Controls.Data.DataForm.Toolkit"
```

A continuación, añadimos, después del control **DataGrid**, el código XAML siguiente correspondiente al control **DataForm**:

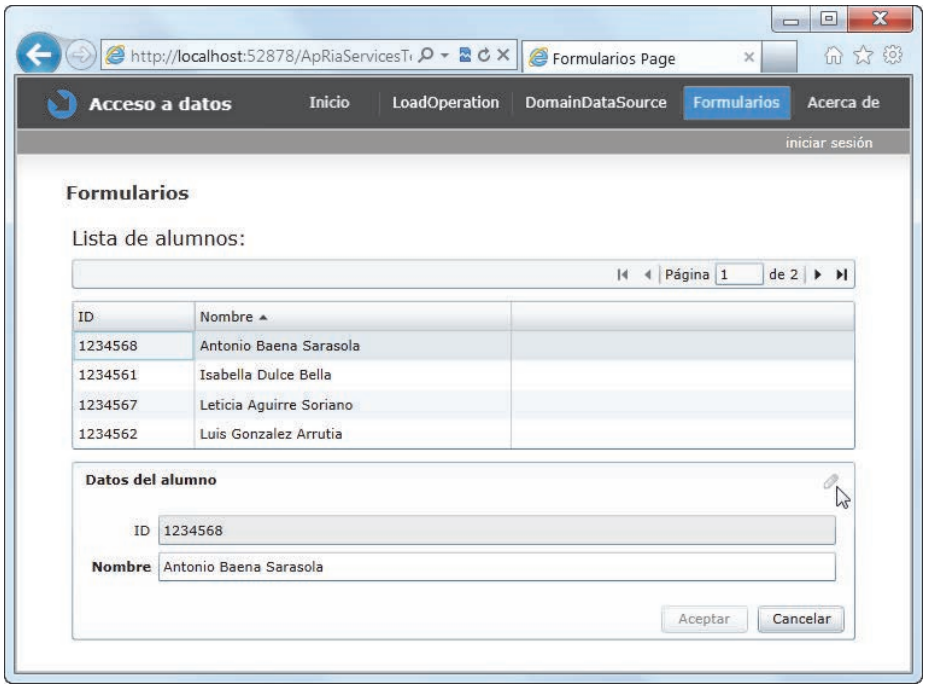
```
<df:DataForm Margin="5"
             ItemsSource="{Binding Data, ElementName=ddsAlumnos}"/>
```

Si ejecuta ahora la aplicación y navega a la página *Formularios*, observará cómo el **DataForm** muestra los detalles de la fila seleccionada del **DataGrid**. Pero, también observamos que hay columnas, como *alums_asigs*, que no deseamos mostrar, por lo que vamos a personalizar el diseño de este control así:

```
<df:DataForm x:Name="dfAlumnos" Header="Datos del alumno"
       AutoGenerateFields="False" Margin="5"
       AutoEdit="False" AutoCommit="False"
       CommitButtonContent="Aceptar" CancelButtonContent="Cancelar"
       CurrentItem="{Binding SelectedItem, ElementName=dgAlumnos}">
  <df:DataForm.EditTemplate>
    <DataTemplate>
      <StackPanel>
```

```
        <df:DataField Label="ID">
          <TextBox Text="{Binding id_alumno, Mode=OneWay}" />
        </df:DataField>
        <df:DataField Label="Nombre">
          <TextBox Text="{Binding nombre, Mode=TwoWay}" />
        </df:DataField>
      </StackPanel>
    </DataTemplate>
  </df:DataForm.EditTemplate>
</df:DataForm>
```

El resultado que produce este código puede verlo en la figura siguiente:



A continuación se expone un breve resumen de algunas de las propiedades de este control:

- **AutoGenerateFields**. Indica si los campos a mostrar se generan automáticamente.
- **AutoEdit**. Indica si el modo de edición se activa directamente o indirectamente haciendo clic en un icono de lápiz (observe la figura anterior).
- **AutoCommit**. Indica si los datos se graban al cambiar de registro sin necesidad de hacer clic en el botón *Aceptar*.
- **CancelButtonContent**. Especifica el texto que se muestra en el botón destinado a cancelar los cambios.

- **CommandButtonsVisibility**. Indica qué botones se van a mostrar en la barra de herramientas del control (hay que especificarlos: *None*, *Edit*, *Add*, *Delete*, *Commit*, *Cancel*, *Navigation* y *All*).
- **CommitButtonContent**. Especifica el texto que se muestra en el botón destinado a confirmar los cambios.
- **CurrentItem**. Se muestra el elemento actualmente seleccionado.
- **DescriptionViewerPosition**. Posición de la descripción.
- **Header**. Título del control.
- **ItemsSource**. Especifica la colección origen de los datos.
- **LabelPosition**. Posición de las etiquetas de los campos.

Así mismo, la clase **DataForm** implementa, entre otros, los métodos **BeginEdit**, **EndEdit** y **CancelEdit** que se invocarán, si es necesario, en los momentos adecuados mientras se está editando el formulario de datos (eche una ojeada a la interfaz **IEditableObject** en el apartado *Desacoplar la IU del resto de la aplicación* del capítulo *Acceso a una base de datos*). Con este comportamiento se consigue crear un objeto temporal de la clase origen de los datos en el método **BeginEdit** y copiar en él los valores originales del objeto vinculado. El método **EndEdit** borra ese objeto temporal, mientras que el método **CancelEdit** restaura el objeto vinculado a su estado original.

Pruebe ahora a ejecutar la aplicación y a cambiar el nombre del alumno. Observará que todo funciona perfectamente.

También es posible aplicar validación de datos a los campos que se muestran. Esto se hace según se explicó en el apartado *Validación de datos* del capítulo *Introducción a WPF*. Por ejemplo:

```
<df:DataField Label="Nota">
  <TextBox Text="{Binding nota, Mode=TwoWay,
    NotifyOnValidationError=True,
    ValidatesOnExceptions=True }"  />
</df:DataField>
```

O también, según se explicó en el apartado *Atributos de anotación de datos* del capítulo *Silverlight*. Como ejemplo, eche una ojeada al fichero *NotasAlumsDomSvc.metadata.vb* en el proyecto de servidor:

```
// Por ejemplo, lo siguiente marca la propiedad Xyz como una
// propiedad requerida y especifica el formato de los valores válidos:
[Required]
[RegularExpression("[A-Z][A-Za-z0-9]*")]
[StringLength(32)]
public string Xyz { get; set; }
```

Veamos ahora cómo salvar en la base de datos los cambios realizados. Anteriormente, cuando añadimos al proyecto de servidor el servicio de dominio que enlaza con el modelo *ADO.NET Entity Data Model* dijimos que los métodos de inserción, actualización y eliminación se creaban sólo si la casilla *Habilitar edición* de la ventana *Agregar nueva clase de servicio de dominio* estaba seleccionada. Y esto fue lo que hicimos, por lo tanto tenemos estas operaciones habilitadas.

Entonces, para actualizar los cambios que hayamos hecho en la lista de alumnos, vamos a añadir a la página *Formularios* un botón "Guardar cambios" justo a continuación del control **DataForm**:

```
<StackPanel Orientation="Horizontal" Margin="0,15,0,0">
  <Button x:Name="guardarCambios" Content="Guardar cambios"
          Width="150" Height="23" Margin="5,0,0,0"
          Click="guardarCambios_Click"/>
</StackPanel>
```

Evidentemente, los cambios realizados en los elementos del **DataGrid** a través del control **DataForm** son realizados automáticamente en los orígenes de datos y por lo tanto, en las entidades vinculadas (observe, por una parte, los objetos **Binding** y por otra, sepa que las entidades como *alumno*, *asignatura* y *alum_asig* se derivan de la clase **Entity** y esta clase implementa la interfaz **INotifyPropertyChanged**; para más detalles repase el apartado *Enlace de datos con el motor de WPF* del capítulo *Introducción a WPF*). Entonces, sólo nos queda saber cómo enviar esos cambios a la base de datos.

Para enviar los cambios a la base de datos, el control **DomainDataSource** tiene un método denominado **SubmitChanges**. Cuando este método se ejecuta los cambios de todas las entidades del contexto de dominio (**DomainContext**) son enviados a la base de datos. Y, ¿cuándo invocamos a este método? Pues cuando el usuario haga clic en el botón *guardarCambios*. Añadimos entonces el controlador para este botón y lo editamos así:

```
Private Sub guardarCambios_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  guardarCambios.IsEnabled = False

  ddsAlumnos.SubmitChanges()
End Sub
```

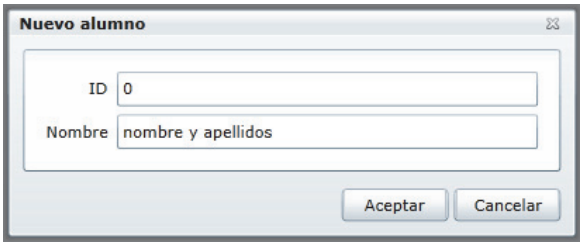
Cuando se complete la operación de envío, el control **DomainDataSource**, en nuestro caso *ddsAlumnos*, generará el evento **SubmittedChanges**. Entonces, para saber cómo finalizó la operación de enviar los datos podemos añadir el controlador para este evento y editarlo como se indica a continuación:

```
Private Sub ddsAlumnos_SubmittedChanges(sender As System.Object,
e As System.Windows.Controls.SubmittedChangesEventArgs)
  If e.HasError Then
    Dim window As New ErrorWindow("Error:",
      String.Format(
        "Error, los cambios no fueron salvados: {0}",
        e.[Error].Message))
    window.Show()
    e.MarkErrorAsHandled()
  End If
  guardarCambios.IsEnabled = True
End Sub
```

Observe que cuando **HasError** valga **true**, el método **MarkErrorAsHandled** marca el error como controlado. Si no se llamara a este método, se lanzaría una excepción.

Añadir nuevos registros

En este apartado, vamos a explicar cómo añadir un nuevo registro a la tabla *alumnos* de la base de datos. Esta operación la vamos a realizar recogiendo los datos por medio de una ventana. Según esto, vamos a añadir al proyecto de cliente, en su carpeta *Views*, una ventana secundaria de Silverlight denominada *VentanaNuevoAlumno*. Después de añadirla, abra el fichero *VentanaNuevoAlumno.xaml* y realice el diseño siguiente:



Para realizar este diseño, añada un control **DataForm** antes del botón *Cancelar* de la ventana secundaria, lo cual requiere que añada el espacio de nombres XML correspondiente al espacio del CLR *System.Windows.Controls.Data.DataForm.Toolkit*, según vimos anteriormente.

```
<df:DataForm x:Name="dfNuevoAlumno"
             AutoGenerateFields="False" AutoCommit="True"
             AutoEdit="True" CommandButtonsVisibility="None">
  <df:DataForm.EditTemplate>
    <DataTemplate>
      <StackPanel>
        <df:DataField Label="ID">
```

```
          <TextBox Text="{Binding Id, Mode=TwoWay,
            NotifyOnValidationError=True,
            ValidatesOnExceptions=True}" />
        </df:DataField>
        <df:DataField Label="Nombre">
          <TextBox Text="{Binding NomAlumno, Mode=TwoWay}" />
        </df:DataField>
      </StackPanel>
    </DataTemplate>
  </df:DataForm.EditTemplate>
</df:DataForm>
```

Observe que las cajas de texto del **DataForm** están vinculadas a las propiedades *Id* y *NomAlumno*. Estas propiedades corresponden a un objeto cuya clase *CAlumno* vamos a escribir justo antes de la clase de la ventana secundaria, esto es, en el mismo fichero *VentanaNuevoAlumno.xaml.vb* que ésta:

```
Public Class CAlumno
  Public Property Id() As Integer
  Public Property NomAlumno() As String

  Public Sub New()
    Id = 0
    NomAlumno = "nombre y apellidos"
  End Sub
End Class

Partial Public Class VentanaNuevoAlumno
  Inherits ChildWindow
  ' ...
End Class
```

¿Por qué no hemos vinculado las cajas de texto del formulario directamente con las propiedades *id_alumno* y *nombre* de la entidad *alumno*? Porque al ser *id_alumno* la clave de la tabla *alumnos*, el formulario muestra este campo en un control de sólo lectura. Otra solución podría haber sido haber creado la base de datos con este campo clave autoincremental.

Definamos entonces el objeto *CAlumno* que va a ser el origen de datos del formulario como una propiedad de la clase *VentanaNuevoAlumno*. Después, en el constructor de esta clase, creamos este objeto y lo asignamos a la propiedad **CurrentItem** del **DataForm**, y en los controladores de los botones *Aceptar* y *Cancelar*, controlamos la confirmación o la cancelación de la inserción.

```
Public Class CAlumno
  ' ...
End Class
```

```
Partial Public Class VentanaNuevoAlumno
  Inherits ChildWindow
  Public Property Nuevo_alumno() As CAlumno

  Public Sub New()
    InitializeComponent()
    Nuevo_alumno = New CAlumno()
    dfNuevoAlumno.CurrentItem = Nuevo_alumno
    dfNuevoAlumno.BeginEdit()
  End Sub

  Private Sub OKButton_Click(ByVal sender As Object,
  ByVal e As RoutedEventArgs) Handles OKButton.Click
    dfNuevoAlumno.CommitEdit()
    Me.DialogResult = True
  End Sub

  Private Sub CancelButton_Click(ByVal sender As Object,
  ByVal e As RoutedEventArgs) Handles CancelButton.Click
    Nuevo_alumno = Nothing
    dfNuevoAlumno.CancelEdit()
    Me.DialogResult = False
  End Sub
End Class
```

Observe en el código anterior la utilización de los métodos **BeginEdit**, **CommitEdit** y **CancelEdit**, que comentamos anteriormente. Su utilización sería necesaria si el objeto enlazado fuera una entidad *alumno* del conjunto *alumnos*, pero al tratarse de un objeto *CAlumno* aislado, no es necesaria su utilización; esto quiere decir que esas llamadas han sido puestas en este ejemplo por motivos puramente didácticos.

Ya tenemos implementada la ventana secundaria para solicitar los valores del nuevo alumno al usuario de la aplicación. Esta ventana será mostrada cuando el usuario haga clic en un botón titulado "Agregar alumno" que colocaremos en la página *Formularios.xaml* junto al botón *guardarCambios*:

```
<StackPanel Orientation="Horizontal" Margin="0,15,0,0">
  <Button x:Name="guardarCambios" ... />
  <Button x:Name="agregarAlumno" Content="Agregar alumno"
          Width="150" Height="23"  Margin="5,0,0,0"
          Click="agregarAlumno_Click"/>
</StackPanel>
```

El controlador del evento **Click** de este botón creará la ventana, establecerá el evento **Closed** de la misma y la mostrará:

```
Private Sub agregarAlumno_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs)
  Dim ventana As New VentanaNuevoAlumno()
  AddHandler ventana.Closed, AddressOf ventana_Closed
  ventana.Show()
End Sub
```

Cuando el usuario cierre la ventana secundaria, generará el evento **Closed**. La respuesta a este evento, como se indica a continuación, será acceder a los datos introducidos por el usuario, asignárselos a la nueva entidad *alumno* que deseamos añadir, obtener el contexto de dominio del objeto **DomainDataSource**, añadir esa entidad al conjunto de entidades *alumnos* del contexto de dominio y enviar los cambios a la base de datos invocando a su método **SubmitChanges**:

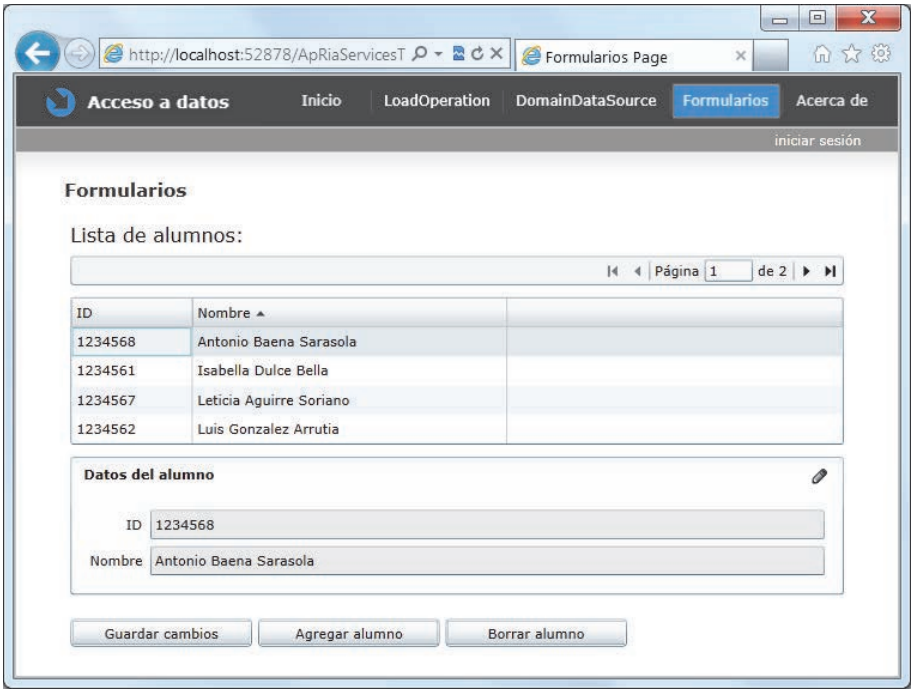
```
Private Sub ventana_Closed(sender As Object, e As EventArgs)
  Dim ventana As VentanaNuevoAlumno =
    TryCast(sender, VentanaNuevoAlumno)
  If ventana.Nuevo_alumno IsNot Nothing Then
    Dim alum As New alumno()
    alum.id_alumno = ventana.Nuevo_alumno.Id
    alum.nombre = ventana.Nuevo_alumno.NomAlumno
    Dim contextoDom As NotasAlumsDomSvc =
      TryCast(ddsAlumnos.DomainContext, NotasAlumsDomSvc)
    contextoDom.alumnos.Add(alum)
    ddsAlumnos.SubmitChanges()
  End If
End Sub
```

Ejecute ahora la aplicación, haga clic en la pestaña *Formularios* del menú, haga clic en el botón *Agregar alumno*, introduzca los datos, acéptelos y actualice la página. Observará que el nuevo alumno aparece en el control **DataGrid**.

Borrar registros

En este apartado, vamos a explicar cómo borrar un registro de la tabla *alumnos* de la base de datos. Esta operación la vamos a realizar seleccionando el registro a borrar en el **DataGrid** y ordenando el borrado desde un botón. Según esto, vamos a añadir a la página *Formularios.xaml* un nuevo botón titulado "Borrar alumno" que colocaremos junto al botón *agregarAlumno*:

```
<StackPanel Orientation="Horizontal" Margin="0,15,0,0" >
  <Button x:Name="guardarCambios" .../>
  <Button x:Name="agregarAlumno" .../>
  <Button x:Name="borrarAlumno" Content="Borrar alumno"
          Width="150" Height="23"  Margin="5,0,0,0"
          Click="borrarAlumno_Click"/>
</StackPanel>
```



A continuación, añadimos el controlador para su evento **Click**. Este controlador obtendrá el contexto de dominio del objeto **DomainDataSource**, invocará al método *Remove* de su conjunto de entidades *alumnos* pasando como argumento la entidad a borrar, y enviará los cambios a la base de datos invocando al método **SubmitChanges** del contexto de dominio:

```
Private Sub borrarAlumno_Click(sender As System.Object, e As Sys-
tem.Windows.RoutedEventArgs)
  Dim contextoDom As NotasAlumsDomSvc =
    TryCast(ddsAlumnos.DomainContext, NotasAlumsDomSvc)
  contextoDom.alumnos.Remove(TryCast(dgAlumnos.SelectedItem, alumno))
  ddsAlumnos.SubmitChanges()
End Sub
```

Como habrá observado, tanto la operación de añadir como la de borrar se realizan sobre los conjuntos de entidades del contexto de dominio utilizado por el control **DomainDataSource**, para después, utilizando la funcionalidad de este control, enviar los cambios a la base de datos.

RESUMEN

WCF RIA Services pone la lógica de negocio del servidor a disposición del cliente de RIA Services sin que sea necesario duplicar manualmente tal lógica de ne-

gocio. Por lo tanto, es posible tomar ventaja de WCF RIA en escenarios relacionados con el acceso a datos almacenados en una base de datos. Para ver todo esto de una forma práctica, creamos una aplicación de RIA Services que tuviera que acceder a los datos de una base de datos, para lo cual, previamente modelamos los datos de la base utilizando *Entity Data Model*. Después, para consultar los datos desde el cliente agregamos un servicio de dominio (**DomainService**) que expusiera las entidades y mostrara esos datos en el proyecto de cliente.

Una vez añadido el servicio de dominio y tras compilar la solución, se generaron automáticamente en el proyecto de cliente el contexto de dominio (**DomainContext**), y las entidades, funcionalidad que utilizamos para acceder a los datos de la base de datos por medio de dos técnicas diferentes: una basada en la clase genérica **LoadOperation** y la otra en el control **DomainDataSource**. La ventaja de utilizar la segunda técnica frente a la primera es que este control permite especificar con muy poco esfuerzo ordenación, filtrado y paginación en la presentación de datos. Para implementar la paginación el control **DomainDataSource** utiliza un control **DataPager**.

Finalmente, vimos las operaciones de actualizar, añadir o borrar registros de la base de datos. Para facilitar la operación de actualización utilizamos un control **DataForm**, para la operación de añadir un nuevo registro solicitamos los datos al usuario por medio de una ventana secundaria, y la operación de borrado la realizamos sobre el registro actualmente seleccionado en el **DataGrid**.

APÉNDICE A

© F.J.Ceballos/RA-MA

ENTORNO DE DESARROLLO INTEGRADO

Se puede desarrollar una aplicación que muestre una interfaz gráfica utilizando como herramientas *Microsoft Framework SDK* (proporciona, entre otras cosas, la biblioteca de clases .NET y el compilador de Visual Basic) y un simple editor de texto, o bien utilizando un entorno de desarrollo integrado (EDI). En el primer caso hay que escribir el código fuente línea a línea, para después, desde la línea de órdenes, compilarlo, ejecutarlo y depurarlo. Lógicamente, escribir todo el código necesario para crear la interfaz gráfica de la aplicación es una tarea repetitiva que, de poder mecanizarse, ahorraría mucho tiempo en la implementación de una aplicación y permitiría centrarse más y mejor en resolver los problemas relativos a su lógica y no a su aspecto. Justamente esto es lo nuevo que aporta *Visual Studio* o, en su defecto, las versiones de *Visual Basic Express* y *Visual Web Developer Express*.

El propósito de este apéndice es mostrar cómo crear una aplicación sencilla de *Windows Presentation Foundation* (WPF) y familiarizarse con el entorno de desarrollo integrado (EDI) de *Visual Studio*/*Visual Basic Express*. Al igual que las aplicaciones de formularios Windows Forms, las aplicaciones WPF se pueden diseñar arrastrando controles desde la caja de herramientas hasta el panel de diseño.

MICROSOFT VISUAL STUDIO

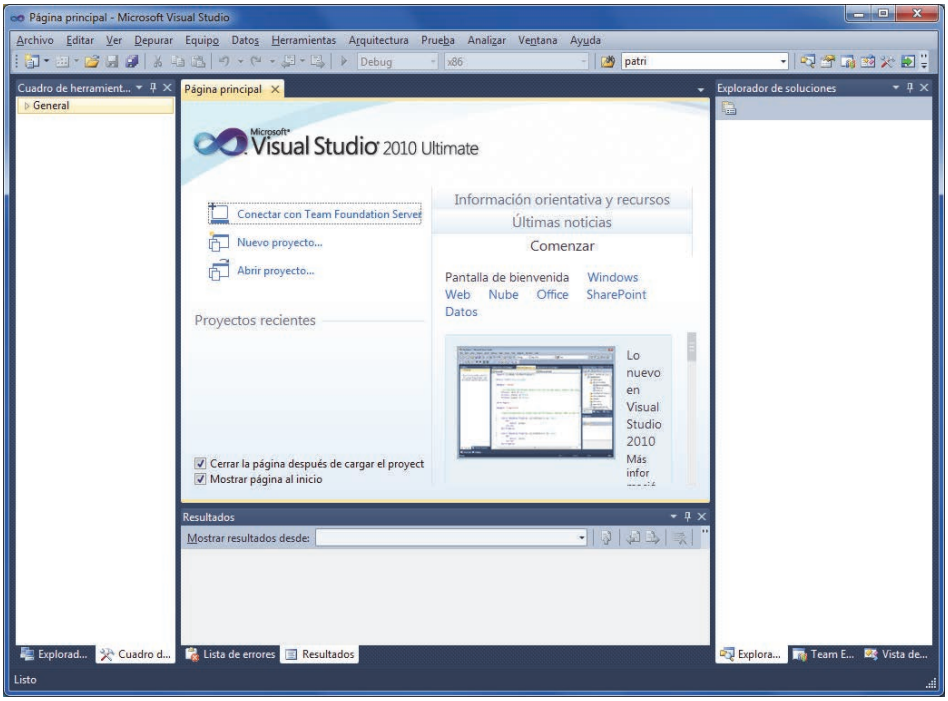
Visual Studio permite diseñar la interfaz gráfica de una aplicación WPF de manera visual, sin más que arrastrar con el ratón los controles que necesitemos sobre la ventana destino de los mismos. Unas líneas de guía o una rejilla mostrada sobre el formulario (o ventana) nos ayudarán a colocar estos controles y a darles el tamaño adecuado, y una página de propiedades nos facilitará la modificación de los valo-

res de las propiedades de cada uno de los controles. Todo lo expuesto lo realizaremos sin tener que escribir ni una sola línea de código. Después, un editor de código inteligente nos ayudará a escribir el código necesario y detectará los errores sintácticos que introduzcamos, y un depurador nos ayudará a poner a punto nuestra aplicación cuando lo necesitemos.

Como ejemplo, vamos a realizar una aplicación Windows denominada *Saludo*, que presente una interfaz al usuario como la de la figura siguiente:



Para empezar, arranque Visual Studio. Se visualizará una ventana similar a la siguiente:

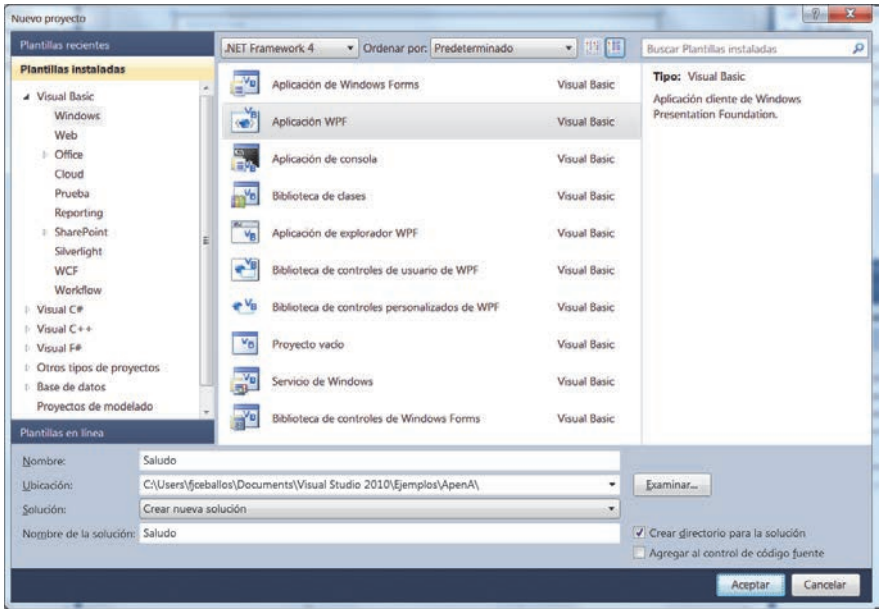


¿Cuáles son los siguientes pasos para desarrollar una aplicación WPF? En general, para construir una aplicación de este tipo con Visual Studio, siga los pasos indicados a continuación:

1. Cree un nuevo proyecto (una nueva aplicación), entendiendo por proyecto un conjunto de ficheros, normalmente distribuidos en carpetas y recursos que pueden ser compilados como una sola unidad. Visual Studio mostrará una página de diseño con un formulario vacío por omisión (una ventana).
2. Dibuje los controles sobre el formulario. Los controles serán tomados de una caja de herramientas.
3. Defina las propiedades del formulario y de los controles.
4. Escriba el código para controlar los eventos que consideremos de cada uno de los objetos.
5. Guarde, compile y ejecute la aplicación.
6. Opcionalmente, utilice un depurador para poner a punto la aplicación.

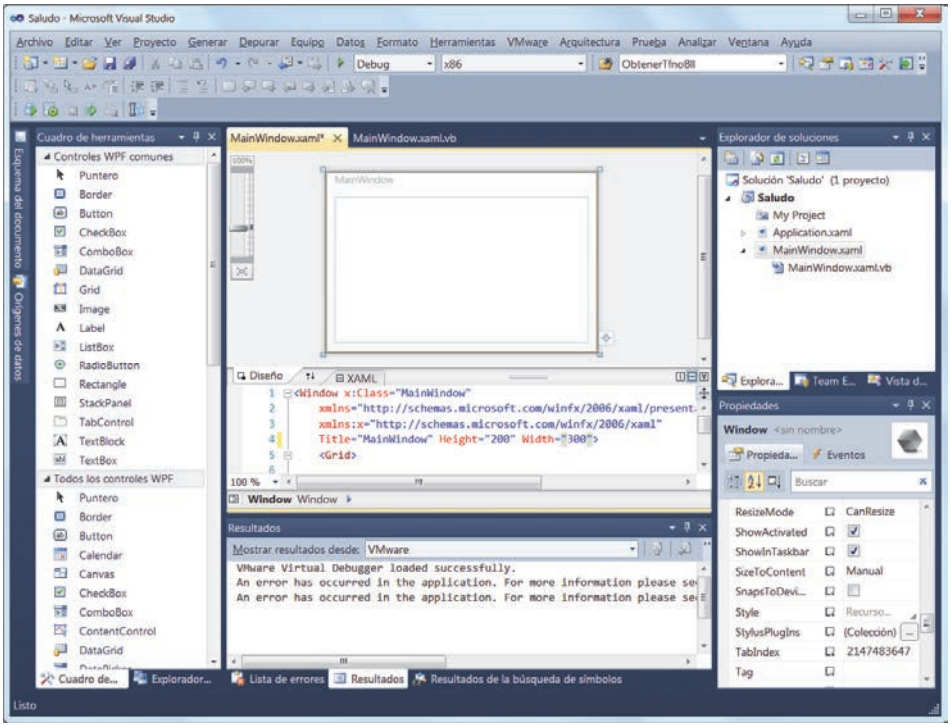
Crear un nuevo proyecto

Para crear un nuevo proyecto, diríjase a la barra de menús y ejecute *Archivo* > *Nuevo Proyecto*. En el diálogo que se visualiza, seleccione el tipo de proyecto *Visual Basic* > *Windows* > *Aplicación WPF* y asígnele el nombre *Saludo*. Observe, en la parte superior de la ventana, que puede elegir la versión de *.NET Framework*. Después, para continuar, haga clic en el botón *Aceptar*:



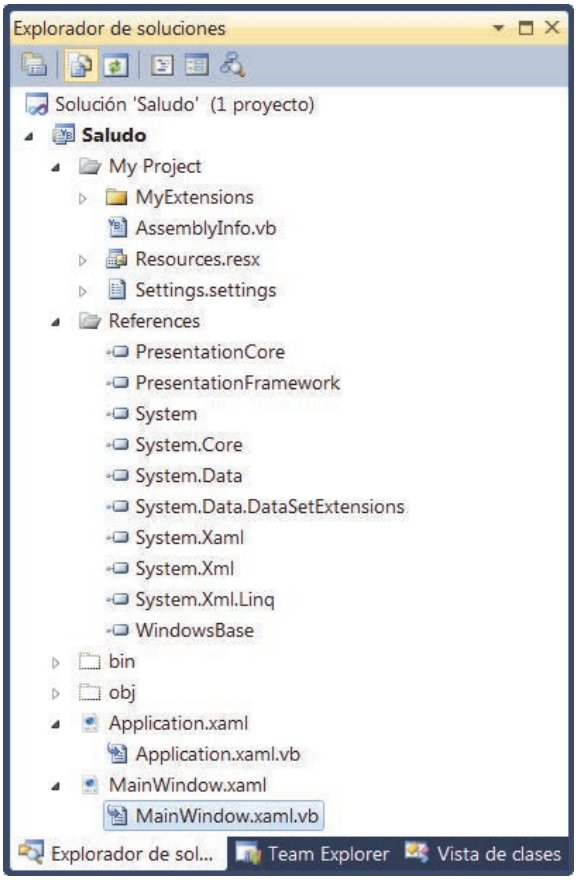
Obsérvese que se ha elegido la carpeta donde se almacenará el proyecto. Esta tarea puede posponerse sin que afecte al desarrollo de la aplicación. Ahora bien, si desea que esta tarea se realice automáticamente en el momento de crear el proyecto, caso del autor, ejecute *Herramientas > Opciones*, seleccione la opción *Proyectos y Soluciones* y marque la casilla *Guardar los proyectos nuevos al crearlos*.

Después de crear una nueva aplicación Windows, el entorno de desarrollo Visual Studio mostrará un formulario, *MainWindow*, en el diseñador. También pondrá a nuestra disposición una caja de herramientas con una gran cantidad de controles WPF listos para ser incluidos en la ventana.



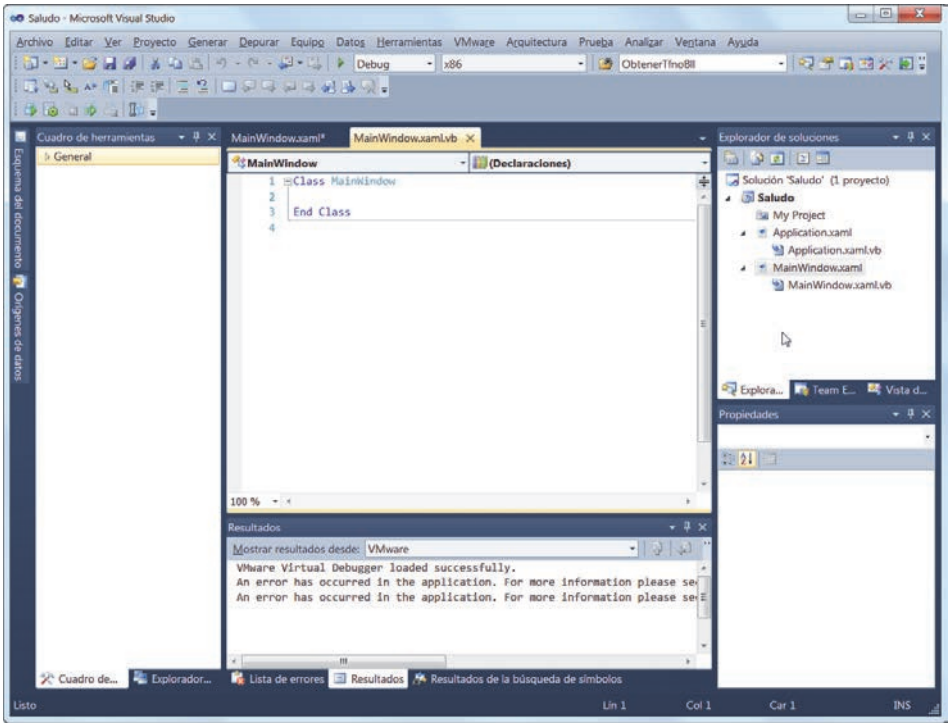
Otra característica interesante de este entorno de desarrollo es la ayuda que facilita. Basta con colocar el punto de inserción sobre una palabra clave y pulsar la tecla *F1* para que le muestre la ayuda correspondiente a la palabra seleccionada.

En la esquina superior derecha también se localiza otra ventana con varias páginas: explorador de soluciones, vista de clases, etc.; en la figura siguiente vemos el *Explorador de soluciones*:



El *Explorador de soluciones* muestra el nombre de la solución (una solución engloba uno o más proyectos), el nombre del proyecto (un proyecto administra los ficheros que componen la aplicación) y el de todos los formularios y demás módulos que intervienen en la aplicación; en nuestro caso, observamos un formulario, denominado *MainWindow*, descrito por los ficheros de código *MainWindow.xaml* y *MainWindow.xaml.vb*; el primero es el utilizado por el diseñador de formularios y el segundo, el utilizado por el programador para escribir el código. También se observa un nodo *References* que agrupa las referencias a las bibliotecas de clases de objetos que utilizará la aplicación en curso; podemos añadir nuevas referencias a otras bibliotecas haciendo clic con el botón secundario del ratón sobre ese nodo o bien eliminarlas.

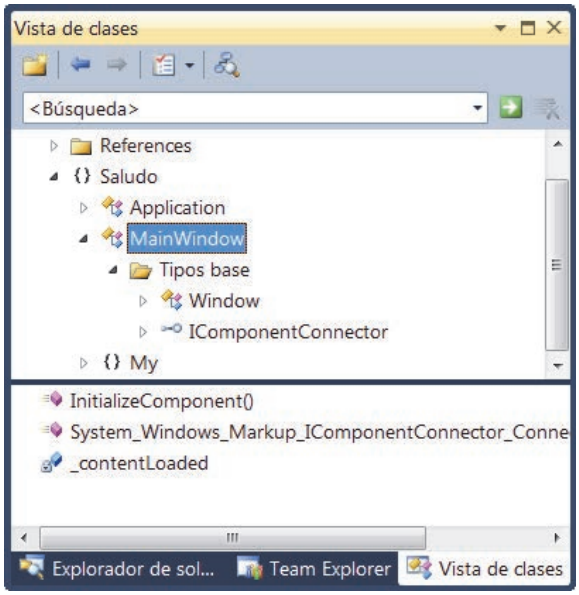
Así mismo, en su parte superior, muestra una barra de botones que permiten ver el *código*, el *diseñador* de formularios, la ventana de *propiedades*, etc. Por ejemplo, si estamos viendo el diseñador de formularios y hacemos clic en el botón *Ver código*, la página de diseño será sustituida por el editor de código, como se puede observar en la figura siguiente:



Una característica digna de resaltar del editor de Visual Studio es la incorporación de bloques de código contraíbles. En la figura superior podemos ver uno de estos bloques; si hacemos clic en un nodo –, contraeremos el bloque y ese nodo se convertirá en otro + que permitirá expandir de nuevo el bloque.

Otra característica del editor es la finalización y el formato de código automáticos. Por ejemplo, al escribir un método, el editor mostrará automáticamente la ayuda en línea de la palabra clave (**Public**, **Sub**, **Integer**, etc.) que intenta escribir; si escribimos una sentencia **If**, exactamente igual. Puede personalizar las características del editor ejecutando *Herramientas* > *Opciones* > *Editor de texto*.

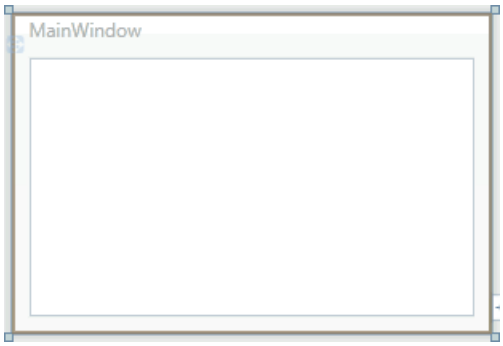
Si cuando se está visualizando el explorador de soluciones desea mostrar la vista de clases de su aplicación, sólo tiene que hacer clic en la pestaña *Vista de clases*. Esta ventana en su parte superior muestra las clases que componen la aplicación y en su parte inferior los métodos pertenecientes a la clase seleccionada.



Expandiendo el nodo del proyecto, vemos, en primer lugar, el espacio de nombres al que pertenecen las clases que definen la aplicación: *Saludo* (un espacio de nombres define un ámbito). Si ahora expandimos este otro nodo, veremos que incluye las clases mencionadas, la que define el objeto aplicación, *Application*, y la que define la ventana principal, *MainWindow*, y si expandimos a su vez este nodo, podremos observar su clase base, **Window**. Si seleccionó el nodo *MainWindow*, en el panel inferior de la figura podemos observar los métodos de la clase como, por ejemplo, el método *InitializeComponent*, entre otros.

El formulario

El formulario, objeto de la clase **Window** (una ventana), es el plano de fondo para los controles. Después de crear un nuevo proyecto, la página de diseño muestra uno como el de la figura siguiente. Lo que ve en la figura es el aspecto gráfico de un objeto de la clase *MainWindow*. Para modificar su tamaño ponga el cursor del ratón sobre alguno de los lados del cuadrado que lo rodea y arrastre en el sentido deseado.



Si ahora ejecutamos esta aplicación, para lo cual podemos pulsar las teclas *Ctrl+F5*, o bien elegir la orden correspondiente del menú *Depurar*, aparecerá sobre la pantalla el formulario (un objeto ventana), con el tamaño asignado, y podremos actuar sobre cualquiera de sus controles, o bien sobre las órdenes del menú de control, para minimizarlo, maximizarlo, moverlo, ajustar su tamaño, etc. Ésta es la parte que el diseñador de Visual Studio realiza por nosotros y para nosotros; pruébelo. Finalmente, para cerrar la ejecución de la aplicación disponemos de varias posibilidades:

1. Hacer clic en el botón que cierra la ventana.
2. Hacer un doble clic en el icono situado a la izquierda en la barra de título de la ventana.
3. Activar el menú de control de la ventana *MainWindow* y ejecutar *Cerrar*.
4. Pulsar las teclas *Alt+F4*.

Dibujar los controles

En Visual Studio disponemos fundamentalmente de dos tipos de objetos: *ventanas* y *controles*. Las ventanas son los objetos sobre los que se dibujan los controles como cajas de texto, botones o etiquetas, dando lugar a la interfaz gráfica que el usuario tiene que utilizar para comunicarse con la aplicación y que genéricamente denominamos formulario.

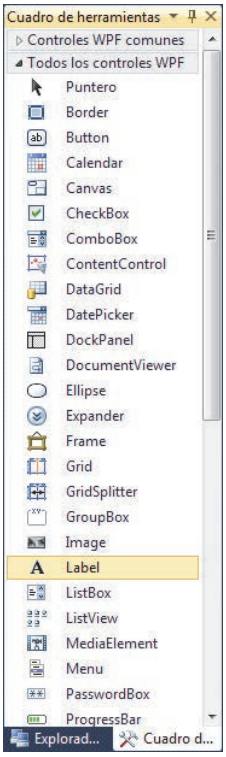
Para añadir un control a un formulario, utilizaremos la caja de herramientas que se muestra en la figura siguiente. Cada herramienta de la caja crea un único control. El significado de algunos de los controles más comunes se expone a continuación.

Puntero. El puntero no es un control. Se utiliza para seleccionar, mover y ajustar el tamaño de los objetos.

Label. Una *etiqueta* permite mostrar un texto de una o más líneas que no puede ser modificado por el usuario. Son útiles para dar instrucciones al usuario.

Button. Un *botón de pulsación* normalmente tendrá asociada una orden con él. Esta orden se ejecutará cuando el usuario haga clic sobre el botón.

TextBox. Una *caja de texto* es un área dentro del formulario en la que el usuario puede escribir o visualizar texto.



Menu. Permite añadir un menú de opciones para que el usuario seleccione una.

CheckBox. Una *casilla de verificación* se utiliza para seleccionar una opción. Utilizando estos controles se pueden elegir varias opciones de un grupo.

RadioButton. El control *botón de opción* se utiliza para seleccionar una opción entre varias. Utilizando estos controles se puede elegir una sola opción de un grupo de ellas.

GroupBox. Un *marco* se utiliza para realzar el aspecto del formulario. También los utilizamos para formar grupos de botones de opción, o bien para agrupar controles relacionados entre sí.

Grid. Control que actúa como contenedor de otros controles.

DataGrid. Proporciona una tabla para visualizar los datos de un origen de datos de una forma personalizada.

ListBox. El control *lista fija* (lista desplegada) contiene una lista de elementos de la que el usuario puede seleccionar uno o varios.

ComboBox. El control *lista desplegable* combina una caja de texto y una lista desplegable. Permite al usuario escribir lo que desea seleccionar o elegir un elemento de la lista.

ListView. El control *vista de lista* muestra una colección de elementos que se pueden visualizar mediante una de varias vistas distintas.

TreeView. Representa un control que muestra datos jerárquicos en una estructura de árbol con nodos que se pueden expandir y contraer.

TabControl. Es un control que agrupa un conjunto relacionado de fichas.

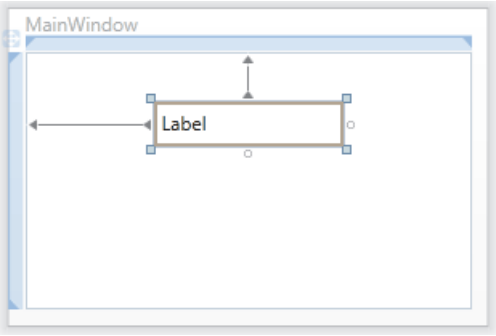
DatePicker. Control que permite seleccionar la fecha y hora.

ScrollBar. Representa una *barra de desplazamiento horizontal* o *vertical* que permite seleccionar un valor dentro de un rango de valores. Estos controles son utilizados independientemente de otros objetos, y no son lo mismo que las barras de desplazamiento de una ventana.

Otros controles de interés son la barra de progreso (*ProgressBar*), la caja de texto enriquecido (*RichTexBox*), la barra de estado (*StatusBar*), etc.

Observe que el formulario *MainWindow* ya contiene un **Grid** que actuará como contenedor de los controles que arrastremos sobre el mismo.

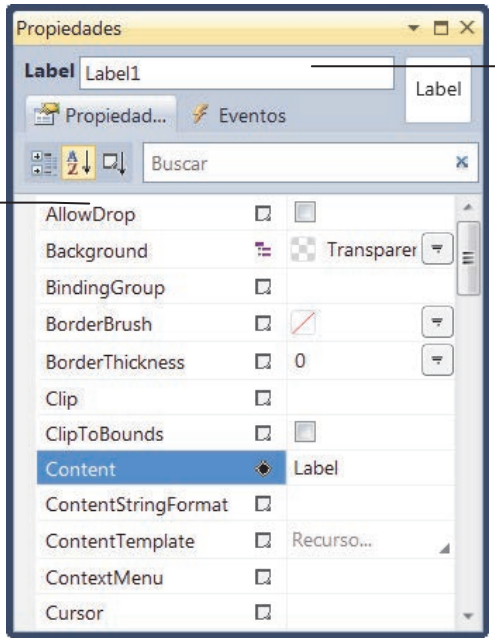
Siguiendo con nuestra aplicación, seleccionamos de la caja de herramientas que acabamos de describir los controles que vamos a utilizar. En primer lugar vamos a añadir al formulario una etiqueta. Para ello, hacemos clic sobre la herramienta etiqueta (*Label*) y, sin soltar el botón del ratón, la arrastramos sobre el formulario. Cuando soltemos el botón del ratón aparecerá una etiqueta de un tamaño predefinido, según se muestra en la figura siguiente:



Observe en la página de propiedades del entorno de desarrollo, mostrada en la figura siguiente, las propiedades **Name**, nombre, y **Content**, contenido. La primera tiene asignado el valor *label1* que es el nombre por defecto dado al control *Label*, y la segunda tiene asignado por defecto el valor "Label", contenido que mostrará la etiqueta.

Si la ventana propiedades no está visible, ejecute la orden *Ventana de propiedades* en el menú *Ver* o pulse *F4*.

El nombre de un control se utiliza para referirnos a dicho control en el código de la aplicación.



Propiedad **Name**.

Lista de las propiedades del objeto seleccionado. Una propiedad se modifica *in situ*.

Estando la etiqueta seleccionada se observa sobre la misma un rectángulo con unos cuadrados distribuidos a lo largo de su perímetro, que reciben el nombre de *modificadores de tamaño*, indicando que se puede modificar el tamaño del control que estamos dibujando. Para ello, primero selecciónelo haciendo clic sobre él, después apunte con el ratón a alguno de los lados del rectángulo que lo envuelve, observe que aparece una doble flecha, y, entonces, con el botón izquierdo del ratón pulsado, arrastre en el sentido que desee ajustar el tamaño.

También puede mover el control a un lugar deseado dentro del formulario. Para mover un control, primero selecciónelo haciendo clic sobre él y después apunte con el ratón a alguna zona perteneciente al mismo y, con el botón izquierdo del ratón pulsado, arrastre hasta situarlo en el lugar deseado.

Después de haber arrastrado un control sobre el formulario, Visual Studio crea automáticamente el código XAML que hará que el control se muestre cuando se ejecute el programa. De manera predeterminada, el editor XAML aparece bajo el diseñador. En la parte superior de este editor hay botones que le permitirán ver el marcado XAML en modo pantalla completa, en modo pantalla compartida horizontalmente o verticalmente, en la parte superior, etc., y en la parte inferior izquierda hay otro botón que muestra el esquema del documento que le permitirá navegar por el árbol de elementos de la ventana.



Borrar un control

Para borrar un control, primero se selecciona haciendo clic sobre él y, a continuación, se pulsa la tecla *Supr* (*Del*). Para borrar dos o más controles, primero se seleccionan haciendo clic sobre cada uno de ellos, al mismo tiempo que se mantiene pulsada la tecla *Ctrl*, y después se pulsa *Supr*.

Se pueden seleccionar también dos o más controles contiguos, pulsando el botón izquierdo y arrastrando el ratón hasta rodearlos.

Propiedades de los objetos

Cada clase de objeto tiene predefinido un conjunto de propiedades, como nombre, tamaño, color, etc. Las propiedades de un objeto representan todos los atributos que por definición están asociados con ese objeto.

Cuando se selecciona más de un objeto, la página de propiedades visualiza las propiedades comunes a esos objetos.

Cada propiedad de un objeto tiene un valor por defecto que puede ser modificado *in situ* si se desea. Por ejemplo, la propiedad **Title** del formulario del ejemplo que nos ocupa tiene el valor *MainWindow*.

Para cambiar el valor de una propiedad de un objeto, siga los pasos indicados a continuación:

1. Seleccione el objeto. Para ello, haga clic sobre el objeto o pulse sucesivamente la tecla *Tab* hasta que esté seleccionado (el control seleccionado aparecerá rodeado por un rectángulo modificador de tamaño).

2. Seleccione en la lista de propiedades la propiedad que desea cambiar.

3. Modifique el valor que actualmente tiene la propiedad seleccionada. El valor actual de la propiedad en cuestión aparece escrito a continuación del nombre de la misma. Para cambiar este valor, sobrescriba el valor actual o, si es posible, seleccione uno de la lista que se despliega haciendo clic sobre la misma. Para algunas propiedades, esta lista es sustituida por una caja de diálogo.

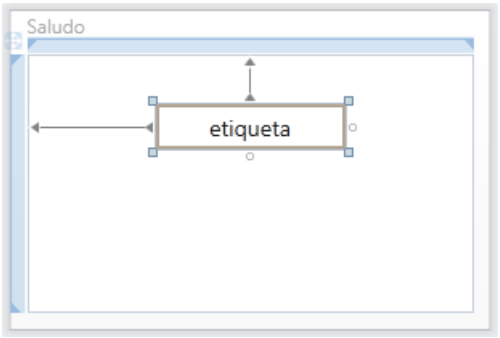
Se puede también modificar una propiedad durante la ejecución de la aplicación. Esto implica añadir el código necesario en el método que deba realizar la modificación.

Para verificar el valor de una misma propiedad en varios objetos, se selecciona ésta en la página de propiedades para uno de ellos, y a continuación se pasa de un objeto al siguiente haciendo clic con el ratón sobre cada uno de ellos, o simplemente pulsando la tecla *Tab*.

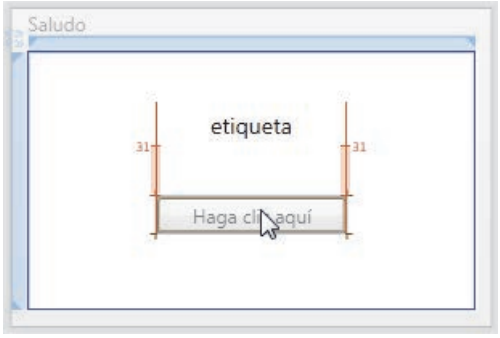
Siguiendo con nuestro ejemplo, vamos a cambiar el título del formulario. Para ello, seleccione el formulario y a continuación la propiedad **Title** en la página de propiedades. Después, sobrescriba el texto “MainWindow” con el texto “Saludo”.

Veamos ahora las propiedades de la etiqueta. Seleccione la etiqueta y observe la lista de propiedades. Algunas de estas propiedades son **Background** (color del fondo de la etiqueta), **Name** (identificador de la etiqueta para referirnos a ella en

el código) y **Content** (contenido de la etiqueta). Siguiendo los pasos descritos anteriormente, cambie el valor actual de la propiedad **Name** al valor *etSaludo*, el contenido "Label" de la propiedad **Content** a "etiqueta" y alinee este texto para que se muestre centrado tanto horizontal como verticalmente; esto requiere asignar a las propiedades **HorizontalContentAlignment** y **VerticalContentAlignment** el valor *Center*. A continuación, vamos a modificar el tamaño de la letra de la etiqueta; para ello, seleccione la propiedad **FontSize** en la página de propiedades, despliegue la lista de la derecha y elija como tamaño, por ejemplo, 14. El resto de las propiedades las dejamos como están.



El paso siguiente será añadir un botón. Para ello, hacemos clic sobre la herramienta *Button* de la caja de herramientas y arrastramos el botón sobre el formulario. Modificamos sus propiedades y asignamos a **Content** el valor *Haga clic aquí*, y a **Name**, el valor *btSaludo*. Después, movemos el botón y ajustamos su tamaño para conseguir el diseño que observamos en la figura siguiente.



También observamos que al colocar el control aparecen unas líneas indicando la alineación de éste con respecto a otros controles. Es una ayuda para alinear los controles que coloquemos dentro del formulario. Puede elegir entre los modos *SnapLines* (líneas de ayuda), es el modo que estamos utilizando, o *SnapToGrid* (rejilla de ayuda; se visualizan los puntos que dibujan la rejilla). Para elegir el modo de ayuda, ejecute *Herramientas* > *Opciones*, seleccione la opción *Diseña-*

dor de formularios Windows y asigne a la propiedad **LayoutMode** el modo deseado. Para que las opciones elegidas tengan efecto, tiene que cerrar el diseñador y volverlo a abrir.

Icono de la aplicación

Todos los formularios visualizan un icono en la esquina superior izquierda que generalmente ilustra la finalidad de la aplicación y que también aparece cuando se minimiza el formulario. Por omisión, Visual Studio utiliza un icono genérico.

Para utilizar su propio icono (de 16 × 16 o de 32 × 32 píxeles), sólo tiene que asignarlo a la propiedad **Icon** del formulario; esto es, seleccione el formulario, vaya a la página de propiedades, elija la propiedad **Icon**, pulse el botón que se muestra a la derecha y asigne el fichero *.ico* que contiene el icono.

Escribir los controladores de eventos

Sabemos que el nombre de un objeto, propiedad **Name**, nos permite referirnos a él dentro del código de la aplicación; por ejemplo, en las líneas de código siguiente, la primera asigna el valor "¡¡¡Hola mundo!!!" a la propiedad **Content** del objeto *etSaludo* y la siguiente obtiene el valor de la etiqueta y lo almacena en la variable *sTexto*:

```
etSaludo.Content = "¡¡¡Hola mundo!!!"
Dim sTexto As String = etSaludo.Content.ToString()
```

En Visual Basic la forma general de referirse a una propiedad de un determinado objeto es:

Objeto.Propiedad

donde *Objeto* es el nombre del formulario o control y *Propiedad* es el nombre de la propiedad del objeto cuyo valor queremos asignar u obtener.

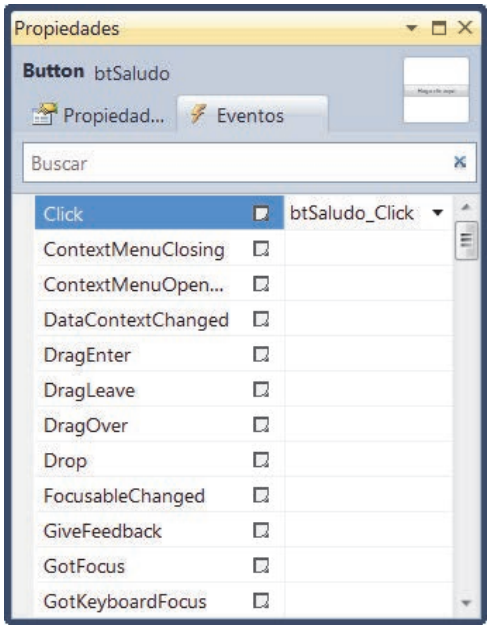
Una vez que hemos creado la interfaz o medio de comunicación entre la aplicación y el usuario, tenemos que escribir los métodos para controlar, de cada uno de los objetos, aquellos eventos que necesitemos manipular.

Hemos dicho que una aplicación en Windows es conducida por *eventos* y *orientada a objetos*. Esto es, cuando sobre un objeto ocurre un suceso (por ejemplo, el usuario hizo clic sobre un botón) se produce un evento (por ejemplo, el evento **Click**); si nosotros deseamos que nuestra aplicación responda a ese evento, tendremos que escribir un método que incluya el código que debe ejecutarse y vincularlo con el objeto que genera el evento. El método pertenecerá a la interfaz

del objeto o del objeto padre. Por ejemplo, el método que responda al evento **Click** de un botón pertenecerá a la interfaz de su ventana padre, esto es, a su contenedor.

¿Dónde podemos ver la lista de los eventos a los que puede responder un objeto de nuestra aplicación? En la ventana de propiedades.

Por ejemplo, seleccione el botón *btSaludo* en la ventana de diseño, vaya a la ventana de propiedades y muestre la lista de eventos para el control seleccionado, haciendo clic en el botón *Eventos*. Haga doble clic en el evento **Click**, o bien escriba manualmente el nombre del controlador y pulse *Entrar*.



El resultado es que se añade a la clase *MainWindow* un manejador para este evento (fichero *MainWindow.xaml.vb*):

```
Private Sub btSaludo_Click(sender As System.Object,
e As System.Windows.RoutedEventArgs) Handles btSaludo.Click

End Sub
```

Alternativamente, podríamos añadir este manejador sin especificar la cláusula **Handles** y especificando en el código XAML (fichero *MainWindow.xaml*) que este método, *btSaludo_Click*, manejará el evento **Click**, según muestra el código escrito a continuación:

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/.../presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Saludo" Height="200" Width="300">
  <Grid>
    <Label Name="etSaludo" Content="etiqueta" FontSize="14"
           Height="28" Margin="80,31,0,0" Width="118"
           HorizontalAlignment="Left" VerticalAlignment="Top"
           HorizontalContentAlignment="Center"
           VerticalContentAlignment="Center" />
    <Button Name="btSaludo" Content="Haga clic aquí"
            Height="23" Width="118" Margin="80,96,0,0"
            HorizontalAlignment="Left" VerticalAlignment="Top"
            Click="btSaludo_Click" />
  </Grid>
</Window>
```

y el método *btSaludo_Click* que responderá a ese evento **Click** cada vez que se genere (fichero *MainWindow.xaml.vb*):

```
Private Sub btSaludo_Click(sender As Object, e As RoutedEventArgs)
  ' Escriba aquí el código que tiene que ejecutarse para responder
  ' al evento Click que se genera al pulsar el botón
End Sub
```

El primer parámetro del método anterior hace referencia al objeto que generó el evento y el segundo contiene datos relacionados con el evento.

Una vez añadido el controlador para el evento **Click** del botón *btSaludo*, ¿cómo lo completamos? Lo que deseábamos era que la etiqueta mostrara el mensaje "¡¡¡Hola mundo!!!" cuando el usuario hiciera clic en el botón. Según esto, complete este controlador así:

```
private void btSaludo_Click(object sender, RoutedEventArgs e)
{
  etSaludo.Content = "¡¡¡Hola mundo!!!"
}
```

Para añadir el controlador anterior, también podríamos habernos dirigido a la página de diseño y haber hecho doble clic sobre el botón de pulsación.

Además del evento **Click**, hay otros eventos asociados con un botón de pulsación, según se puede observar en la figura anterior.

Un detalle de estilo a la hora de escribir el código. Observe que *Visual Studio* antepone a los nombres de las clases y otras estructuras de datos el nombre del espacio de nombres al que pertenecen (por ejemplo **System.Object** en lugar de es-

cribir solamente el nombre de la clase **Object**), aunque no sería necesario, porque las referencias añadidas en el explorador de soluciones, equivalentes a las sentencias **Imports** que se muestran a continuación, ya los especifican.

```
Imports System
Imports System.Xaml
Imports System.Windows.Controls
```

Análogamente a como las carpetas o directorios ayudan a organizar los ficheros en un disco duro, los espacios de nombres ayudan a organizar las clases en grupos para facilitar el acceso a las mismas y proporcionan una forma de crear tipos globales únicos, evitando conflictos en el caso de clases de igual nombre pero de distintos fabricantes, ya que se diferenciarán en su espacio de nombres.

Guardar la aplicación

Una vez finalizada la aplicación, se debe guardar en el disco para que pueda tener continuidad; por ejemplo, por si más tarde se quiere modificar. Esta operación puede ser que se realice automáticamente cuando se compila o se ejecuta la aplicación y si no, puede requerir guardar la aplicación en cualquier instante ejecutando la orden *Guardar todo* del menú *Archivo*.

Si desplegamos el menú *Archivo*, nos encontraremos, además de con la orden *Guardar todo*, con dos órdenes más: *Guardar nombre-fichero* y *Guardar nombre-fichero como...* La orden *Guardar nombre-fichero* guarda en el disco el fichero actualmente seleccionado y la orden *Guardar nombre-fichero como...* realiza la misma operación, y además nos permite cambiar el nombre, lo cual es útil cuando el fichero ya existe.

No es conveniente que utilice los nombres que Visual Basic asigna por defecto, porque pueden ser fácilmente sobrescritos al guardar aplicaciones posteriores.

Verificar la aplicación

Para ver cómo se ejecuta la aplicación y los resultados que produce, hay que seleccionar la orden *Iniciar sin depurar* del menú *Depurar* o pulsar *Ctrl+F5*.

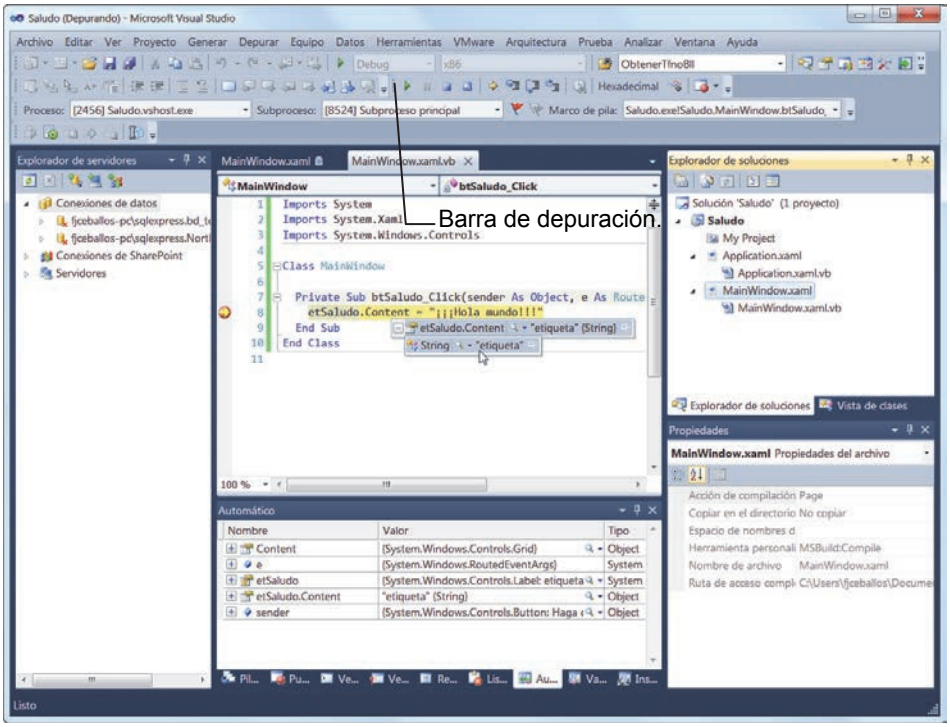
Si durante la ejecución encuentra problemas o la solución no es satisfactoria y no es capaz de solucionarlos por sus propios medios, puede utilizar, fundamentalmente, las órdenes *Paso a paso por instrucciones* (*F11*), *Paso a paso por procedimientos* (*F10*), *Alternar puntos de interrupción* (*F9*), todas ellas del menú *Depurar*, para hacer un seguimiento paso a paso de la aplicación, y las opciones

del menú *Depurar* > *Ventanas* para observar los valores que van tomando las variables y expresiones de la aplicación.

La orden *Paso a paso por instrucciones* permite ejecutar cada sentencia de cada método de la aplicación paso a paso. Esta modalidad se activa y se continúa pulsando *F11*. Cuando una sentencia se corresponde con una llamada a un método y quiere que éste se ejecute en un solo paso, utilice la tecla *F10* (*Paso a paso por procedimientos*). Para detener la depuración pulse las teclas *Mayús+F5*.

La orden *Alternar un punto de interrupción* (*F9*) permite colocar una pausa en cualquier línea. Esto permite ejecutar la aplicación hasta la pausa en un solo paso (*F5*) y ver en la ventana *Automático* los valores que tienen las variables en ese instante. Para poner o quitar una pausa, se coloca el cursor donde se desea que tenga lugar dicha pausa y se pulsa *F9*, o bien se hace clic con el ratón sobre la barra situada a la izquierda del código.

Alternativamente al menú de depuración, puede utilizar la barra de herramientas de depuración. La figura siguiente muestra esta barra dentro de la ventana de Visual Studio en un proceso de depuración. La línea de código sombreada es la siguiente sentencia a ejecutar.



También puede utilizar el ratón para arrastrar el puntero de ejecución (observe la flecha en el margen izquierdo de la ventana anterior) a otro lugar dentro del mismo método con la intención de alterar el flujo normal de ejecución.

Durante el proceso de depuración, puede ver en la ventana *Automático* los valores de las variables y expresiones que desee. Además, en la ventana *Inspección* puede escribir la expresión cuyo resultado desea ver.

También, puede seleccionar en la ventana de código el objeto o propiedad cuyo valor quiere inspeccionar y ejecutar *Inspección rápida...* del menú *Depurar*. Una forma más rápida de hacer esto último es dirigirse al código y situar el puntero del ratón sobre el identificador del objeto o propiedad; le aparecerá una etiqueta con el valor, como se puede observar en la ventana de código anterior.

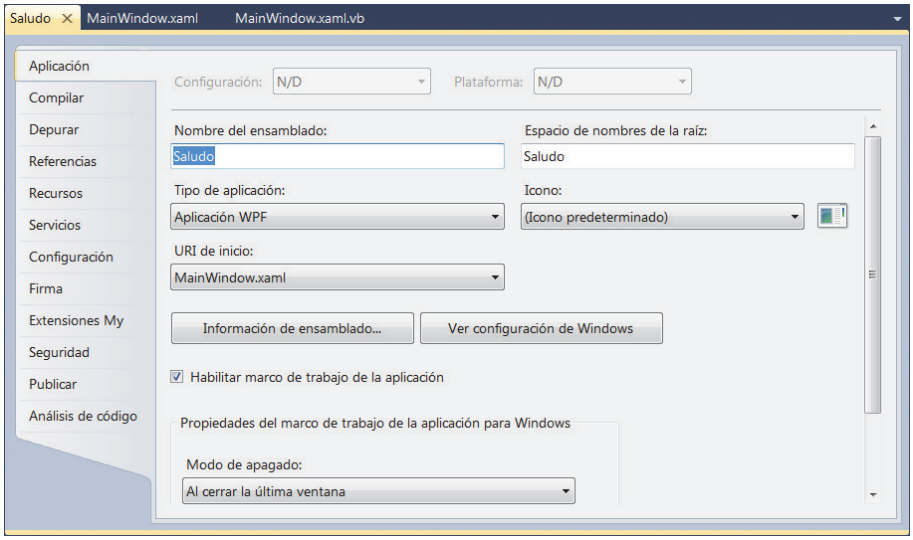
Así mismo, según se observa en la figura siguiente, puede ejecutar en la *Ventana Inmediato* cualquier sentencia de una forma inmediata. Para mostrar u ocultar esta ventana ejecute la orden *Ventanas > Inmediato* del menú *Depurar*. La figura siguiente muestra un ejemplo de cómo se puede inspeccionar el valor de la propiedad **Content** de la etiqueta *etSaludo* (observe el uso del símbolo ?).



Una vez iniciada la ejecución de la aplicación, si se pulsa la tecla *F5*, la ejecución continúa desde la última sentencia ejecutada en un método hasta finalizar ese método o hasta otro punto de parada.

Propiedades del proyecto

Cuando sea necesario establecer determinadas propiedades del proyecto actual o revisar las actuales hay que ejecutar la orden *Proyecto > Propiedades de nombre-proyecto...* Se le mostrará una ventana con varios paneles. Seleccione el deseado y modifique las propiedades que considere.



Crear soluciones de varios proyectos

Una solución agrupa uno o más proyectos. Por omisión, cuando se crea un nuevo proyecto, en la misma carpeta física se crea la solución (fichero con extensión *.sln*) a la que pertenece, con el mismo nombre que el proyecto. Esta solución permite que los ficheros que forman parte del proyecto se almacenen bajo una estructura de directorios que facilite su posterior localización así como las tareas de compartir la solución con otros desarrolladores de un supuesto equipo.

¿Qué tenemos que hacer si necesitamos agrupar varios proyectos bajo una misma solución? Crear una solución vacía y añadir nuevos proyectos a la solución o añadir nuevos proyectos a la solución existente. Asegúrese de que se va a mostrar siempre el nombre de la solución en el explorador de soluciones. Para ello, ejecute *Herramientas > Opciones > Proyectos y Soluciones > Mostrar siempre la solución*.

Para crear una nueva solución vacía:

1. Ejecute la orden *Archivos > Nuevo > Proyecto*.

2. Como tipo de proyecto, seleccione *Otros Tipos de Proyectos*.

3. Y como plantilla, seleccione *Solución en Blanco*.

4. Finalmente, introduzca el nombre que desea dar a la solución. Se creará un fichero *.sln* con el nombre dado, almacenado en una carpeta con el mismo

nombre. Puede elegir, si lo desea, la posición que ocupará esa carpeta en el sistema de ficheros de su plataforma.

Para añadir un nuevo proyecto a una solución existente:

1. Diríjase al explorador de soluciones y haga clic sobre el nombre de la solución utilizando el botón secundario del ratón. Del menú contextual que se visualiza, ejecute la orden *Añadir > Nuevo Proyecto...*

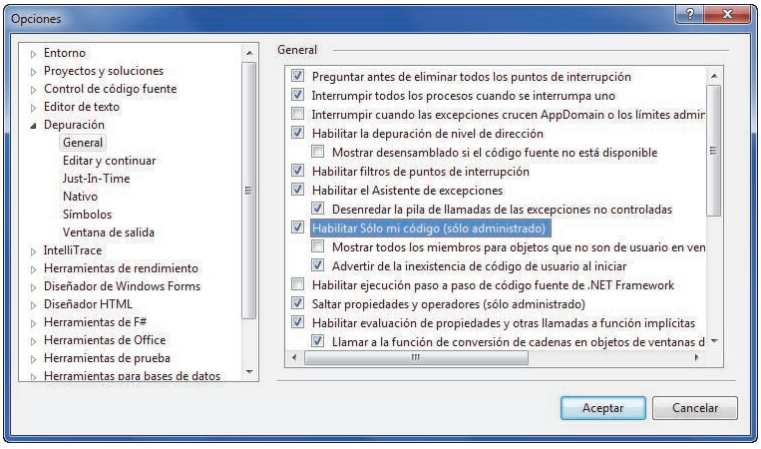
2. Seleccione el tipo de proyecto y la plantilla que va a utilizar para crearlo.

3. Para añadir nuevos proyectos repita los pasos anteriores.

4. Para activar el proyecto sobre el que va a trabajar, haga clic sobre el nombre del proyecto utilizando el botón secundario del ratón y del menú contextual que se visualiza, ejecute la orden *Establecer como proyecto de inicio*.

Opciones del EDI

Las opciones del entorno que se pueden establecer en el entorno de desarrollo integrado (EDI) son mostradas por la ventana *Opciones* que observa en la figura siguiente. Para mostrar esta ventana tiene que ejecutar la orden *Herramientas > Opciones...* Se puede observar que desde esta ventana es posible establecer opciones para el entorno de desarrollo, para los proyectos y soluciones, para el diseñador, para el depurador, etc. Por ejemplo, para que el depurador sólo navegue a través del código escrito por el usuario, no sobre el código añadido por los asistentes, tiene que estar activada la opción "habilitar sólo mi código"; *Herramientas > Opciones > Depuración > General > Habilitar sólo mi código*.



Personalizar el EDI

Para personalizar el entorno de desarrollo tiene que ejecutar la orden *Herramientas > Personalizar...* Desde esta ventana podrá añadir o quitar elementos de un menú, añadir o quitar una barra de herramientas, añadir o quitar un botón de una barra de herramientas, etc.

SQL SERVER EXPRESS

SQL Server Express es el motor de base de datos gratuito, potente, pero sencillo, que se integra perfectamente con el resto de productos Express. Se trata de una versión aligerada de la nueva generación de SQL Server.

Este producto tiene el mismo motor de base de datos que toda la familia SQL Server y utiliza el mismo lenguaje SQL, pero tiene ciertas limitaciones como por ejemplo que sólo puede usar una UCP y un máximo de 1GB de RAM, y las bases de datos no pueden sobrepasar los 4GB. Otra característica interesante es la movilidad de las bases de datos de un servidor a otro con **XCOPY**. Con esta utilidad podemos mover un fichero MDF de una máquina a otra a cualquier ubicación dentro de su sistema de ficheros, quedando la base de datos movida lista para trabajar. Para utilizar esta base de datos deberemos hacer uso de la opción **AttachDBFilename** en la cadena de conexión, según se muestra a continuación:

```
connectionString="Data Source=.\sqlexpress;Initial Catalog=;
Integrated Security=True;AttachDBFileName=C:\bd\bd_telefonos.mdf"
```

La entrada *Data Source* especifica el servidor de base de datos que vamos a utilizar y **AttachDBFilename**, la localización del fichero de base de datos. Obsérvese que la entrada *Initial Catalog* está vacía. O también, si la base de datos la copiamos en el directorio *App_Data* de una aplicación web, la cadena de conexión sería similar a esta otra:

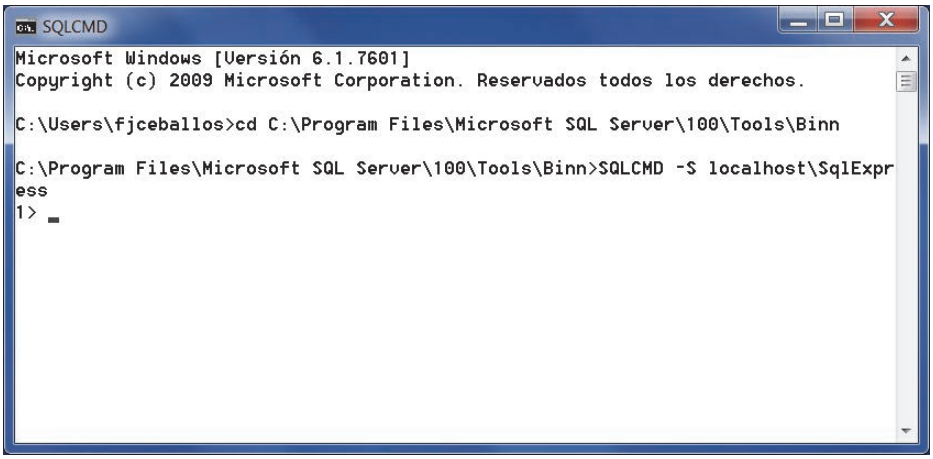
```
connectionString = "Data Source=.\sqlexpress;" +
"Integrated Security=True;" +
"AttachDBFilename=|DataDirectory|\bd_telefonos.mdf;" +
"User Instance=True";
```

El nombre del fichero comienza con |*DataDirectory*|. Esto automáticamente apunta a la carpeta *App_Data* del proyecto web. La entrada *User Instance* especifica que la base de datos es una instancia del usuario porque ésta no está registrada en el catálogo maestro del SQL Server. Esto es, cada base de datos que generamos mediante un *script*, por ejemplo, su nombre es registrado en el catálogo maestro, nombre que después será utilizado en la cadena de conexión por *Initial Catalog*, pero si copiamos una base de datos, ésta no es registrada. En este

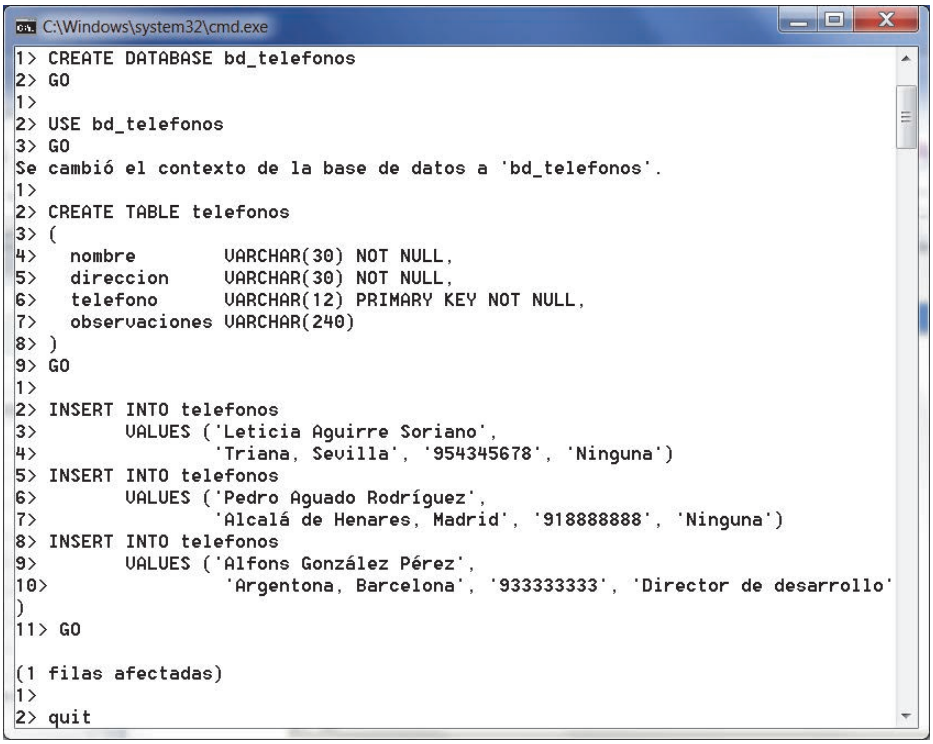
caso Sql Server Express permite acceder a ella tratándola como una instancia del usuario, característica que no está disponible en Sql Server.

Para crear una base de datos utilizando SQL Server Express tiene que hacerlo desde la línea de órdenes (véase también el capítulo titulado *Acceso a una base de datos*). Para iniciar la consola que le permita trabajar contra el motor de base de datos SQL Server, localice en su instalación el fichero SQLCMD.EXE, cambie a ese directorio y ejecute la orden:

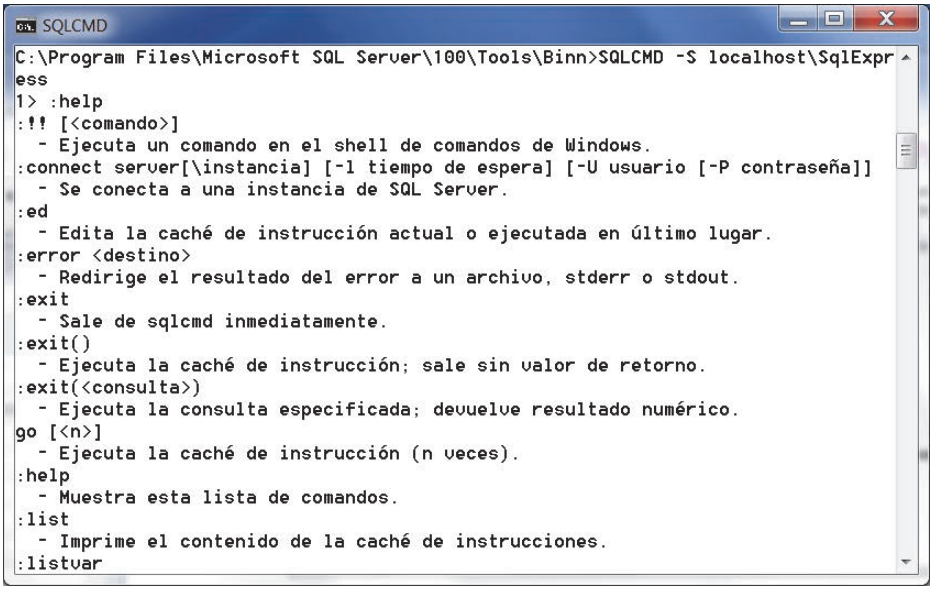
```
SQLCMD -S nombre-del-ordenador\SqlExpress
```



Una vez iniciada la consola, puede escribir órdenes SQL a continuación del símbolo “>”. Para ejecutar un bloque de sentencias escriba GO. Para salir, escriba QUIT. Por ejemplo, el guión que muestra la figura siguiente crea la base de datos *bd_telefonos* con una tabla *telefonos* y añade tres filas a la tabla:



Para ver la relación de órdenes que puede utilizar a través de la aplicación *SQLCMD* ejecute la orden **help** como se muestra en la figura siguiente. Obsérvese que cada orden va precedida por dos puntos (:).



SQL SERVER MANAGEMENT STUDIO EXPRESS

Si instaló SQL Server, habrá comprobado que la única herramienta que proporciona al usuario es *SQL Computer Manager* que sirve para gestionar los servicios básicos de SQL Server y para configurar los protocolos de red. Por tal motivo, Microsoft también ha desarrollado una nueva aplicación para gestionar bases de datos que puede obtener de forma gratuita de Internet en la dirección especificada a continuación:

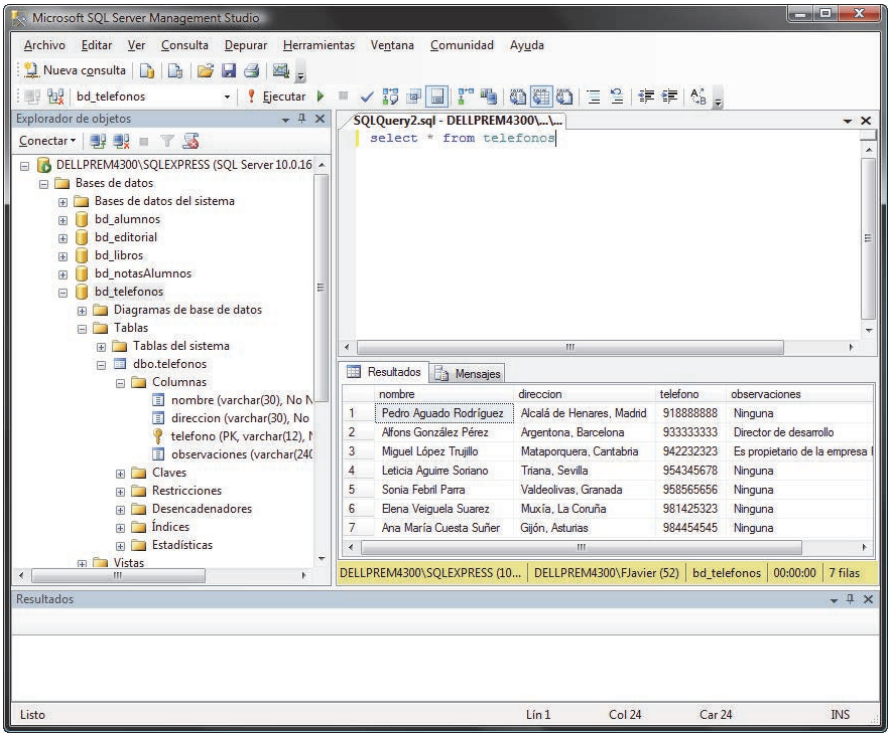
http://www.microsoft.com/downloads/es-es/

Esta aplicación presenta una interfaz gráfica, muy sencilla de utilizar, para realizar tareas típicas como crear bases de datos, gestionar las tablas de la base, los procedimientos almacenados, crear usuarios, etc.

Cuando inicie *SQL Server Management Studio Express*, le serán solicitados el nombre del servidor de bases de datos, el tipo de autenticación, y el usuario y la contraseña sólo si eligió autenticación SQL Server:



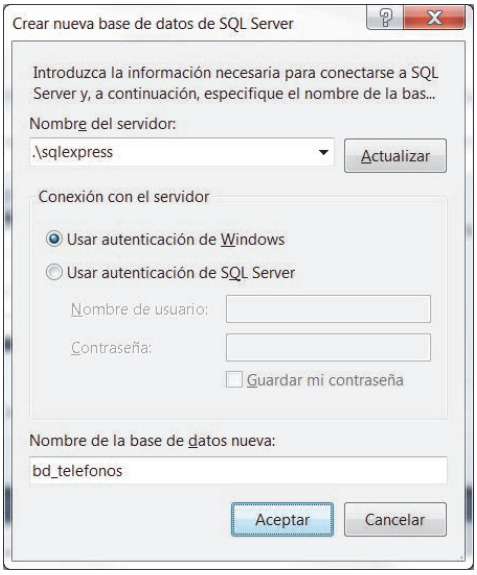
Una vez realizada la conexión con el gestor de bases de datos, le será mostrada la ventana de la figura siguiente. Seleccione en la lista del panel de la izquierda la base de datos con la que desea trabajar, haga clic en el botón *Nueva consulta* de la barra de herramientas y, después, escriba en el mismo las sentencias SQL que desee ejecutar. Para ejecutar una sentencia SQL haga clic en el botón *Ejecutar* de la barra de herramientas.



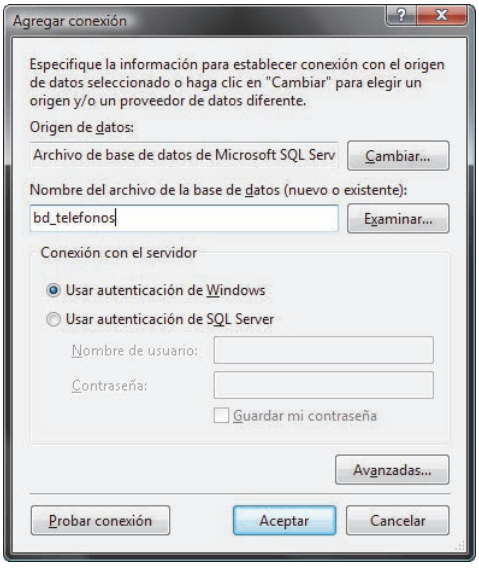
EXPLORADOR DE BASES DE DATOS

Otra forma de crear bases de datos es a través del explorador de bases de datos del EDI de Visual Studio. Si instaló Visual Studio o Visual Basic Express y SQL Server Express, entonces es posible añadir al proyecto elementos nuevos de tipo base de datos utilizando el explorador de bases de datos. Para ello, abra el *Explorador de bases de datos* (*Ver* > *Explorador de servidores* en Visual Studio o *Ver* > *Otras ventanas* > *Explorador de bases de datos* en Visual Basic Express). Haga clic con el botón secundario del ratón sobre el nodo *Conexiones de datos* y seleccione *Crear nueva base de datos SQL Server* (o *Nueva conexión* en Visual Basic Express).

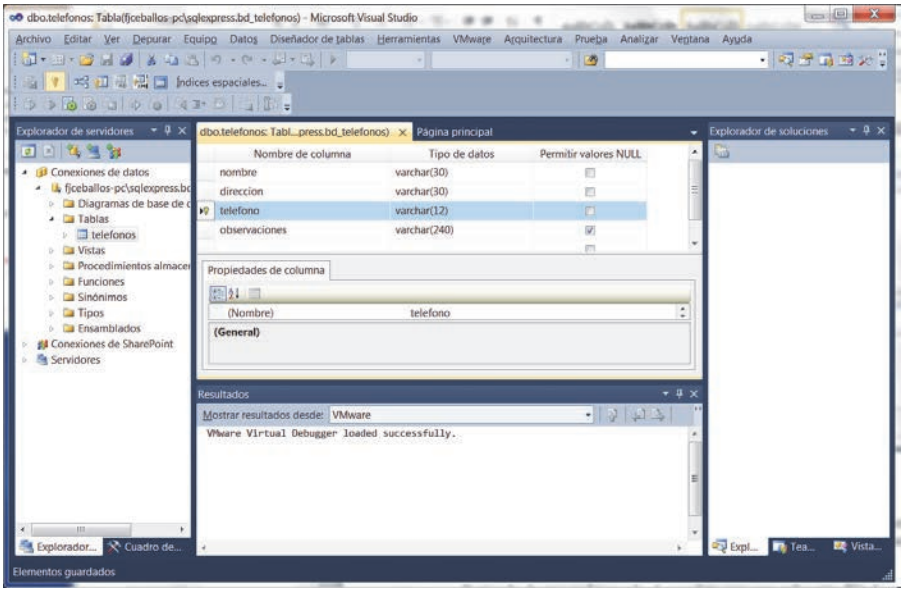
Si está utilizando Visual Studio, se visualizará la ventana siguiente. Elija como nombre del servidor .*sqlexpress*, escriba el nombre de la base de datos y haga clic en *Aceptar*.



Si está utilizando Visual Basic Express, se visualizará esta otra ventana. Elija como origen de datos *Microsoft SQL Server*, escriba el nombre de la base de datos y haga clic en *Aceptar*.



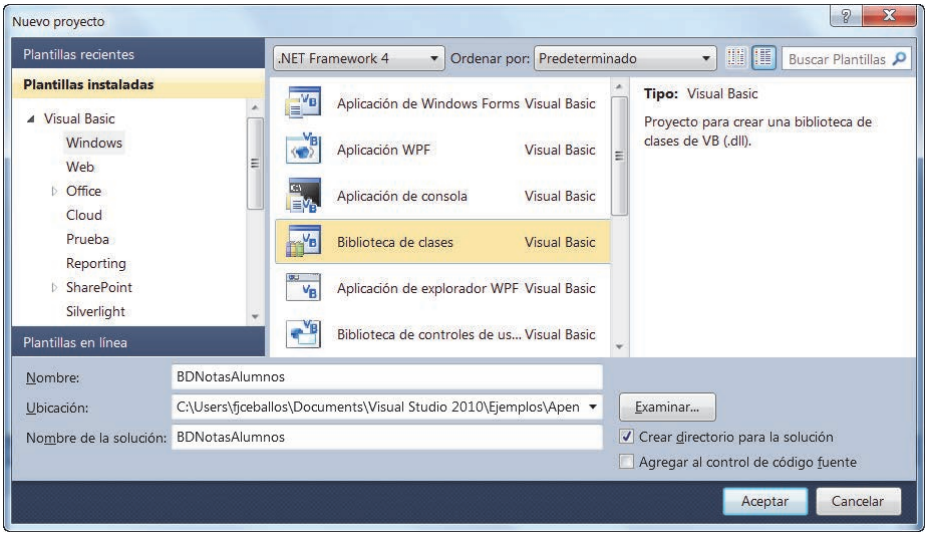
Una vez creada la base de datos, el paso siguiente es añadir las tablas. Despliegue el árbol correspondiente a la nueva conexión, haga clic con el botón secundario del ratón sobre el nodo *Tablas* y agregue una nueva tabla. Después complete el proceso de creación de la misma.



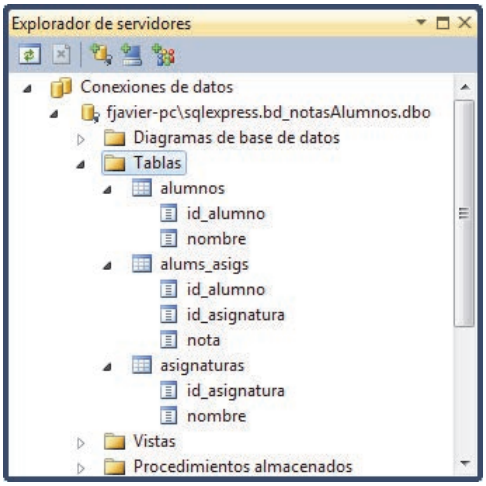
AÑADIR UN DATASET AL PROYECTO

Para acceder a una base de datos podemos escribir código basado en los objetos que proporciona ADO.NET, o bien podemos añadir un **DataSet** (conjunto de datos) al proyecto. Cuando se añade un **DataSet** al proyecto lo que sucede es que el asistente de diseño de Visual Studio crea una capa de acceso (DAL) a las tablas seleccionadas de la base de datos que nos abstrae de SQL. Dicha capa incluye todo un conjunto de clases con la funcionalidad suficiente para modificar, añadir o borrar filas en las tablas de la base de datos.

Como ejemplo, vamos a crear un proyecto *BDNotasAlumnos*, de tipo *Biblioteca de clases*, que nos proporcione una interfaz de clases Visual Basic para acceder a una base de datos formada por tres tablas, *alumnos*, *asignaturas* y *alums_asigs*, para gestionar las notas de las asignaturas en las que se ha matriculado cada uno de los alumnos (puede ver más detalles acerca de esta base de datos en el capítulo 6). Esta forma de proceder nos permitirá utilizar esta biblioteca en cualquier proyecto donde sea necesaria, simplemente añadiendo al proyecto en cuestión una referencia a la misma.



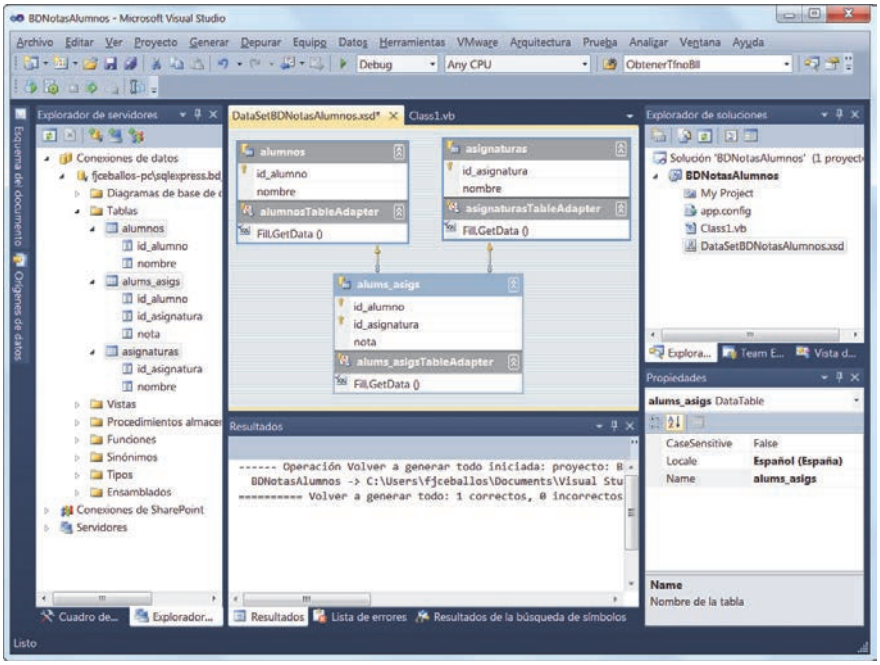
Suponiendo que tenemos creada la base de datos SQL, abrimos el explorador de servidores (explorador de base de datos) y añadimos una conexión a esta base de datos:



Para crear la capa de acceso a la base de datos, añadimos un **DataSet** al proyecto. Para ello, haga clic con el botón secundario del ratón sobre el nombre del proyecto y seleccione *Agregar* > *Nuevo Elemento* > *Conjunto de datos*, elija un nombre, por ejemplo *DataSetBDNotasAlumnos.xsd* y haga clic en *Agregar*. Observará en el *Explorador de soluciones* que se ha añadido un esquema XSD inicialmente vacío. También observará que la ventana de edición muestra el diseñador de **DataSet** para crear y editar visualmente conjuntos de datos con tipo.

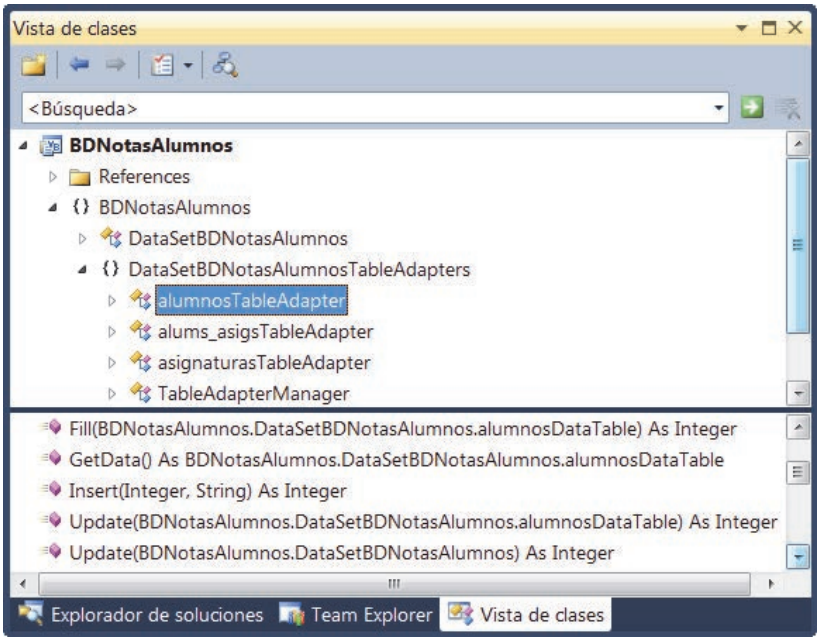
También puede realizar estas operaciones desde el menú *Datos* de Visual Studio.

Como siguiente paso, arrastre desde el *Explorador de servidores* los elementos de la base de datos con los que vaya a trabajar. En nuestro caso seleccionaremos las tres tablas.



También puede agregar nuevos elementos haciendo clic con el botón secundario del ratón sobre la superficie de diseño.

Si ahora muestra la vista de clases, podrá observar la capa de acceso a datos, formada por clases Visual Basic, que le permitirá realizar operaciones sobre los elementos de la base de datos que seleccionó y añadió al diseñador.



Esquemas XSD

Se pueden crear vistas XML de datos relacionales utilizando el lenguaje de definición de esquemas XML (XSD). Estas vistas pueden consultarse después utilizando consultas XPath (*lenguaje de rutas XML*).

Un esquema XML describe la estructura de un documento XML y también describe las distintas restricciones en los datos del documento. Todas las declaraciones de elementos deben incluirse dentro del elemento *<xsd:schema>* o *<xs:schema>*:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="DataSetBDNotasAlumnos"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

</xs:schema>
```

Un esquema XSD con anotaciones que describan la base de datos y las operaciones sobre la misma, puede utilizarlo para consultar la base de datos y devolver los resultados en forma de documento XML.

Base de datos XML

Para realizar pruebas rápidas en diferentes proyectos puede ser interesante simular una determinada base de datos con el esquema XSD correspondiente a las tablas del conjunto de datos construido, con los datos de las distintas tablas especificados en XML y con el código Visual Basic personalizado necesario para construir, a partir de la información anterior, un **DataSet** que modele la base de datos. Después, la funcionalidad proporcionada por la clase **DataSet** nos permitirá acceder a la base de datos.

Como ejemplo, partiendo de la aplicación anterior, vamos a construir en primer lugar los ficheros XML que definan el **DataSet** y los datos almacenados en las distintas tablas que incluimos en el conjunto de datos. Una forma fácil de hacer esto es construir una nueva solución con un nuevo proyecto de tipo *Aplicación WPF*, denominado, por ejemplo, *EnlaceDeDatosXML*, que incluya una referencia a la biblioteca que creamos anteriormente (para mayor facilidad, añada el proyecto *BDNotasAlumnos* a esta solución), y ejecutar desde el mismo, para cada una de las tablas, un código análogo al siguiente:

```
Imports BDNotasAlumnos
...
Dim dt1 As New DataSetBDNotasAlumnos.alumnosDataTable()
Dim t1 As New
  DataSetBDNotasAlumnosTableAdapters.alumnosTableAdapter()
t1.Fill(dt1)
dt1.WriteXmlSchema("xsd_alumnos.xml")
dt1.WriteXml("datos_alumnos.xml")
```

Una vez obtenido el esquema XSD de cada una de las tablas, fusionamos todas ellas en un único fichero; por ejemplo en *bd_notasAlumnos.xsd*:

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="bd_notasAlumnos" xmlns=""
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="bd_notasAlumnos"
    msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="alumnos">

        </xs:element>
        <xs:element name="asignaturas">

        </xs:element>
        <xs:element name="alums_asigs">
```

```
        </xs:element>
      </xs:choice>
    </xs:complexType>

    <xs:unique name="Constraint1" msdata:PrimaryKey="true">

    </xs:unique>
    <xs:unique name="Constraint2" msdata:PrimaryKey="true">

    </xs:unique>
    <xs:unique name="Constraint3" msdata:PrimaryKey="true">

    </xs:unique>
  </xs:element>
</xs:schema>
```

Hacemos lo mismo con los datos de cada una de las tablas; por ejemplo fusionamos los datos en *bd_notasAlumnos.xml*. Observe que la etiqueta que envuelve todo el documento, *bd_notasAlumnos*, coincide con la identificación del esquema XSD.

```
<?xml version="1.0" standalone="yes"?>
<bd_notasAlumnos>
  <alumnos>
    <id_alumno>1234567</id_alumno>
    <nombre>Leticia Aguirre Soriano</nombre>
  </alumnos>
  ...
  <asignaturas>
    <id_asignatura>20590</id_asignatura>
    <nombre>PROGRAMACION AVANZADA</nombre>
  </asignaturas>
  ...
  <alums_asigs>
    <id_alumno>1234569</id_alumno>
    <id_asignatura>33693</id_asignatura>
    <nota>9.5</nota>
  </alums_asigs>
</bd_notasAlumnos>
```

Ahora ya podemos eliminar de la biblioteca *BDNotasAlumnos* el **DataSet** que agregamos inicialmente y, en su lugar, añadimos a este proyecto estos dos ficheros que acabamos de crear: *bd_notasAlumnos.xsd* y *bd_notasAlumnos.xml*. En realidad no es necesario añadirlos; lo hacemos simplemente a efectos de modificaciones. Donde sí tienen que estar es en la carpeta desde la que se ejecute la aplicación WPF que haga referencia a esta biblioteca que hemos generado; en nuestro caso en *EnlaceDeDatosXML*.

Para construir un objeto **DataSet** a partir de los ficheros *bd_notasAlumnos.xsd* y *bd_notasAlumnos.xml*, añadimos a la biblioteca *BDNotasAlumnos* una nueva clase, *DataSetBDNotasAlumnos*, que almacenamos en el fichero *bd_notasAlumnosDataSet.cs*:

```
Public Class DataSetBDNotasAlumnos
  ' Construcción del DataSet a partir de
  ' bd_notasAlumnos.xsd y bd_notasAlumnos.xml
  Public Shared Function ObtenerTablaAsignaturas() As DataTable
    Return ReadDataSet().Tables("asignaturas")
  End Function

  Public Shared Function ObtenerAlumsAsigsNotas() As DataSet
    Return ReadDataSet()
  End Function

  Friend Shared Function ReadDataSet() As DataSet
    Try
      Dim ds As New DataSet()
      ds.ReadXmlSchema("../../bd_notasAlumnos.xsd")
      ds.ReadXml("../../bd_notasAlumnos.xml")
      Return ds
    Catch exc As Exception
      System.Diagnostics.Debug.WriteLine(exc.Message)
      Return Nothing
    End Try
  End Function
End Class
```

Y para implementar la lógica de negocio que permita acceder a la base de datos añadimos otra clase más, *bd_notasAlumnos*, que almacenamos en el fichero *bd_notasAlumnos.cs*:

```
Imports System.Collections.ObjectModel

Public Class bd_notasAlumnos
  Public Function ObtenerAsignatura(ID As Integer) As Asignatura
    Try
      Dim tablaAsigs As DataTable =
        DataSetBDNotasAlumnos.ObtenerTablaAsignaturas()
      Dim filaAsignaturas As DataRow =
        tablaAsigs.[Select]("id_asignatura = " & ID.ToString())(0)
      Dim asig As New Asignatura(
        CInt(filaAsignaturas("id_asignatura")),
        DirectCast(filaAsignaturas("nombre"), String))
      Return asig
    Catch exc As Exception
      System.Diagnostics.Debug.WriteLine(exc.Message)
      Return Nothing
```

```
    End Try
  End Function

  Public Function ObtenerAsignaturas() As ICollection(Of Asignatura)
    Dim tablaAsigs As DataTable =
      DataSetBDNotasAlumnos.ObtenerTablaAsignaturas()

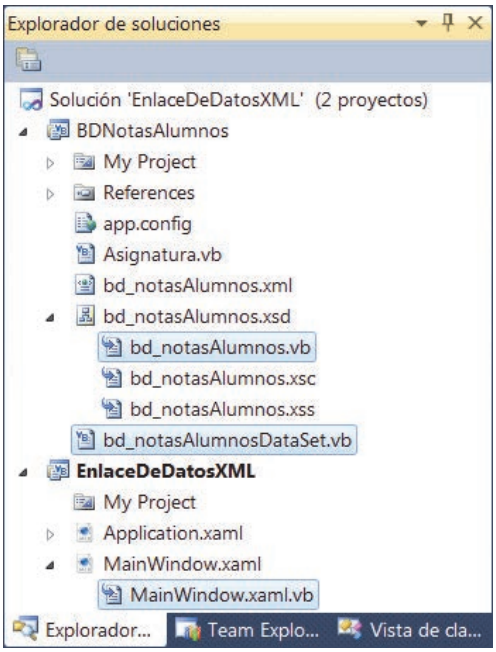
    Dim asigs As New ObservableCollection(Of Asignatura)()

    For Each fila As DataRow In tablaAsigs.Rows
      asigs.Add(New Asignatura(CInt(fila("id_asignatura")),
        DirectCast(fila("nombre"), String)))
    Next
    Return asigs
  End Function

  ' ...
End Class
```

Esta clase requiere añadir también la clase *Asignatura* que implementaremos de la misma forma que lo hicimos en el capítulo *Enlace de datos en WPF*.

En la figura siguiente se pueden observar ambos proyectos, el correspondiente a la biblioteca, *BDNotasAlumnos*, y el correspondiente a la aplicación WPF, *EnlaceDeDatosXML*, que utilizará esa biblioteca.



VISUAL WEB DEVELOPER

Visual Web Developer Express (VWD) es una herramienta enfocada exclusivamente al desarrollo de aplicaciones webs dinámicas con ASP.NET. Para ello, proporciona:

- Diseñadores visuales para crear las páginas webs.
- Un editor de código potente que muestra referencias del lenguaje a medida que se escribe código, sin tener que dejar el editor. Con este editor podrá escribir código en los diferentes lenguajes .NET (Visual VB o C#) y código HTML.
- Desarrollo rápido de aplicaciones webs integradas con bases de datos. Para ello proporciona acceso integrado a SQL Server Express.

Crear un sitio web con VWD y probar su funcionamiento es algo sencillo porque integra un servidor web, por lo que no es necesario desplegar la aplicación en el servidor IIS de Windows (*Internet Information Server*). En realidad, cualquier carpeta con páginas *.aspx* puede ser considerada virtualmente como un sitio web. Este servidor web integrado será utilizado automáticamente cuando elijamos como tipo de sitio web "sistema de archivos". Otros tipos de sitios webs son HTTP y FTP, ambos necesitan tener instalado IIS.

La forma de trabajar con esta herramienta es la misma que hemos explicado en los últimos capítulos de este libro para crear sitios y servicios webs con Visual Studio, ya que esta herramienta está integrada en Visual Studio.

INSTALACIÓN DE ASP.NET EN WINDOWS

ASP.NET queda instalado automáticamente cuando instaló Visual Studio o, en su defecto, Visual Web Developer. Ahora bien, para crear y ejecutar aplicaciones para Internet necesitará un servidor de aplicaciones, en el caso de la plataforma Windows éste es IIS (*Internet Information Server*). La instalación de IIS debe ser anterior a la de .NET Framework. Si no se hizo así, ASP.NET no estará habilitado en IIS y no podremos realizar aplicaciones ASP.NET. En este caso, la solución es registrar manualmente ASP.NET en IIS.

Registro manual de ASP.NET en IIS

Para registrar manualmente ASP.NET en IIS, abra una ventana de consola, sitúese en la carpeta *C:\WINDOWS\Microsoft.NET\Framework\vx.x.xxxxx* y ejecute:

```
aspnet_regiis.exe -i -enable
```

Esto habilitará IIS para la ejecución de ASP.NET, registrará las extensiones de ASP.NET (*aspx*, *amsx*, *asax*, etc.) y ya podremos empezar a trabajar con ASP.NET.

Si echa una ojeada a todas las opciones disponibles (*aspnet_regiis.exe -help*), podrá observar algunas muy interesantes, como *-c*, que instala las secuencias de órdenes del cliente de esta versión en el subdirectorio *aspnet_client* de todos los directorios de sitios IIS.

Si la operación anterior no solucionó el problema, pruebe a reparar la instalación actual a partir del CD de instalación del producto. Si esto tampoco solucionara el problema, sólo queda desinstalar el producto y volverlo a instalar.

APÉNDICE B

© F.J.Ceballos/RA-MA

CD

La forma de descargar el CD se indica en el prólogo.

ÍNDICE

A

AcceptsReturn, 184
AcceptsTab, 184
acceso conectado a bases de datos, 408
acceso desconectado a datos, 451
Access, 400
AccessText, 67
aceleradores, 144
Acerca de, 198, 223
Activate, 42, 43
Activated, 41, 49
actualizar datos, 766
actualizar la base de datos, 843
actualizar las filas, 525
adaptador, 402
Add, colección, 205
Added, 397
AddedItems, 438
AddHandler, 71, 73, 662
AddNew, 768
AddObject, 527
administración de sitios webs, 798
ADO.NET, 394
AdornedElementPlaceholder, 124
agregar datos, 767
agrupación de los datos, 842
agrupar los elementos de la vista, 353
alineación del texto, 27
AllowsTransparency, 44
ámbito de foco, 149
animación, 326
animaciones, 647
animar Opacity, 326
animar una transformación, 650
añadir imagen a un elemento de un menú, 175
añadir nuevos objetos, 701
añadir texto a una caja de texto, 80
añadir un nuevo registro, 848
aplicación
  crear, 857
  de negocios de Silverlight, 815
  depurar, 872
  desarrollo, 63
  instanciar sólo una vez, 51
  mínima, 6
aplicaciones XBAP, 576
app.config, 46
AppendText, 188
Application, 13, 41, 58
Application.Current, 55
ApplicationServices, 790
árbol con nodos, 864
árboles de expresiones, 481
área cliente, 42
área de trabajo, 3
argumentos en la línea de órdenes, 54
asincrónico, modelo, 724
ASP.NET
  acceso a los recursos, 746
  arquitectura, 778
  instalación, 891
aspecto de un control, 122
aspnet_regiis, 892
Assembly, 49

atajo, 155
ataques de inyección SQL, 411
atributos de anotación de datos, 621
atributos globales de una aplicación, 48, 224
Attach, 754
AttachDBFilename, 877
audio, 654
autenticación, 777, 819
  de Windows, 779, 816
  mediante formularios, 779, 816
autenticación y autorización, 675
Authenticate, 787
AuthenticationService, 790, 793, 800
AutoCompleteBox, 631
AutoPlay, 655
autorización, 777, 808
ayuda dinámica, 858

B

BAML, 16
barra de desplazamiento, 2, 251, 864
  eventos, 254
barra de estado, 181
barra de herramientas, 177
  añadir botón, 178
  diseño, 191
  orientación, 180
barra de menús, 1, 139
  crear, 139
  diseño, 189
barra de progreso, 256
barra de tareas, botón de la ventana, 42
barra de título, 2
base de datos, 887
  crear, 878, 881
  mover, 877
  XML, 821
BasedOn, 120
BeginEdit, 440
BeginInvoke, 304
BeginTime, 650
binary, 746
BinaryExpression, 484
Binding, 21, 23, 97, 98, 316, 678
BindingExpression, 114
BindingGroup, 374, 449
BindingListCollectionView, 320
Bitmap, 695
BitmapImage, 695
Block, 285
BlockUIContainer, 286
Boolean?, 220
Border, 26, 61, 324, 616
borrar datos, 768
borrar objetos, 705
borrar registros en una tabla, 389
borrar tabla de la base de datos, 389
borrar un control, 866
borrar un registro, 851
botón de opción, 232, 863
botón de pulsación, 28, 62, 863
botón predeterminado, 67
bucle de mensajes, 4, 304
Button, 26, 60, 62
ButtonBase, 60
byte[], 697

C

caché, 581
cadena de conexión, 811
caja de diálogo
  Abrir, 278
  Color, 281
  crear, 217
  Guardar, 280
  Imprimir, 282
  modal o no modal, 212
  mostrar, 219
  para mostrar un mensaje, 212
  recuperar datos, 221
caja de herramientas, 862
caja de texto, 863
  multilínea, 184
  seleccionar el contenido, 79
cajas de diálogo estándar, 278
cajas de diálogo personalizadas, 215
calculadora, 135
Calendar, 261
cambios en los datos, 522
cambios, seguimiento, 535
campos, 385
CancelEdit, 440
CanExecute, 151, 174
CanExecuteChanged, 152, 173
CanUndo, 188, 202
Canvas, 29, 61
capa de acceso a datos, 431
capa de lógica de negocio, 435

capa de presentación, 425
  desacoplar, 439
  lógica, 436
CaptureSource, 664
carpeta virtual, 578
cascada, operaciones en, 531
casilla de verificación, 227, 863
CenterOwner, 217
cerrar, 2
cerrar la aplicación, 52
CheckBox, 26, 60, 227, 245
Checked, 228
CheckStateChanged, 232
Children, 296
ChildWindow, 637
ciclo de vida de una aplicación, 49
ciclo de vida de una ventana, 41
cifrado, texto, 217
cifrar, 783
clase
  Application, 13
  Binding, 98, 316
  Border, 26
  Button, 26
  CheckBox, 26, 227
  Clipboard, 187
  ColorDialog, 281
  ComboBox, 26, 246
  ContentElement, 57
  Control, 26
  Convert, 236
  DataRow, 397
  DataTable, 397
  de entidad, 509
  de un objeto, determinar, 294
  Debug, 144
  Dispatcher, 304
  FrameworkContentElement, 57
  FrameworkElement, 27, 57
  GroupBox, 232
  Label, 26
  ListBox, 26, 238
  Menu, 139
  MenuItem, 139
  Mouse, 81
  NavigationWindow, 549
  ObjectContext, 510
  OpenFileDialog, 279
  Page, 551
  ProgressBar, 256
  RadioButton, 26, 232
  RoutedCommand, 146
  SaveFileDialog, 280
  ScrollBar, 26, 251
  Separator, 139
  Slider, 255
  TextBlock, 26
  TextBox, 26, 188
  TextBoxBase, 188
  TextRange, 298
  TimeSpan, 307
  UIElement, 57
  Validation, 109
  Window, 9
clases, vista, 860
Clear, 188, 243
Click, 4, 76
ClientAccessPolicy.xml, 729
cliente Silverlight, 756
cliente WCF, 720
Clipboard, 187
Close, 43, 143, 220, 402
Closed, 42
Closing, 42
clr-namespace, 18, 102
código subyacente, 11
código Visual Basic, insertar en XAML, 12
colección
  añadir, 243
  borrar todo, 243
  borrar, 243
  ConstraintCollection, 397
  DataColumnCollection, 397
  DataRelationCollection, 397
  DataRowCollection, 397
  DataTableCollection, 397
  de elementos, 205, 864
  insertar, 243
colecciones, 317
CollectionChanged, 354, 444
CollectionView, 319, 332
CollectionViewSource, 319, 686
colocar el control, 802
color, 20, 281
Color, propiedad, 282
ColorAnimation, 648
ColorDialog, 281
Column, 33
columnas, 385
ColumnSpan, 33

ComboBox, 26, 60, 246
ComboBoxItem, 60, 246
Command, 148, 398, 401
CommandBinding, 151
CommandParameter, 163
commands, 145
CommandTarget, 149
CommitNew, 768
compilador de marcado de WPF, 11
Completed, 839
componentes de acceso a datos, 424
comunicación entre dominios, 728
concurrencia, 532, 744, 826
ConcurrencyMode, 532
conexión abierta/cerrada, 511
conexión con la base de datos, 404
conexión, probar, 405
configuración de la aplicación, 46
configuración del cliente, 724
configuración servicio WCF, 741
ConfigurationManager, 811
confirmar, 214
conjunto de datos, 396
Connection, 398
ConstantExpression, 484
Constraint, 397
ConstraintCollection, 397
contenedor, 798
Content, 27, 550
ContentControl, 60, 620
ContentElement, 59
ContentLoader, 675
ContentRendered, 41
ContextMenu, 60, 176
contexto de datos, 99
contexto de dominio, 831
  Load, 832
contexto de objetos, 510
contextos de corta duración, 522
contrato, 713
contrato de datos, 717
contrato de servicio, 714
Control, 26, 59
control adornado, 124
control de usuario, 590
control PasswordBox, 217
control, borrar, 866
control, mover, 866
controlador de eventos, 37, 72, 74
  añadir, 75
controles, 3
controles de contenido, 620
controles de rango definido, 251
controles WPF, 226
ControlTemplate, 123
conversor, 104, 695
conversores intrínsecos, 104
Convert, 77, 104, 236, 695
ConvertBack, 104, 695
ConverterCulture, 101
convertir un entero en una cadena de caracteres en una base, 237
convertir una cadena de caracteres en una base a un entero, 236
Copy, 188
Count, 479
crear una aplicación, 857
crear una base de datos, 386, 391, 393
crear una caja de diálogo, 217
crear una tabla, 386
CREATE DATABASE, 386
CREATE TABLE, 386
credentials, 782
CSDL, 496
cuadro combinado, 247
Current, 55
CurrentChanged, 692
CurrentItem, 333, 350, 692
CurrentSource, 550
Cut, 188

D

Data Mapper, 501
Data Transfer Object, 425
DataAdapter, 398, 453
DataColumn, 397
DataColumnCollection, 397
DataContext, 99
DataContract, 718
DataErrorValidationRule, 108
DataForm, 843
DataGrid, 271, 426, 436, 685
  colección de objetos, 376
  columnas, 378
  detalles de las filas, 381
  eventos, 381
  filas, 379
  selección de celdas, 380
  validar, 447

DataGridRow, 274, 379
DataMember, 718
DataPager, 687, 842
DataReader, 398, 402
DataRelationCollection, 397
DataRow, 397, 459
DataRowCollection, 397, 464
DataSet, 452
  añadir al proyecto, 883
  métodos, 452
DataSourceProvider, 333
DataTable, 397, 459
DataTableCollection, 397
DataTemplate, 125, 264, 322
DataTrigger, 122
DataView, 462, 467
DatePicker, 261
datos jerárquicos, 328
DbProviderFactory, 399
Deactivated, 41, 49
Debug, 144
Decorator, 61
deep link, 670
deep zoom, 664
degradados, 634
delegado, 37, 693
delegado Func, 477
delegados, 475
DELETE, 389
DeleteCommand, 403
Deleted, 397
DeleteObject, 529
DependencyObject, 40, 58
depuración, 144
depurar una aplicación, 872
descripción abreviada, 29
DescriptionViewer, 625
desencadenadores, 121, 325
deshacer, 188, 202
deslizador, 255
Detach, 745
diálogo Acerca de, 223
diálogo, introducir datos, 361
diálogo, permanecer o cancelar, 373
DialogResult, 220
diccionario de recursos, 127, 130
dirección, 713
directorio virtual, 578
diseño, 29
diseño de tipo maestro-detalle, 339
Dispatcher, 304
DispatcherObject, 58, 304
DispatcherTimer, 304, 308
DispatcherUnhandledException, 49
DisplayMemberPath, 420
Dispose, 409
DockPanel, 32, 61
documento dinámico, 284
  en un fichero, 297
  modificar, 293
  navegar, 293
documentos dinámicos, 283
documentos fijos, 283
DocumentViewer, 283
DomainContext, 832
DomainDataSource, 840
DomainService, 829
DoubleAnimation, 650
DragMove, 44
DROP TABLE, 389
duración y diario de las páginas, 564
DynamicResource, 22, 127

E

EDI, 855
  opciones, 876
  personalizar, 877
EditingElementStyle, 449
editor de textos, 183
ejecutar, 16
ejecutar aplicación, 862
Elapsed, evento, 304, 305
ElementName, 98, 103, 316
elemento actual, 350
elemento adornado, 124
elemento de un menú, marcar, 202
eliminar filas, 529
Ellipse, 61
EndEdit, 441
enfocar un componente, 77
enlace, 713
  a colecciones, 317
  a una colección, 345
  con otros controles, 103
  de datos de WPF, 89
  de datos WPF, 96
  de datos, 315
  de datos, crear, 99
  información, 114

profundo, 670
TwoWay, 349
Entities, 835
Entity Client, 497
Entity Framework, 495
Entity SQL, 496
EntityCollection, 511
EntityKey, 536
EntityQuery, 835
EntityReference, 512
EntitySet, 536
entorno de desarrollo integrado, 855
Entrar, Aceptar, 218
enumeración HorizontalAlignment, 28
error durante el diseño, 587
ErrorContent, 111
Errors, 109
ErrorTemplate, 109, 124
Esc, Cancelar, 218
escribir datos en una tabla, 388
espacios de nombres, 15, 872
declarar en XAML, 102
XAML, 18
eSQL, 496
esquema del documento, 866
esquema XSD, 886
esquemas XML, 886
estilo, 119
estilos, 556
etiqueta, 27, 763
evento, 4
adjunto, 246
Checked, 228
Click, 76
controlado, 70
de propagación, directo y de túnel, 70
Initialized, 199
Loaded, 77, 196, 199
LoadingRow, 274
NavigationFailed, 567
PropertyChanged, 93
RowEditEnding, 439
SelectionChanged, 437
TextChanged, 92
Unchecked, 228
Unloaded, 199
Validation.Error, 110
eventos, 36, 69, 869
adjuntos, 38
de entrada, 70
de la aplicación, 49
de WPF, 71
del teclado, 74
enrutados, 69
lista, 870
suscribirse a, 72
EventSetter, 120
excepciones, 359
ExceptionValidationRule, 107, 367
Execute, 174
Executed, 151
ExecuteNonQuery, 401
ExecuteReader, 401
Exit, 49
explorador de bases de datos, 881
explorador de soluciones, 858
expresión de consulta, 480, 484
compilación, 487
expresiones lambda, 475
Expression, 481, 484
Extension, 474, 488
extensiones de marcado, 21
extremo, 713, 795

F

fecha y hora, 864
fechas, 260
fichas, 864
ficheros y documentos dinámicos, 297
Figure, 289
filas, 385
FileInfo, 700
FileName, 280
FileStream, 700
Fill, 404
Filter, 280, 693
FilterDescriptors, 842
FilterIndex, 280
filtrar datos, 693, 842
filtrar los elementos de una colección, 352
filtro de nombres de fichero, 280
filtros, 468
finalizar la ejecución, 16, 862
Finally, 407
FindLogicalNode, 40
FindName, 40
FindResource, 102, 128
FixedDocument, 283
Floater, 289

FlowDocument, 285
FlowDocumentPageViewer, 284
FlowDocumentReader, 283
FlowDocumentScrollViewer, 284
foco, 61, 77, 218
Focus, 77, 86, 236
Focusable, 78
FocusedElement, 86
FocusManager, 78, 149
Font..., 27
FontFamily, 196
FontSize, 197
Format, 77
FormatException, 77
FormsAuthentication, 781, 787
formularios, 3
Frame, 547, 567, 667
FrameworkContentElement, 59
FrameworkElement, 27, 40, 59
From, 478, 487
FromByteArray, 698
FromResource, 699
fuentes, 195
fuentes relativas, 118
Func, 477
función de un usuario autenticado, 805
funciones de página, 569
funciones de usuario, 820

G

GetCustomAttributes, 49
GetDefaultView, 333
GetExecutingAssembly, 49
GetHasError, 373
GetNavigationService, 551
GetObject, 47
GetRolesForCurrentUser, 804
GetString, 47
GetWindow, 43
GoBack, 550
GoForward, 550
GotFocus, 78
GotKeyboardFocus, 78
gráficos, 642
Grid, 33, 61, 615
Grid.Column, 66
Grid.Row, 66
GridSplitter, 35, 615
GridView, 262
Group, 489
Group … By, 479
GroupBox, 232
GroupDescriptors, 842
GroupName, 232
grupos de aplicaciones, 771
guardar una aplicación, 872

H

habilitar o inhabilitar los elementos de un menú, 199
Handled, 70, 75, 83
Handles, 870
HasError, 109, 373, 376
HashPasswordForStoringInConfigFile, 783
HeaderedItemsControl, 60, 328
Height, 67
herramientas, caja, 862
Hide, 43
HierarchicalDataTemplate, 264, 328
historial de navegación, 667
HorizontalAlignment, 28, 68
HorizontalContentAlignment, 28, 68
hospedar aplicaciones WPF, 545
Hyperlink, 551, 565
HyperlinkButton, 669, 674

I

ICommand, 146, 165
ICommandSource, 149
Icon, 44, 175, 204, 869
icono de la aplicación, 869
icono, añadir a la aplicación, 204
IDataErrorInfo, 108, 112
identidad utilizada por el grupo de aplicaciones, 771
IDENTITY, 746
IDisposable, 409
IEditableObject, 440
IEnumerable, 479, 493
Image, 175, 178, 191
ImageBrush, 126
imagen, añadir a un botón, 191
imagen, cargar, 699
imágenes, 619, 694
ImageSource, 695
imprimir, 282
imprimir un documento dinámico, 300

INavigationContentLoader, 675
información sobre un enlace, 114
iniciadores de colecciones, 474
iniciadores de objetos, 473
inicio de sesión, 818
InitialDirectory, 280
Initialized, 199
Inline, 289
InlineUIContainer, 290
INotifyCollectionChanged, 354
INotifyPropertyChanged, 93, 317, 349, 678
InputBinding, 156, 158
InputGesture, 156, 158
InputGestureText, 145, 155
Insert, 243
INSERT, 388
insertar filas, 526
insertar o borrar elementos en una vista, 353
InsertCommand, 403
interceptar la tecla pulsada, 84
interfaces de usuario dinámicas, 41
interfaces gráficas, 5
interfaz, 3
  ICommand, 146, 165
  IEditableObject, 440
Into, 491
introducir datos, 360
Invoke, 304
inyección de código SQL, 411
inyectar código XAML, 38
IQueryable, 479, 493
IsActive, 41, 43
IsCancel, 218
IsCheckable, 175
IsChecked, 175, 228
IsCurrentUserInRole, 804
IsDefault, 67, 218
IsEditable, 247
IsEnabled, 176, 200
IsFocusScope, 150
IsKeyDown, 86
IsKeyToggled, 86
IsKeyUp, 86
isla de datos XML, 331
IsLoggedIn, 800
IsMouseDirectlyOver, 81
IsMouseOver, 81
IsMuted, 655
IsReadOnly, 247
IsThreeState, 228
IsVisible, 78
it, 500
Items, 205
  propiedad, 240
ItemsControl, 60
ItemsSource, 272, 321
ItemsSource, propiedad, 240
IValueConverter, 104

J

Join, 479, 490
Journal, 565
JournalOwnership, 671

K

KeepAlive, 551, 564, 601
Key, 85, 156
KeyBinding, 155
Keyboard, 78, 86
KeyDown, 70, 74
KeyGesture, 156, 158
KeyUp, 74

L

Label, 26, 60, 61, 625
LargeChange, 251
LayoutMode, 869
Lazy Loading, 501
lector de datos, 401
Left, 44
Let, 492
Line, 61
LinearGradientBrush, 635
líneas de ayuda, 868
LineBreak, 224, 290
LINQ, 471
LINQ to Entities, 496
List, 286
lista, 238, 864
  acceder a sus elementos, 242
  borrar elemento, 244
  de elementos CheckBox, 245
  de los eventos, 870
  desplegable, 246, 864
    añadir elemento, 250
    borrar elemento, 250
    diseñar, 248

diseño, 241
elemento seleccionado, 240
selección múltiple, 243
ListBox, 26, 60, 238, 320
ListBoxItem, 60, 238
ListCollectionView, 320
ListView, 262
Load documento dinámico, 300
Loaded, 41, 77, 196, 199
LoadingRow, 274
LoadOperation, 832
LoadSize, 842
LocationChanged, 44
LogicalTreeHelper, 40
Login, 800
Logout, 801
LostFocus, 79
LostKeyboardFocus, 78

M

maestro-detalle, 339, 354
mage.exe, 581
Main, 13
MainWindow, 41, 55
manejador de eventos, 37
marcas de tiempo, 745
marco, 863
marco de la ventana, 2
Margin, 28, 68
máscaras, 130
Matrix, 643
MatrixTransform, 646
Max, 479
MaxDropDownHeight, 247
MaxHeight, 67
maximizar, 2
Maximum, 251
MaxWidth, 67
MediaElement, 654
MediaEnded, 655
MediaFailed, 655
MediaOpened, 655
MemberExpression, 484
membership, 796, 781
mensaje, mostrar, 144
Menu, 60, 139
menú, 138
añadir imagen a un elemento, 175
contextual, 175
controlador de un elemento, 142
de control, 2
de desbordamiento, 180
dinámico, 204
emergente, 175
señalar elemento, 201
MenuBase, 60
MenuItem, 60, 139
MenuItem.Icon, 175
menús, 137, 863
detalles, 175
diseño, 139
líneas de separación, 142
MessageBox, 144, 212
MessageBox.Show, 91
MessageBoxResult, valor retornado..., 213
metadatos, 564
MethodCallExpression, 484
método abreviado, 155
método Main, 13
método Run, 14
método Shutdown, 14
Métodos del generador de consultas, 497
métodos extensores, 474
Microsoft Access, 400
Microsoft.ACE.OLEDB, 401
Microsoft.Jet.OLEDB, 401
Microsoft.Win32, 278
Min, 479
MinHeight, 67
minimizar, 2
Minimum, 251
MinWidth, 67
modal o no modal, 212
Mode, 97, 678
modelo asincrónico, 724
modelo de datos, 343
modelo de entidades, 496
modelos de contenido, 62
Model-View-ViewModel, 166
modificadores de tamaño, 866
modificar datos, 522
modificar datos en una tabla, 388
Modified, 397
ModifierKeys, 156
Modifiers, 86, 156
mostrar datos, 764
mostrar un mensaje, 144
Mouse, 81
MouseBinding, 158

MouseDown, 80
MouseEnter, 80
MouseLeave, 81
MouseLeftButtonDown, 662
MouseLeftButtonUp, 662
MouseMove, 80
MouseUp, 80
MoveCurrentToFirst, 691
MoveCurrentToNext, 335, 351
mover el control, 866
MSL, 496
MultiTrigger, 121
Mutex, 52

N

navegación
  de fragmento, 567
  de Silverlight, 666
  de tipo web, 546
  estructurada, 569
  externa, 674
  falla, 567
  historial, 568
  personalizada de Silverlight, 665
navegar por los elementos de una vista, 351
Navigate, 550, 564, 668
NavigateUri, 565, 669
Navigating, 670, 807, 822
Navigation, 668, 672
NavigationFailed, 671
NavigationService, 547, 551
NavigationUIVisibility, 568
NavigationWindow, 547
nemónico, 67, 144
N-Layer architecture, 744, 775
NotifyOnValidationError, 109
NT AUTHORITY\Servicio de Red, 746
N-Tier architecture, 744, 775
Nullable(Of Boolean), 220

O

Object, 58
ObjectCollection, 243, 249
ObjectContext, 503
ObjectDataProvider, 358
ObjectQuery, 497
ObjectSet(Of), 510
ObjectStateEntry, 523, 536
ObjectStateManager, 523, 536
objeto aplicación, referencia, 55
objetos como orígenes de datos, 343
objetos de negocio, 428
ObservableCollection, eventos, 442
ObservableCollection, InsertItem, 442
OdbcConnection, 399
OdbcDataAdapter, 403
OleDbConnection, 399
OleDbDataAdapter, 403
OleDbDataReader, 402
OnReturn, 574
OnStartup, 52
OpenFile, 280
OpenFileDialog, 279, 700
OpenRead, 700
operadores de consulta, 478
OperationContract, 714
OptimisticConcurrencyException, 532
OracleConnection, 399
OracleDataAdapter, 403
orden, 403
orden enrutada, 145
  información adicional, 159
  parámetros, 162
orden parametrizada, 414
orden Tab, 61
ordenación de los datos, 842
ordenar una vista de elementos, 352
órdenes, 145
  enrutadas comunes, 146
Order By, 479, 487
Orientation, 255
origen de datos implícito, 100, 103
OriginalSource, 75
ORM, 495
OwnedWindows, 43, 226
Owner, 43, 226

P

Padding, 69
Page, 551
  duración, 564
PagedCollectionView, 686, 763
PageFunction, 569
PageSize, 842
página de Silverlight, 668
paginación, 687, 842
páginas, pasar datos entre, 563

Panel, 29, 60
paneles de diseño, 29
pantalla de presentación, 53
Paragraph, 286, 291
Parameter, órdenes enrutadas, 163
ParameterExpression, 484
Parameters, 414
parámetros de consulta, 841
parámetros en órdenes SQL, 414
pasos, 314
Password, 217
PasswordBox, 59, 217, 624
PasswordChar, 217
Paste, 188
Path, 98, 101, 316
patrones de diseño, 501
perfiles, 808, 821
plantilla, 123
plantillas de datos, 320
pool de conexiones, 407
Popup, 636
portapapeles, 186
posición del ratón, 81
posición inicial de la ventana, 41
Position, 656
PostgreSql, 400
predicado, 693
PresentationHost, 545
PreviewGotKeyboardFocus, 78
PreviewKeyDown, 70, 74, 85
PreviewKeyUp, 74
PreviewLostKeyboardFocus, 78
PreviewMouseDown, 80
PreviewMouseMove, 80
PreviewMouseUp, 80
PreviewTextInput, 74
PRIMARY KEY, 387
Print, 300
PrintDialog, 278, 301
PrintDocument, 283
PrintVisual, 301
procedimiento almacenado, 415, 427
producto cartesiano, 490
ProfileService, 790, 795, 808
ProgressBar, 60, 256
Property, 25
PropertyChanged, 93, 97, 316
propiedad
  asociada, 23
  auto-implementada, 473
  cambió, notificar, 93
  como atributo, 19
  como elemento, 20
  Content, 27
  de contenido, 20
  de dependencia, 24, 165
  Font..., 27
  HorizontalContentAlignment, 28
  Icon, 869
  Items, 205
  Margin, 28
  Mode, 97
  PropertyChanged, 97, 316
  Resources, 126
  StartupUri, 14
  ToolTip, 29
  UpdateSourceTrigger, 97
  Visibility, 207
  xmlns, 9
propiedades
  asociadas, 109
  de navegación, 510
  de un objeto, 867
  del proyecto, 874
  HorizontalAlignment y VerticalAlignment, 28
proveedor de datos, 398
proveedor de datos de objetos, 358
proveedor de entidades, 497
proveedor XmlDataProvider, 327
proveedores de LINQ, 493
publicar un servicio WCF, 769
publicar una aplicación Silverlight, 730
Publicar XBAP, 578
puntero, 862
punto de inserción, 77
punto final, 713, 795

Q

QueryParameters, 841

R

RadialGradientBrush, 635
RadioButton, 26, 60, 232
RangeBase, 60
Read, 402
Rectangle, 61
recursos, 22, 126
  creados mediante código, 128

de una aplicación, 47
del sistema, 129
Redo, 188
referencias, 859
reflexión, 49, 618
Refresh, 381, 534
registrar usuario, 819
registros, 385
reglas de validación, 107
rehacer, 188
rejilla de ayuda, 868
RelativeSource, 23, 118, 451
reloj, 303
despertador, 309, 312
digital, 304
Remove, 243
RemoveAt, 243
colección, 208
RemoveBackEntry, 550
RemovedItems, 438
RequiresAuthenticationAttribute, 820
RequiresRoleAttribute, 820
ResizeMode, 44, 217
ResourceDictionary, 127
ResourceManager, 47
Resources, 126
RestoreBounds, 44
Result, 574
ReturnEventArgs, 574
RIA Services, 813, 825
RichTextBox, 59, 197, 283, 297
roles, 820
RoleService, 790, 795, 804
RootVisual, 611
RotateTransform, 646
RoutedCommand, 146
RoutedEvent, 69
RoutedEventArgs, 70
RoutedEventHandler, 37, 73
RoutedUICommand, 148
Row, 33, 439, 446
RowChanged, 466
RowDeleted, 466
RowEditEnding, 439
RowFilter, 468
RowSpan, 33
RowState, 464
RowValidationRules, 450
Run, 4, 14, 224, 290, 291

S

Save documento dinámico, 299
SaveChanges, 510
SaveFileDialog, 280
ScaleTransform, 645
Scroll, 254
ScrollBar, 26, 60, 251
ScrollIntoView, 319, 335, 351
ScrollViewer, 35, 60, 258
Section, 286
SecurePassword, 217
SecureString, 217
seguimiento de los cambios, 535
seguridad en XBAP, 581
seleccionar datos de una tabla, 389
seleccionar el contenido de una caja de texto, 79
seleccionar un objeto, 867
Select, 80, 188, 487
SELECT, 389
selectAll, 79
SelectAll, 188
SelectCommand, 403
SelectedIndex, 240
SelectedIndexChanged, 520
SelectedItem, 240, 249
SelectedItems, 243
SelectedText, 80, 188
SelectedValue, 420
SelectedValuePath, 420
SelectionChanged, 271, 437
SelectionLength, 80, 188
SelectionStart, 79, 188
Selector, 60
sentencia Using, 409
señalar un elemento de un menú, 201
separadores, 142
Separator, 59, 139
seriación/deseriación, 11
ServiceContract, 714
servicio, 711
de conexiones, 407
de dominio, 815, 829, 831
de navegación, 551
de red, 747
de suscripciones, 796
WCF, 712
WCF de funciones, 804
WCF habilitado para Silverlight, 734

WCF para acceso a datos, 748
servicios de aplicación, 778, 790
configurar, 793
Servicios de objetos, 498
servicios webs y LINQ, 743
SessionEnding, 49
SetResourceReference, 129
Setter, 119, 326
Shape, 61
Show, 43, 212, 215, 226
ShowActivated, 44
ShowDialog, 41, 215, 226, 279, 700
ShowGridLines, 33
ShowInTaskbar, 42, 44, 217
ShowsNavigationUI, 550
Shutdown, 14
ShutdownMode, 52
SignOut, 787
Silverlight, 605, 708
<Object>, 613
acceso a datos, 676
administrador de identificadores de recursos, 672
aplicación fuera del explorador, 675
arquitectura de una aplicación, 609
arquitectura, 606
autenticación y autorización, 675
compilación de la aplicación, 612
comunicación entre dominios, 728
controles, 615
crear aplicación, 608
diccionario de recursos, 667
diseño de una página, 615
Frame, 666
llamadas asíncronas, 725
navegación, 676
Page, 666
página de entrada, 613
SDK, 624, 630
vista de una colección, 686
simultaneidad, 532, 744
sincronizar con CurrentItem, 333
sistema de diseño, 29
sistema de navegación Silverlight, extender, 675
sitio web, 578
administración, 798
SizeToContent, 44
SkewTransform, 646
Slider, 60, 255
SmallChange, 251
SOAP, mensaje, 712
SolidColorBrush, 649
soluciones y proyectos, 875
Sort, 468
SortDescriptors, 842
Source, 75, 98, 316, 550
Span, 291
SplashScreen, 53
SQL, 386
SQL Server, 400, 408, 746, 877
SQL Server Management Studio Express, 880
SQLCMD, 878
SqlConnection, 399
SqlDataAdapter, 403
SqlDataReader, 402
SSDL, 496
StackPanel, 30, 61, 178
Startup, 49, 51
StartupUri, 14, 50
StateChanged, 45
StaticResource, 22, 127
StatusBar, 60, 182
StatusBarItem, 60
Storyboard, 326, 648, 650
Stretch, 655
Style, 58, 119, 270, 326
subprocesos, 304
Sum, 479
suscribirse a un evento, 36, 72
System.Data.Linq, 471
System.Linq, 471
System.Linq.Expressions, 483
System.Runtime.CompilerServices, 474, 488
System.Windows, 28
System.Windows.Markup, 11
SystemColors, 129
SystemFonts, 129
SystemParameters, 129

T

Tab, 61
TabControl, 259
TabIndex, 61
tabla, 385, 864
Table, 287
Take While, 479
tamaño de los controles, 866
tamaño de una fuente, 197

tamaño de una ventana, 64
TargetNullValue, 445
TargetType, 119
tecla de acceso, 28, 61, 67
tecla Entrar, 62
tecla pulsada, interceptar, 84
temas, 130
TemplateBinding, 23
temporizador, 303
  resolución, 306
Text, 75, 77, 247
TextBlock, 26, 60, 224, 618
TextBox, 26, 59, 62, 188, 624
TextBoxBase, 59, 188
TextChanged, 76, 92, 235
TextInput, 74
texto seleccionado, 200
TextProperty, 98
TextRange, 298
TextWrapping, 184
Tick, evento, 304
TickFrequency, 255
TickPlacement, 255
ticks, 314
Timer, 304
TimeSpan, 307, 661
tip, 29
tipo anónimo, 472
tipo implícito, 472
tipos SQL, 387
ToByteArray, 697
ToDouble, 77
ToggleButton, 60
ToInt32, 236
ToList, 511
ToolBar, 60, 177
ToolBarTray, 180
Toolkit.dll, 843
ToolTip, 29, 60
Top, 44
Topmost, 45
ToString, 237
ToUpper, 237
transacciones, 417
  explícitas, 421
TransactionScope, 417
transformaciones, 643
TransformGroup, 646
TranslateTransform, 644
TreeView, 264
TreeViewItem, 264
Trigger, 121
Triggers, 325
TryCast, 76
TryFindResource, 128

U

UIElement, 58, 97
Unchanged, 397
Unchecked, 228
Undo, 188, 202
UNIQUE, 387
Unloaded, 199
Update, 404, 466
UPDATE, 388
UpdateCommand, 403
UpdateSource, 114
UpdateSourceTrigger, 97, 101
URI, 567, 672
UriMapper, 672
UriMapping, 672
User, 820
UserControl, 60, 590, 611
Using, 409

V

validación, 366
  de los datos, 107, 625
  personalizada, 116, 370
  proceso, 112, 371
  visualizar errores, 368
validar un campo de texto, 87
validar un grupo de enlaces, 374
Validate, 116, 370
ValidateUser, 801
Validation, 109, 369
Validation.Error, 110
Validation.GetHasError, 110
ValidationAttribute, 621
ValidationRule, 107, 116, 366
ValidationSummary, 625
Value, 251, 257
Value Object, 425, 501
ValueChanged, 251
var, 472
varbinary, 746
variable, declaración implícita, 472
ventana, 1

ventana activa, 41
  modal, 41
  no modal, 41
  posición, 41
ventanas, 3
  colección, 41
VerticalAlignment, 68
VerticalContentAlignment, 28, 68
VerticalScrollBarVisibility, 185
vídeo, 654, 662
View, 319
Viewbox, 35
vinculación de datos, 89
virtualizar, 359
VirtualizingStackPanel, 359
Visibility, 69, 207
vista de colección, 318, 347
vista de un DataTable, 467
vista de una colección, 333
vista PagedCollectionView, 763
vista predeterminada, 350
vista
  agrupar sus elementos, 337
  elemento actual, 333
  filtrar sus elementos, 336, 592
  navegar por sus elementos, 334
  obtener, 333
  ordenar sus elementos, 335
Visual, 58
Volume, 655

W

WCF, 711
WCF RIA Services, 813, 825
WebBrowser, 602
webcam, 664
WebContext, 816, 820
Where, 487
Width, 67
Window, 9, 60
Windows, 41
Windows Forms, 5
WindowStartupLocation, 41, 45
WindowState, 45, 55
WindowStyle, 45
WindowTitle, 551
WPF, 5, 57
WPF, XBAP y Silverlight, 545
WrapPanel, 31, 61
WriteableBitmap, 697
WriteLine, 144

X

x:Array, 23
x:Name, 592
x:Null, 22
x:Static, 23, 130
x:Type, 23
x:XData, 331
XAML, 8, 10
  extensiones de marcado, 21
  información general, 17
XamlReader, 11, 40
XamlWriter, 11
XBAP, 6, 547, 576
  publicar, 578
  seguridad, 581
  y bases de datos, 583
XCOPY, 877
XML, simular BBDD, 887
XmlDataProvider, 327
XmlElement, 334
xmlns, 9, 18
XPath, 316, 330
XPath=@..., 269, 330
XSD, 886

Ra-Ma®

Del mismo autor

• Curso de programación con **PASCAL**	ISBN: 978-84-86381-36-3 224 págs.
• Curso de programación **GW BASIC/BASICA**	ISBN: 978-84-86381-87-5 320 págs.
• Manual para **TURBO BASIC** Guía del programador	ISBN: 978-84-86381-43-1 444 págs.
• Manual para **Quick C 2** Guía del programador	ISBN: 978-84-86381-65-3 540 págs.
• Manual para **Quick BASIC 4.5** Guía del programador	ISBN: 978-84-86381-74-5 496 págs.
• Curso de programación **Microsoft COBOL**	ISBN: 978-84-7897-001-8 480 págs.
• Enciclopedia del lenguaje **C**	ISBN: 978-84-7897-053-7 888 págs.
• Curso de programación **QBASIC y MS-DOS 5**	ISBN: 978-84-7897-059-9 384 págs.
• Curso de programación **RM/COBOL-85**	ISBN: 978-84-7897-070-4 396 págs.
• El abecé de **MS-DOS 6**	ISBN: 978-84-7897-114-5 224 págs.
• Microsoft **Visual C ++** (ver. 1.5x de 16 bits) Aplicaciones para Windows	ISBN: 978-84-7897-180-0 846 págs. + 2 disquetes
• Microsoft **Visual C ++** Aplicaciones para Win32 (2ª edición)	ISBN: 978-84-7897-561-7 792 págs. + disquete
• Microsoft **Visual C ++** Programación avanzada en Win32	ISBN: 978-84-7897-344-6 888 págs. + CD-ROM
• **Visual Basic 6** Curso de programación (2ª edición)	ISBN: 978-84-7897-357-6 528 págs. + disquete
• Enciclopedia de Microsoft **Visual Basic 6**	ISBN: 978-84-7897-386-6 1.072 págs. + CD-ROM
• El lenguaje de programación **Java**	ISBN: 978-84-7897-485-6 320 págs. + CD-ROM
• El lenguaje de programación **C#**	ISBN: 978-84-7897-500-6 320 págs. + CD-ROM

Ra-Ma®

Del mismo autor

• El lenguaje de programación **Visual Basic.NET**	ISBN: 978-84-7897-525-9 464 págs. + CD-ROM
• **Java 2** Lenguaje y aplicaciones	ISBN: 978-84-7897-745-1 392 págs. + CD-ROM
• Programación orientada a objetos con **C ++** (4ª edición)	ISBN: 978-84-7897-761-1 648 págs. + CD-ROM
• **C/C++** Curso de programación (3ª edición)	ISBN: 978-84-7897-762-8 708 págs. + CD-ROM
• **Microsoft C#** Lenguaje y aplicaciones (2ª edición)	ISBN: 978-84-7897-813-7 520 págs. + CD-ROM
• **Java 2**. Interfaces gráficas y aplicaciones para Internet (3ª edición)	ISBN: 978-84-7897-859-5 718 págs. + CD-ROM
• **Aplicaciones .Net multiplataforma** (Proyecto Mono)	ISBN: 978-84-7897-880-9 212 págs. + CD-ROM
• Enciclopedia del lenguaje **C ++** (2ª edición)	ISBN: 978-84-7897-915-8 902 págs. + CD-ROM
• Enciclopedia de Microsoft **Visual C#** (3ª edición)	ISBN: 978-84-7897-986-8 1.110 págs. + CD-ROM
• Enciclopedia de Microsoft **Visual Basic** (2ª edición)	ISBN: 978-84-7897-987-5 1.090 págs. + CD-ROM
• **Microsoft Visual Basic .NET** Lenguaje y aplicaciones (3ª edición)	ISBN: 978-84-9964-020-4 520 págs. + CD-ROM
• **Java 2** Curso de programación (4ª edición)	ISBN: 978-84-9964-032-7 820 págs. + CD-ROM
• **Microsoft C#** Curso de programación (2ª edición)	ISBN: 978-84-9964-068-6 850 págs. + CD-ROM
• **Visual C#**. Interfaces gráficas y aplicaciones con WPF, WCF y Silverlight	ISBN: 978-84-9964-203-1 958 págs. + CD-ROM

INSTALACIÓN

Para instalar el kit de desarrollo de Visual Basic y los ejemplos de este libro descargue la plataforma de desarrollo de la dirección: *http://www.microsoft.com/express/*.

PLATAFORMA WINDOWS

Instalación de .NET Framework SDK

Hay que instalar .NET Framework Redistributable Package antes de instalar .NET Framework SDK. El primer paquete incluye todo lo necesario para ejecutar aplicaciones desarrolladas con .NET Framework. El segundo paquete incorpora todo lo necesario para escribir, construir, verificar y desplegar aplicaciones desarrolladas con.NET Framework.

Para realizar la instalación, siga las instrucciones mostradas por el asistente de instalación. Esta instalación se realiza en la carpeta *Microsoft.NET* de Windows y en la carpeta *Microsoft.NET* de archivos de programas.

Instalación de Microsoft Visual Studio

La instalación de Visual Studio (el paquete completo o las versiones *Express* que necesite) no requiere la instalación previa del SDK porque está incluido en éste. Descargue de Internet los paquetes Visual Basic 2010 Express y Visual Web Developer 2010 Express (ambos incluyen SQL Server Express y la ayuda en línea), o bien, si tiene acceso, Visual Studio 2010, e instálelos.

Ejemplos del libro

Los ejemplos del libro puede instalarlos en la carpeta *Projects* de Visual Studio o los puede recuperar directamente desde el CD cuando los quiera consultar. La forma de descargar el CD se indica en el prólogo.

LICENCIA

Todo el contenido de este CD, excepto los ejemplos del libro, es propiedad de las firmas que los representan (Microsoft, etc.). La inclusión en este libro se debe a su gentileza y es totalmente gratuita y con la finalidad de apoyar el aprendizaje del software correspondiente. Para obtener más información y actualizaciones visite las direcciones indicadas en dicho software:

http://www.microsoft.com/es/es/default.aspx

Al realizar el proceso de instalación, haga el favor de consultar el acuerdo de licencia para cada uno de los productos.

WEB DEL AUTOR: http://www.fjceballos.es

En esta web podrá echar una ojeada a mis publicaciones más recientes y acceder a la descarga del software necesario para el estudio de esta obra así como a otros recursos.

Microsoft®

Visual Basic™

Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight

Microsoft Visual Basic es una evolución del lenguaje Visual Basic que continúa la tradición de ofrecer una manera rápida y fácil de crear aplicaciones basadas en .NET Framework, permitiendo a los desarrolladores generar de manera productiva aplicaciones Windows para la Web y para dispositivos móviles, con seguridad de tipos y orientadas a objetos.

Las últimas versiones (2005, 2008 ó 2010) han aportado nuevas características para el desarrollo rápido de aplicaciones. Con ellas se permite: la eliminación garantizada de recursos, la sobrecarga de operadores, los tipos genéricos, los eventos personalizados, la declaración implícita de variables locales, los tipos anónimos, la propiedades auto-implementadas, etc., y el acceso rápido, gracias a la opción My, a las tareas frecuentes de .NET Framework relacionadas, por ejemplo, con la aplicación y su entorno durante la ejecución. Además proporciona interoperabilidad de lenguajes, recolección de objetos no referenciados, seguridad mejorada y control de versiones.

Al mismo tiempo, Microsoft ha publicado una gama de productos denominada Express, de descarga gratuita, que está enfocada a programadores no profesionales. Estos entornos de desarrollo son Visual Basic, Visual C#, Visual C++, Visual Web Developer (para el desarrollo de aplicaciones web) y SQL Server Express, además de un nuevo producto llamado Visual Studio 2010 Express for Windows Phone para el desarrollo de aplicaciones para los dispositivos móviles basados en Windows Phone. Los desarrolladores profesionales podrán elegir también entre las versiones profesionales de Visual Studio, en las que encontrarán herramientas específicas para arquitectos, desarrolladores o probadores, o herramientas de software de ciclo de vida.

Visual Basic - Interfaces gráficas y aplicaciones para Internet con WPF, WCF y Silverlight es un libro totalmente centrado en las tecnologías WPF, WCF y Silverlight, para aprender a desarrollar aplicaciones WPF y Silverlight utilizando herramientas de diseño rápido como Microsoft Visual Studio 2010 (o en su defecto Visual C# 2010 Express, Visual Web Developer 2010 Express y SQL Server Express), que le permitirá crear aplicaciones cliente Windows tradicionales con interfaces gráficas muy ricas utilizando el código de marcado XAML, utilizando la tecnología de enlace a datos potente y sencilla propuesta por WPF, aplicaciones cliente servidor, aplicaciones para acceso a bases de datos (ADO.NET), acceso a bases de datos utilizando LINQ más Entity Framework, aplicaciones que utilizan un modelo de navegación tipo Web (páginas web), aplicaciones para la Web utilizando la tecnología Silverlight, servicios WCF, y a desarrollar aplicaciones Silverlight de N capas y N niveles, que tienen que acceder a una base de datos a través de servicios WCF.



Podrá descargarse de **www.ra-ma.es**, en la página web correspondiente al libro, un CD-ROM con los ejemplos realizados, así como las direcciones para la descarga del software necesario para que el lector pueda reproducirlos durante el estudio.

9 788499 642048

ra-ma.es

Ra-Ma®