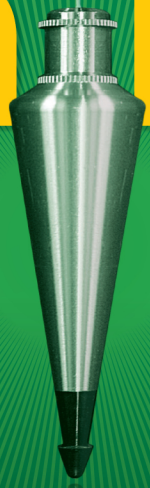
Microsoft

Designing and Developing Windows® Applications Using Microsoft® .NET Framework 4

Tony Northrup, Matthew A. Stoecker

MCPD

Exam Ref



EXAM

70-518

Exam Ref 70-518

Designing and Developing Windows® Applications Using Microsoft® .NET Framework 4

Professional-level prep for the professional-level exam.

Prepare for MCPD Exam 70-518—and help demonstrate your real-world mastery of Windows application design and development with .NET Framework 4. Designed for experienced, MCTS-certified professionals ready to advance their status—*Exam Ref* focuses on the critical-thinking and decision-making acumen needed for success at the MCPD level.

Focus on the expertise measured by these objectives:

- Designing the Layers of a Solution
- Designing the Presentation Layer
- Designing the Data Access Layer
- Planning a Solution Deployment
- Designing for Stability and Maintenance

Exam Ref features:

- Focus on job-role expertise
- Organized by exam objectives
- Strategic, what-if scenarios
- 15% exam discount from Microsoft. Offer expires 12/31/2016. Details inside.

CERTIFICATION

The Microsoft Certified Professional Developer (MCPD) certification helps validate the comprehensive skills needed to develop desktop applications using Microsoft Visual Studio® and .NET Framework 4.

JOB ROLE

Professionals certified as *Windows Developer 4* build desktop applications using Windows Presentation Foundation and Windows Forms technologies.

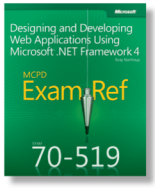
REQUIRED EXPERIENCE

Successful candidates generally have three or more years of real-world experience.

See full details at:
microsoft.com/learning/certification

MEET THE FAMILY

- Train
- Prep
- Practice



- Prep
- Optional Practice*

** Select titles coming soon*

- Review

About the Authors

Tony Northrup, MCPD, MCITP, MCSE, CISSP, is a consultant and the author of more than 25 books on Windows and web development, networking, and security.

Matthew A. Stoecker, MCP, has written numerous books and articles on Microsoft Visual Basic®, Visual C#®, Windows Forms, and Windows Presentation Foundation, including several *Training Kits*.

ISBN: 978-0-7356-5723-6

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Certification/
Microsoft Visual Studio

Microsoft® Visual Studio® 2010

Microsoft®

Microsoft®

MCPD 70-518 Exam Ref:

Designing and Developing Windows® Applications Using Microsoft® .NET Framework 4

Tony Northrup
Matthew A. Stoecker

ISBN: 978-0-7356-5723-6

1 2 3 4 5 6 7 8 9 QG 6 5 4 3 2 1

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions and Developmental Editor: Ken Jones

Production Editor: Holly Bauer

Editorial Production: S4Carlisle Publishing Services

Technical Reviewer: Bill Chapman

Copyeditor: Susan McClung

Indexer: Potomac Indexing, LLC

Cover Composition: Karen Montgomery

Illustrator: S4Carlisle Publishing Services

Contents at a Glance

	Introduction	*xv*
	Preparing for the Exam	*xviii*
CHAPTER 1	**Designing the Layers of a Solution**	**1**
CHAPTER 2	**Designing the Presentation Layer**	**89**
CHAPTER 3	**Designing the Data Access Layer**	**173**
CHAPTER 4	**Planning a Solution Deployment**	**225**
CHAPTER 5	**Designing for Stability and Maintenance**	**265**
	Index	*303*

Contents

Introduction **xv**

Microsoft Certified Professional Program *xv*

Acknowledgments *xvi*

Support and Feedback *xvi*

Preparing for the Exam **xviii**

Chapter 1 Designing the Layers of a Solution **1**

Objective 1.1: Design a Loosely Coupled Layered Architecture 2

Designing Service-Oriented Architectures 2

Providing Separation of Concern 4

Designing a System Topology 4

Choosing Between Presentation and Business Logic 6

Using WCF Routing 8

Understanding BizTalk Server 10

Objective Summary 11

Objective Review 11

Objective 1.2: Design Service Interaction 13

Designing Service and Method Granularity 14

Choosing Protocols and Binding Types 16

Using REST 18

Using Message and Data Contracts 19

Using Custom SOAP Headers 22

Managing Data Integrity 24

Choosing Synchronous vs. Asynchronous 24

Choosing a Message Exchange Pattern 25
Versioning 25
Hosting WCF Services 27
Objective Summary 28
Objective Review 28
Objective 1.3: Design the Security Implementation 30
Planning for User Account Control 31
Designing for Least Privilege 31
Understanding Process Identity 35
Understanding Impersonation and Delegation 36
Implementing Authorization 41
Planning Role Management 44
Using Cryptography 45
Objective Summary 49
Objective Review 50
Objective 1.4: Design for Interoperability with External Systems. 52
Accessing Assemblies from Unmanaged Code 52
Accessing COM Objects 53
Objective Summary 54
Objective Review 54
Objective 1.5: Design for Optimal Processing 56
Planning for Long-Running Processes 56
Scaling Applications 60
Moving to the Cloud 63
Using Queuing 63
Minimizing Latency 64
Using a Service Bus 65
Objective Summary 66
Objective Review 66
Objective 1.6: Design for Globalization and Localization 69
Choosing Between *CurrentCulture* and *CurrentUICulture* 70
Format Text for Differing Cultures 71
Translating Applications 72
Working with Time 72

Comparing Data 73
Designing Databases for Globalization 74
Objective Summary 75
Objective Review 75
Chapter Summary 78
Answers. 80
Objective 1.1: Review 80
Objective 1.1: Thought Experiment 81
Objective 1.2: Review 81
Objective 1.2: Thought Experiment 82
Objective 1.3: Review 82
Objective 1.3: Thought Experiment 83
Objective 1.4: Review 84
Objective 1.4: Thought Experiment 84
Objective 1.5: Review 85
Objective 1.5: Thought Experiment 86
Objective 1.6: Review 86
Objective 1.6: Thought Experiment 87

Chapter 2 Designing the Presentation Layer 89

Objective 2.1: Choose the Appropriate Windows Technology 90
Windows Forms 90
WPF 90
Choosing Between Windows Forms and WPF 92
Interoperating Between Windows Forms and WPF 92
Choosing a Presentation Pattern 97
Objective Summary 99
Objective Review 99
Objective 2.2: Design the UI Layout and Structure. 100
Evaluate the Conceptual Design 100
Designing for Inheritance and the Reuse of Visual Elements 101
Creating a Resource Dictionary 108
Designing for Accessibility 109
Deciding When Custom Controls Are Needed 111

Objective Summary 112
Objective Review 112
Objective 2.3: Design Application Workflow 113
Implementing User Navigation 114
Navigation Applications in WPF 117
Using *PageFunction* Objects 124
Simple Navigation and Structured Navigation 125
Designing for Different Input Types 126
Objective Summary 127
Objective Review 127
Objective 2.4: Design Data Presentation and Input 129
Designing Data Validation 129
Design a Data Binding Strategy 134
Managing Data Shared Between Forms 139
Managing Media 140
Objective Summary 140
Objective Review 141
Objective 2.5: Design Presentation Behavior 143
Determine Which Behaviors Will Be Implemented and How 143
Creating Attached Behaviors 147
Implementing Drag-and-Drop Functionality 148
Objective Summary 154
Objective Review 154
Objective 2.6: Design for UI Responsiveness 155
Offloading Operations from the UI Thread and Reporting Progress 156
Using *Dispatcher* to Access Controls Safely on Another Thread in WPF 161
Avoiding Unnecessary Screen Refreshes 162
Determining Whether to Sort and Filter Data on the Client or Server 163
Addressing UI Memory Issues 164
Objective Summary 165
Objective Review 165
Chapter Summary 167

Answers. 168
Objective 2.1: Review 168
Objective 2.1: Thought Experiment 168
Objective 2.2: Review 168
Objective 2.2: Thought Experiment 169
Objective 2.3: Review 169
Objective 2.3: Thought Experiment 170
Objective 2.4: Review 170
Objective 2.4: Thought Experiment 171
Objective 2.5: Review 171
Objective 2.5: Thought Experiment 171
Objective 2.6: Review 172
Objective 2.6: Thought Experiment 172

Chapter 3 Designing the Data Access Layer 173

Objective 3.1: Choose the Appropriate Data Access Strategy 174
Understanding .NET Data Access Technologies 174
Supporting Different Data Sources 177
Choosing a Data Access Strategy 178
Objective Summary 179
Objective Review 179

Objective 3.2: Design the Data Object Model. 181
Mapping to Persistent Storage 182
Designing a Schema Change Management Strategy 184
Abstracting from the Service Layer 185
Objective Summary 187
Objective Review 187

Objective 3.3: Design Data Caching 189
Understanding Caching 189
Using *MemoryCache* 190
Caching Web Services 190
Objective Summary 191
Objective Review 192

Objective 3.4: Design Offline Storage and Data Synchronization 194
Determining the Need for Offline Data Storage 194
Using Sync Framework 195
Designing Synchronization 198
Objective Summary 201
Objective Review 201
Objective 3.5: Design for a Concurrent Multiuser Environment 203
Planning for Multiuser Conflicts 203
Understanding Deadlock Conflicts 205
Designing Concurrency for Web Services 206
Using Cross-Tier Distributed Transactions 207
Objective Summary 208
Objective Review 208
Objective 3.6: Analyze Data Services for Optimization 210
Understanding ORM Performance 211
Understanding Lazy and Eager Loading 211
Optimizing Round-Trips 213
Objective Summary 214
Objective Review 214
Chapter Summary 216
Answers 217
Objective 3.1: Review 217
Objective 3.1: Thought Experiment 218
Objective 3.2: Review 218
Objective 3.2: Thought Experiment 219
Objective 3.3: Review 219
Objective 3.3: Thought Experiment 220
Objective 3.4: Review 220
Objective 3.4: Thought Experiment 221
Objective 3.5: Review 221
Objective 3.5: Thought Experiment 222
Objective 3.6: Review 222
Objective 3.6: Thought Experiment 223

Chapter 4 Planning a Solution Deployment 225

Objective 4.1: Define a Client Deployment Strategy 226
Understanding Installation Methods 226
Choosing an Installation Method 231
Deploying the .NET Framework 232
Deploying COM Objects 234
Objective Summary 235
Objective Review 235

Objective 4.2: Plan a Database Deployment 237
Understanding Database Deployment Files 237
Using SQL Scripts 237
Using the Vsdbcmd.exe Tool 238
Using Data-Tier Projects 239
Using SQL Server Database Projects 239
Publishing Databases from Server Explorer 240
Publishing Databases with a WCF Web Service 241
Understanding Deployment Conflicts 242
Deploying an Embedded Database Privately 242
Objective Summary 244
Objective Review 244

Objective 4.3: Design a Solution Update Strategy 246
Updating ClickOnce Applications 247
Updating with Windows Installer 248
Packaging Shared Components 248
Checking for Windows Installer Updates 249
Updating Shared Components 249
Designing Web Services for Updates 249
Objective Summary 251
Objective Review 251

Objective 4.4: Plan for N-Tier Deployment 253
Designing a Physical Topology 254
Determining Component Installation Order 256
Objective Summary 256
Objective Review 257

Chapter Summary 259
Answers. 260
Objective 4.1: Review 260
Objective 4.1: Thought Experiment 260
Objective 4.2: Review 261
Objective 4.2: Thought Experiment 262
Objective 4.3: Review 262
Objective 4.3: Thought Experiment 263
Objective 4.4: Review 263
Objective 4.4: Thought Experiment 264

Chapter 5 Designing for Stability and Maintenance 265

Objective 5.1: Design for Error Handling 265
Designing an Exception Handling Strategy 266
Handling Exceptions Across Tiers 267
Collecting User Feedback 270
Creating Custom Exception Classes 272
Processing Unhandled Exceptions 272
Objective Summary 273
Objective Review 274
Objective 5.2: Evaluate and Recommend a Test Strategy 275
Understanding Black Box and White Box Testing 276
Understanding Functional Tests 277
Understanding UI Tests 279
Understanding Performance Tests 281
Understanding Code Coverage 282
Objective Summary 283
Objective Review 283

Objective 5.3: Design a Diagnostics and Monitoring Strategy 285
Providing Monitoring Information 285
Providing Usage Reporting 291
Choosing Distributed or Centralized Logging 292
Designing a Diagnostics and Monitoring Strategy 293
Profiling .NET Applications 294
Objective Summary 295
Objective Review 295
Chapter Summary 298
Answers 299
Objective 5.1: Review 299
Objective 5.1: Thought Experiment 299
Objective 5.2: Review 300
Objective 5.2: Thought Experiment 300
Objective 5.3: Review 301
Objective 5.3: Thought Experiment 302

Index *303*

Introduction

Most development books take a very low-level approach, teaching you how to use individual classes and accomplish fine-grained tasks. Like the Microsoft 70-518 certification exam, this book takes a high-level approach, building on your knowledge of lower-level Microsoft Windows application development and extending it into application design. Both the exam and the book are so high-level that there is very little coding involved. In fact, most of the code samples this book provides simply illustrate higher-level concepts.

The 70-518 certification exam tests your knowledge of designing and developing Windows applications. By passing this exam, you will prove that you have the knowledge and experience to design complex, multitier Windows applications using Microsoft technologies. This book will review every concept described in the exam objective domains, such as the following:

- Designing the layers of a solution
- Designing the Presentation layer
- Designing the Data access layer
- Planning a solution deployment
- Designing for stability and maintenance

This book covers every exam objective, but it does not necessarily cover every exam question. Microsoft regularly adds new questions to the exam, making it impossible for this (or any) book to provide every answer. Instead, this book is designed to supplement your relevant independent study and real-world experience. If you encounter a topic in this book that you do not feel completely comfortable with, you should visit any links described in the text and spend several hours researching the topic further using MSDN, blogs, and support forums. Ideally, you should also create a practical application with the technology to gain hands-on experience.

Microsoft Certified Professional Program

Microsoft certifications provide the best method for proving your command of current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies. Computer professionals who become Microsoft-certified are recognized as experts and are sought after industrywide. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO **OTHER MICROSOFT CERTIFICATIONS**

For a full list of Microsoft certifications, go to *www.microsoft.com/learning/mcp/default.asp*.

Acknowledgments

First and foremost, I'd like to thank Ken Jones at O'Reilly for his work to design the Exam Ref book series, for choosing me (once again) as an author, and for his work as an editor. It's been great to work with you, as always, Ken!

I'd also like to thank Bill Chapman, the technical reviewer, Holly Bauer, the production editor, Dan Fauxsmith, the production manager, and Susan McClung, the copyeditor.

Finally, I must thank my friends and family for their support, especially Chelsea and Madelyn Knowles (for their support, patience, and companionship) and John and Linda Antonino (for always being gracious hosts).

Support and Feedback

The following sections provide information on errata, book support, feedback, and contact information.

Errata

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735657236

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter:

http://twitter.com/MicrosoftPress

Preparing for the Exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. Although there is no substitute for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you augment your exam preparation plan by using a combination of available study materials and courses. For example, you might use the Exam Ref and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

CHAPTER 2

Designing the Presentation Layer

The Presentation layer is the layer that directly interacts with the user. The user must be able to assimilate the data presented to him quickly and easily through the Presentation layer and also must be able to input data efficiently into the application. The Presentation layer itself is responsible for validating user input, maintaining responsiveness, and providing cues that enable an easy and accessible user experience. In this chapter, you will learn some of the important aspects of designing a Presentation layer. You will learn to choose between Windows Forms and Windows Presentation Foundation (WPF) for the technology used to build the Presentation layer, to design application layout and workflow, to handle data within the Presentation layer, and to develop a well-crafted and responsive user experience.

Objectives in this chapter:

- Objective 2.1: Choose the appropriate Windows technology
- Objective 2.2: Design the UI layout and structure
- Objective 2.3: Design application workflow
- Objective 2.4: Design data presentation and input
- Objective 2.5: Design presentation behavior
- Objective 2.6: Design for UI responsiveness

Real World

The Presentation layer is the only layer of a distributed application that the user sees, and thus from a user standpoint is the most important. When building a Presentation layer, I always try to know my audience and keep the needs of my users paramount. The slickest, best-looking application is useless if it doesn't do what the users need.

Objective 2.1: Choose the Appropriate Windows Technology

When creating applications for the desktop, today's developer has two technologies to choose from: Windows Forms and WPF. Each technology provides its own set of advantages and drawbacks. In this section, you will learn the primary differences between the two technologies and how to decide which is most appropriate for your business situation.

This objective covers how to:

- Choose between Windows Forms and WPF.
- Identify areas for possible interoperation between WPF and Windows Forms.

Windows Forms

Windows Forms are the basis for most Microsoft Windows applications and can be configured to provide a variety of user interface (UI) options. The developer can create forms of various sizes and shapes and customize them to the user's needs. Windows Forms is the older of the two Windows development technologies currently supported by Microsoft, and many skilled developers are available for creating and maintaining Windows Forms projects.

Windows Forms are the basic building blocks of the UI. They provide a container that hosts controls and menus and allow you to present an application in a familiar and consistent fashion. Forms can receive user input in the form of keystrokes or mouse interactions and can display data to the user through hosted controls. Most applications that require sustained user interaction will include at least one Windows Form, and complex applications frequently require several forms to allow the program to execute in a consistent and logical fashion.

Windows Forms have no inherent support for changing styles. Thus, if the look and feel of the application must change according to conditions on a particular client desktop, considerable coding will be required.

However, Windows Forms do provide excellent support for localization and globalization. You can create applications easily that display alternate strings and images based on the locations of deployment.

Navigation in Windows Forms typically involves switching between multiple individual forms. While this allows for parts of the application to be parceled out and presented to the user as discrete functional units, it can also create a disjointed kind of user experience. A more cohesive user experience is possible with Windows Forms using Multiple Document Interface (MDI) Forms.

WPF

WPF is the successor to Windows Forms for desktop application development. WPF applications differ from traditional Windows Forms applications in several ways. The most notable of these is that the code for the UI is separate from the code for application

functionality. Although the code for the functionality of a project can be defined using familiar languages such as Microsoft Visual Basic .NET or Microsoft Visual C#, the UI of a WPF project typically is defined using a relatively new declarative syntax called Extensible Application Markup Language (XAML).

Three basic types of applications can be created with WPF:

- **Windows applications** The most similar to Windows Forms applications. Windows applications are Windows-driven and provide a user experience that is familiar to Windows users and developers alike. Multiple windows can be open at any given time, and there is no built-in sense of navigation or history.
- **Navigation applications** Provide a page-based user experience, similar to the experience of using a website. Typically, only a single page can be open at any given time, and the journal functionality keeps a record of pages visited and allows back-and-forth navigation. Unlike a website, however, a Navigation application is a compiled application that runs on your desktop computer and, like a Windows application, has full access to the resources of your computer.
- **XAML Browser Applications (XBAPs)** Similar to Navigation applications, but they are designed to run in Windows Internet Explorer. These applications can be deployed to a server or to a website and are downloaded when instantiated. Applications of this type do not have full access to a computer's resources. XBAPs run under a partial-trust environment, and resources such as the file system and the registry are inaccessible by XBAPs.

The choice of an application type depends upon several factors, the two most important of which are user experience and application requirements.

- **User experience** Determines whether you choose a Windows application or a page-based application. For a user experience that most closely resembles a traditional Windows Forms application, a Windows application is the best choice. This application type allows you to create a menu-driven, multiwindow application that combines the rich functionality of a desktop application with the rich user experience that WPF provides. For a user experience that more closely resembles a website, you should choose a page-based application. Navigation applications and XBAPs provide built-in navigational functionality that allows you to structure the application paralleling a task, such as in an Internet shopping application or a wizard.
- **Application requirements** If an application requires access to system resources that fall outside the Internet security zone, then an XBAP is not a good choice—a better choice would be a Windows application or a Navigation application. On the other hand, XBAPs allow you to deploy the application to a web server and have users start it from a hyperlink, thus making it easily accessible to a large-scale audience. If your application does not require access to system resources, an XBAP might be a good choice.

All types of WPF applications provide built-in support for changing styles and themes. Thus, if you want your application to respond to the style or theme of the desktop, it is a fairly simple matter to incorporate that functionality.

WPF applications have good support for localization and globalization, but support for this functionality is not as built-in as it is for Windows Forms. Thus, localizing an extensive WPF application will take more time and developer resources than a similarly scoped Windows Forms application.

Choosing Between Windows Forms and WPF

When choosing a technology for a client application, you must take into account several considerations. What are the skills of your developer force? Must the application be localized? What kind of support for styles and themes is needed? What sort of navigational experience is required? The following table illustrates the relative strengths of Windows Forms and WPF to help you make that decision.

TABLE 2-1 Important Properties of the *Style* Class

Criterion	Windows Forms	WPF
Adoption among the developer community	Strong	Growing
Support for localization and globalization	Excellent	Fair
Support for changing styles and themes	No	Excellent
Support for a traditional Windows-based client application	Yes	Yes
Support for navigation through a page-based interface	No	Yes, through WPF Navigation applications
Support for navigation through a multiple document interface (MDI application)	Yes	No

Interoperating Between Windows Forms and WPF

In some cases, you might want to use WPF elements in a Windows Forms application, or Windows Forms elements in a WPF application. You can use the built-in interoperation functionality of the Microsoft .NET Framework easily to incorporate these elements as you choose.

Incorporating WPF Elements into a Windows Forms Application

You might want to incorporate WPF elements into your existing Windows Forms application (for example, a user control developed using WPF that takes advantage of specialized WPF behaviors in what is otherwise a Windows Forms application). You can add preexisting WPF user controls to your Windows Forms project by using the *ElementHost* control. As the name implies, the *ElementHost* control hosts a WPF element.

The most important property of *ElementHost* is the *Child* property. The *Child* property indicates the type of WPF control to be hosted by the *ElementHost* control. If the WPF control to be hosted is in a project that is a member of the solution, you can set the *Child* property

in the Property Grid. Otherwise, the *Child* property must be set to an instance of the WPF control in code, as shown here:

Sample of Visual Basic.NET Code

```
Dim aWPFcontrol As New WPFProject.UserControl1
ElementHost1.Child = aWPFcontrol
```

Sample of C# Code

```
WPFProject.UserControl1 aWPFcontrol = new WPFProject.UserControl1;
ElementHost1.Child = aWPFcontrol;
```

Some of the ambient properties of Windows Forms controls have WPF equivalents. These ambient properties are propagated to the hosted WPF controls and exposed as public properties on the *ElementHost* control. The *ElementHost* control translates each Windows Forms ambient property to its WPF equivalent. For more information, see Windows Forms and WPF Property Mapping at *http://msdn.microsoft.com/library/ms751565.aspx* in the online MSDN library.

Incorporating Windows Forms Elements in a WPF Application

Although WPF provides a wide variety of useful controls and features, you might find that some familiar functionality that you used in Windows Forms programming is not available. Notably absent are controls such as *MaskedTextBox* and *PropertyGrid,* as well as simple dialog boxes. Fortunately, you still can use many Windows Forms controls in your WPF applications.

Using Dialog Boxes in WPF Applications

Dialog boxes are one of the most notable things missing from the WPF menagerie of controls and elements. Because dialog boxes are separate UIs, however, they are relatively easy to incorporate into your WPF applications.

File Dialog Boxes

The file dialog box classes, *OpenFileDialog* and *SaveFileDialog,* are components that you want to use frequently in your applications. They allow you to browse the file system and return the path to the selected file. The *OpenFileDialog* and *SaveFileDialog* classes are very similar and share most important members. Important properties of the file dialog boxes are shown in Table 2-2, and important methods are shown in Table 2-3.

TABLE 2-2 Important Properties of the File Dialog Boxes

Property	Description
AddExtension	Gets or sets a value indicating whether the dialog box automatically adds an extension to a file name if the user omits the extension.
CheckFileExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a file name that does not exist.

Property	Description
CheckPathExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a path that does not exist.
CreatePrompt	Gets or sets a value indicating whether the dialog box prompts the user for permission to create a file if the user specifies a file that does not exist. Available only in *SaveFileDialog*.
FileName	Gets or sets a string containing the file name selected in the file dialog box.
FileNames	Gets the file names of all selected files in the dialog box. Although this member exists for both the *SaveFileDialog* and the *OpenFileDialog* classes, it is relevant only to the *OpenFileDialog* class because it is possible to select more than one file only in *OpenFileDialog*.
Filter	Gets or sets the current file name filter string, which determines the choices that appear in the Save As File Type or Files Of Type box in the dialog box.
InitialDirectory	Gets or sets the initial directory displayed by the file dialog box.
Multiselect	Gets or sets a value indicating whether the dialog box allows multiple files to be selected. Available only in *OpenFileDialog*.
OverwritePrompt	Gets or sets a value indicating whether the Save As dialog box displays a warning if the user specifies a file name that already exists. Available only in *SaveFileDialog*.
ValidateNames	Gets or sets a value indicating whether the dialog box accepts only valid Win32 file names.

TABLE 2-3 Important Methods of the File Dialog Boxes

Method	Description
OpenFile	Opens the selected file as a *System.IO.Stream* object. For *OpenFileDialog* objects, it opens a read-only stream. For *SaveFileDialog* objects, it saves a new copy of the indicated file and then opens it as a read-write stream. You need to be careful when using the *SaveFileDialog.OpenFile* method to keep from overwriting preexisting files of the same name.
ShowDialog	Shows the dialog box modally, thereby halting application execution until the dialog box has been closed. Returns a *DialogResult*.

To use a file dialog box in a WPF application, follow these steps:

1. In Solution Explorer, right-click the project name and choose Add Reference.

   The Add Reference dialog box opens.

2. On the .NET tab, select *System.Windows.Forms,* and then click OK.

3. In code, create a new instance of the desired file dialog box, as shown here:

   Sample of Visual Basic Code

   ```
   Dim aDialog As New System.Windows.Forms.OpenFileDialog()
   ```

Sample of C# Code

```
System.Windows.Forms.OpenFileDialog aDialog =
   new System.Windows.Forms.OpenFileDialog();
```

4. Use the *ShowDialog* method to show the dialog box modally. After the dialog box is shown, you can retrieve the file name that was selected from the *FileNames* property. An example is shown here:

Sample of Visual Basic Code

```
Dim aResult As System.Windows.Forms.DialogResult
aResult = aDialog.ShowDialog()
If aResult = System.Windows.Forms.DialogResult.OK Then
   ' Shows the path to the selected file
   MessageBox.Show(aDialog.FileName)
End If

Sample of C# Code

System.Windows.Forms.DialogResult aResult;
aResult = aDialog.ShowDialog();
if (aResult == System.Windows.Forms.DialogResult.OK)
{
   // Shows the path to the selected file
   MessageBox.Show(aDialog.FileName);
}
```

NOTE

It is not advisable to import the *System.Windows.Forms* namespace because this leads to naming conflicts with several WPF classes.

WindowsFormsHost

While using dialog boxes in WPF applications is fairly straightforward, using controls is a bit more difficult. Fortunately, WPF provides an element specifically designed to ease this task, which is called *WindowsFormsHost*.

WindowsFormsHost is a WPF element that is capable of hosting a single child element that is a Windows Forms control. The hosted Windows Forms control automatically sizes itself to the size of the *WindowsFormsHost*. You can use the *WindowsFormsHost* to create instances of Windows Forms controls declaratively, and you also can set properties on hosted Windows Forms declaratively.

Adding a Windows Forms Control to a WPF Application

To use the *WindowsFormsHost* element in your WPF applications, first you must add to the XAML view a reference to the *System.Windows.Forms.Integration* namespace in the *WindowsFormsIntegration* assembly, as shown here. (This line has been formatted as two lines to fit on the printed page, but it should be on a single line in your XAML.)

```
xmlns:my="clr-namespace:System.Windows.Forms.Integration;
   assembly=WindowsFormsIntegration"
```

If you drag a *WindowsFormsHost* from the Toolbox to the designer, this reference is added automatically. You also must add a reference to the *System.Windows.Forms* namespace, as shown here:

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

Then you can create an instance of the desired Windows Forms control as a child element of a *WindowsFormsHost* element, as shown here:

```
<my:WindowsFormsHost Margin="48,106,30,56" Name="windowsFormsHost1">
   <wf:Button Text="Windows Forms Button" />
</my:WindowsFormsHost>
```

Setting Properties of Windows Forms Controls in a WPF Application

You can set properties on a hosted Windows Forms control declaratively in XAML as you would any WPF element, as shown in bold here:

```
<my:WindowsFormsHost Margin="48,106,30,56" Name="windowsFormsHost1">
   <wf:Button Text="Windows Forms Button" />
</my:WindowsFormsHost>
```

Although you can set properties declaratively on a hosted Windows Forms control, some of those properties will not have any meaning. For example, properties dealing with layout, such as *Anchor, Dock, Top,* and *Left,* have no effect on the position of the Windows Forms control. This is because its container is the *WindowsFormsHost* and the Windows Forms control occupies the entire interior of that element. To manage layout for a hosted Windows Forms control, you should set the layout properties of the *WindowsFormsHost,* as highlighted in bold here:

```
<my:WindowsFormsHost Margin="48,106,30,56" Name="windowsFormsHost1">
   <wf:Button Text="Windows Forms Button" />
</my:WindowsFormsHost>
```

Setting Event Handlers on Windows Forms Controls in a WPF Application

Similarly, you can set event handlers declaratively in XAML, as shown in bold in the following example:

```
<my:WindowsFormsHost Margin="48,106,30,56" Name="windowsFormsHost1">
   <wf:Button Click="Button_Click" Name="Button1" />
</my:WindowsFormsHost>
```

Note that events raised by Windows Forms controls are regular .NET events, not routed events, and therefore they must be handled at the source.

Obtaining a Reference to a Hosted Windows Forms Control in Code

In most cases, using simple declarative syntax with hosted Windows Forms controls is not sufficient—you have to use code to manipulate hosted Windows Forms controls. Although you can set the *Name* property of a hosted Windows Forms control, that name does not give you a code reference to the control. Instead, you must obtain a reference by using the *WindowsFormsHost.Child* property and casting it to the correct type. The following code example demonstrates how to obtain a reference to a hosted Windows Forms *Button* control:

Sample of Visual Basic Code

```
Dim aButton As System.Windows.Forms.Button
aButton = CType(windowsFormsHost1.Child, System.Windows.Forms.Button)
```

Sample of C# Code

```
System.Windows.Forms.Button aButton;
aButton = (System.Windows.Forms.Button)windowsFormsHost1.Child;
```

Choosing a Presentation Pattern

A presentation pattern separates the UI from its behavior and state. Compared to a traditional three-layer architecture, this results in separating the Presentation layer code from the Business Logic and Data code. Using a presentation pattern makes your code easier to maintain and test. Perhaps most important, it allows you to replace or extend the UI easily.

You certainly can develop a WPF or Windows Forms application without using a presentation pattern; that is exactly what Windows developers have been doing for most of the time Windows has existed. In fact, for a simple "Hello, world" application, using a presentation pattern would increase the complexity and decrease the code readability. When you create more complex enterprise applications that need to be tested and maintained, using a presentation pattern can reduce development costs greatly over time.

For .NET Framework applications, you can choose from three different presentation patterns:

- **Model-View-Controller (MVC)** Designed for web applications, this model uses views that consist of HTML and .NET Framework code to create the UI by formatting and displaying data from the model. The model stores the data and interacts with the database. The Controllers process requests and user input.
- **Model-View-ViewModel (M-V-VM)** Designed for WPF applications, this model closely resembles the MVC model. However, this model uses views that consist of XAML code to create the UI. The ViewModel is an abstraction of the view; it resides between the model and the view to present the model in a way that more closely resembles how the user will see it (thus reducing the amount of code required in the view itself). Views bind to the data they display by specifying the ViewModel as their *DataContext,* and views send data to the ViewModel using Commands that typically execute a *ViewModel* method.

- **Model-View-Presenter (MVP)** Designed for WPF applications, this model closely resembles both M-V-VM and MVC. The user interacts with the view, and the view raises events that the presenter responds to. The presenter interacts with the model and updates the values shown in the view.

Figure 2-1 visually compares the three architectures. In particular, notice that the user interacts solely with the controller in the MVC architecture, and then the view collects information from the controller about the user's request and retrieves data directly from the model. In the M-V-VM and MVP models, the user interacts directly with the view, and the ViewModel or presenter communicates with the model to prepare the data for the view.

Currently, MVC is the best choice for web applications. If you are creating a WPF application, you can choose either M-V-VM or MVP.

The key difference between M-V-VM and MVP is the way the ViewModel/presenter connects to the view. As illustrated by the directions of the arrows in Figure 2-1, this relationship is one-way for M-V-VM and two-way for MVP. Therefore, the M-V-VM ViewModel is loosely coupled with the view, whereas the MVP presenter is tightly coupled to the view.

To clarify, with the MVP model, the presenter needs a reference to the view because the presenter is responsible for manipulating the state of the view. With the M-V-VM model, the ViewModel is completely unaware that the view exists. With M-V-VM, the view sets a ViewModel as its *DataContext* and binds to properties on the ViewModel. Any changes to values in the ViewModel are reflected automatically on the view through that binding.

While the M-V-VM and MVP presentation patterns are very similar, M-V-VM is specialized to simplify WPF and Microsoft Silverlight development. Their structure is very similar, but there is simply better support for M-V-VM than for MVP. Therefore, M-V-VM is the best choice for WPF and Silverlight applications.

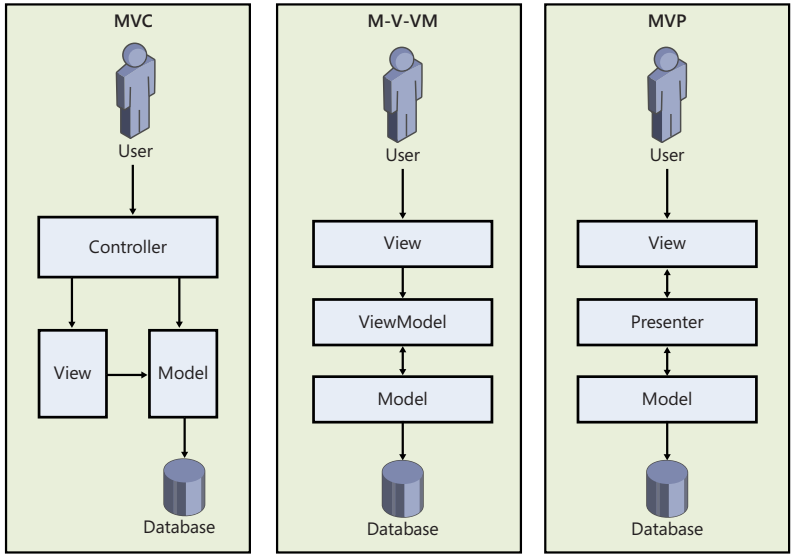


FIGURE 2-1 The three .NET Framework presentation patterns

MORE INFO **UNDERSTANDING PRESENTATION PATTERNS**

For more information, read "WPF Apps With The Model-View-ViewModel Design Pattern" at *http://msdn.microsoft.com/magazine/dd419663.aspx*.

Objective Summary

- Windows Forms is a well-known technology that has a host of developers for development projects and provides excellent support for globalization and localization. However, it is not as good for changing styles, and it has no inherent navigation capability.
- WPF is a relatively new technology with very powerful style and navigation features. However, because the technology is fairly new, it has not been adopted by as many developers, and support for globalization and localization is not as good as for Windows Forms.
- Interoperability between WPF and Windows Forms is possible through the *ElementHost* and *WindowsFormsHost* controls.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of the chapter.

1. What class allows you to host a WPF control in a Windows Forms application?
   - **A.** *WindowsFormsHost*
   - **B.** *ElementHost*
   - **C.** *Grid*
   - **D.** *Form*
2. You are designing a Presentation layer for an application. You want this application to be responsive to changes in the style of the desktop so that it blends seamlessly with the appearance of the desktop, and you need this application to collect a variety of specially formed data via the *MaskedTextBox* control. What is the best development strategy for your Presentation layer?
   - **A.** Build using Windows Forms.
   - **B.** Build using WPF.
   - **C.** Build using Windows Forms that incorporate WPF controls.
   - **D.** Build using WPF and incorporate Windows Forms controls.

THOUGHT EXPERIMENT
Designing the Presentation Layer

In the following thought experiment, design a Presentation layer for an application that will be distributed throughout the United States, Great Britain, and the English-speaking parts of Canada. This application is part of a large-scale psychology study. The application will stream videos in the UI and collect user input responses to the video stream in real time. Furthermore, processing behind the scenes in response to the user's real-time input changes the look and feel of the application in accordance with the user's perceived mood. The input collected from the user will be stored locally and uploaded to a data tier after the user has completed the test.

You can find answers to these questions in the "Answers" section at the end of this chapter.

1. **What technology, Windows Forms or WPF, should be used to implement this application and why?**
2. **How can we maintain responsiveness in the UI while performing processing in the background?**

Objective 2.2: Design the UI Layout and Structure

The layout of an application can mean the difference between an application that is easy to use and efficient and an application that is complicated and frustrates the user. Careful evaluation of the application design is vital to serving the needs of your users. In this section, you will learn how to make decisions that are important to the design of the layout.

This objective covers how to:

- Evaluate the conceptual design.
- Design for inheritance and the reuse of visual elements.
- Design for accessibility.
- Decide when custom controls are required.

Evaluate the Conceptual Design

The design of your UI should fulfill the needs of the application, the client, and the user. The interface should enable the user to complete tasks in the application quickly and easily and without distraction. A good UI will be internally and externally consistent and allow the user

to learn how to use the application intuitively. In their book *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design,* authors Larry Constantine and Lucy Lockwood describe the following principles for UI design:

- **Design for structure** A well-structured UI is logical, consistent, and intuitive. Use a model when designing your UI. Put elements with related functionality together in logically organized groups. Functional groups should be easily recognizable by the user, and dissimilar items or functional groups should be easily differentiated. A well-structured UI will not have users hunting blindly for desired functionality but will allow them to find what they need intuitively.
- **Design for simplicity** A well-designed UI is simple and easy to use. Common tasks are easy to access, and navigation to more infrequently used tasks is uncomplicated and intuitive. The use of menu items for common tasks is an example of incorporating simplicity into a UI.
- **Design for visibility** A good UI will have necessary functional units easily visible and accessible to the user. It is good to keep simplicity in mind with this principle, however, as an overly visible UI is likely to be cluttered, presenting the user with too many options and thwarting the intent of being easy to use.
- **Design for feedback** A good UI will keep the user informed as to changes in the state of the application. Errors that are relevant to the user or actions that need to be performed by the user should be communicated clearly and concisely. Conversely, the internal state of the application, if it does not require user input, should not be unnecessarily exposed.
- **Design for tolerance** Users make mistakes. A well-designed UI will allow the user to recover quickly and easily from mistakes. User interfaces should allow for faulty input to be easily and rapidly corrected and for mistaken changes to be rolled back.
- **Design for reuse** A UI should be consistent. Components and visual styles should be reused whenever possible. Not only does this reduce the amount of time spent coding but it also helps your UI to be consistent and purposeful.

Designing for Inheritance and the Reuse of Visual Elements

Both Windows Forms and WPF allow you to extend and reuse controls and forms in your application. Windows Forms allows extending existing controls and forms through visual inheritance, and WPF takes it further by allowing the visual appearance and behavior of every WPF element to be modified through the use of styles. WPF further allows the reuse of components designated as resources.

Creating Extended Controls in Windows Forms

Extended controls are user-created controls that extend a preexisting .NET Framework control. By extending existing controls, you can retain all the functionality of the control but add properties and methods and, in some cases, alter the rendered appearance of the control.

Extending a Control

You can create an extended control by creating a class that inherits the control in question. The following example demonstrates how to create a control that inherits the *Button* class:

Sample of Visual Basic.NET Code

```
Public Class ExtendedButton
   Inherits System.Windows.Forms.Button
End Class
```

Sample of C# Code

```
public class ExtendedButton : System.Windows.Forms.Button
{}
```

The *ExtendedButton* class created in the previous example has the same appearance, behavior, and properties as the *Button* class, but now you can extend this functionality by adding custom properties or methods. You also can override existing methods to incorporate custom functionality.

Extending a Form

To create a cohesive look and feel to your application, you might want to start with a base form that all other forms in the application derive from. You can extend preexisting forms in the same way that you extend controls. The following example shows how to inherit from a previously created form named BaseForm:

Sample of Visual Basic.NET Code

```
Public Class ExtendedForm
   Inherits MyProject.BaseForm
   ' Implementation omitted
End Class
```

Sample of C# Code

```
public class ExtendedForm : MyProject.BaseForm
{
   // Implementation omitted
}
```

Creating Custom Dialog Boxes

Dialog boxes are commonly used to gather information from the application user. Microsoft Visual Studio provides prebuilt dialog boxes that enable the user to select a file, font, or color. You can create custom dialog boxes to collect specialized information from the user. For example, you might create a dialog box that collects user information and relays it to the main form of the application.

A dialog box generally includes an OK button, a Cancel button, and whatever controls are required to gather information from the user. The general behavior of an OK button is to accept the information provided by the user and then to close the form, returning a result

of *DialogResult.OK*. The general behavior of the Cancel button is to reject the user input and close the form, returning a result of *DialogResult.Cancel*.

A *modal* dialog box is a dialog box that pauses program execution until the dialog box is closed. Conversely, a *modeless* dialog box allows application execution to continue. You can display any form as a modal dialog box by calling the *ShowDialog* method.

Using Styles in WPF

Styles can be thought of as analogous to cascading style sheets as used in HTML pages. Styles basically tell the Presentation layer to substitute a new visual appearance for the standard one. They allow you to make changes easily to the UI as a whole and to provide a consistent look and feel for your application in a variety of situations. Styles enable you to set properties and hook up events on UI elements through the application of those styles. Further, you can create visual elements that respond dynamically to property changes through the application of triggers, which listen for a property change and then apply style changes in response.

The primary class in the application of styles is, unsurprisingly, the *Style* class. The *Style* class contains information about styling a group of properties. A *Style* can be created to apply to a single instance of an element, to all instances of an element type, or across multiple types. The important properties of the *Style* class are shown in Table 2-4.

TABLE 2-4 Important Properties of the *Style* Class

Property	Description
BasedOn	Indicates another style that this style is based on. This property is useful for creating inherited styles.
Resources	Contains a collection of local resources used by the style.
Setters	Contains a collection of *Setter* or *EventSetter* objects. These are used to set properties or events on an element as part of a style.
TargetType	This property identifies the intended element type for the style.
Triggers	Contains a collection of *Trigger* objects and related objects that allow you to designate a change in the UI in response to changes in properties.

The basic skeleton of a <*Style*> element in XAML markup looks like the following:

```
<Style>
   <!-- A collection of setters is enumerated here -->
   <Style.Triggers>
   <!-- A collection of Trigger and related objects is enumerated here -->
   </Style.Triggers>
   <Style.Resources>
      <!-- A collection of local resources for use in the style -->
   </Style.Resources>
</Style>
```

SETTERS

The most common class you will use in the construction of styles is the *Setter.* As their name implies, *Setters* are responsible for setting some aspect of an element. *Setters* come in two flavors: property setters (or just *Setters*, as they are called in markup), which set values for properties; and event setters, which set handlers for events.

PROPERTY SETTERS

Property setters, represented by the <*Setter*> tag in XAML, allow you to set properties of elements to specific values. A property setter has two important properties: the *Property* property, which designates the property that is to be set by the *Setter,* and the *Value* property, which indicates the value to which the property is to be set. The following example demonstrates a *Setter* that sets the *Background* property of a *Button* element to *Red:*

```
<Setter Property="Button.Background" Value="Red" />
```

The value for *Property* must take the following form:

```
Element.PropertyName
```

If you want to create a style that sets a property on multiple different types of elements, you could set the style on a common class that the elements inherit, as shown here:

```
<Style>
   <Setter Property="Control.Background" Value="Red" />
</Style>
```

This style sets the *Background* property of all elements that inherit from the *Control* to which it is applied.

EVENT SETTERS

Event setters (represented by the <*EventSetter*> tag) are similar to property setters, but they set event handlers rather than property values. The two important properties for an *EventSetter* are the *Event* property, which specifies the event for which the handler is being set; and the *Handler* property, which specifies the event handler to attach to that event. An example is shown here:

```
<EventSetter Event="Button.MouseEnter" Handler="Button_MouseEnter" />
```

The value of the *Handler* property must specify an extant event handler with the correct signature for the type of event with which it is connected. Similar to property setters, the format for the *Event* property is <*Element*>.<*EventName*>, where the element type is specified, followed by the event name.

Using Resources in WPF

Logical resources allow you to define objects in XAML that are not part of the visual tree but are available for use by WPF elements in your UI. Elements in your UI can access the resource as needed. An example of an object that you might define as a resource is a *Brush* used to provide a common color scheme for the application.

By defining objects that are used by several elements in a *Resources* section, you gain a few advantages over defining the object each time you use it. First, you gain reusability because you define your object only once rather than multiple times. You also gain flexibility—by separating the objects used by your UI from the UI itself, you can refactor parts of the UI without having to redesign it completely. For example, you might use different collections of resources for different cultures in localization or for different application conditions.

Any type of object can be defined as a resource. Every WPF element defines a *Resources* collection, which can be used to define objects that are available to that element and the elements in its visual tree. Although it is most common to define resources in the *Resources* collection of the *Window,* you can define a resource in any element's *Resources* collection and access it, so long as the accessing element is part of the defining element's visual tree.

Declaring a Logical Resource

You declare a logical resource by adding it to a *Resources* collection, as seen here:

```
<Window.Resources>
   <RadialGradientBrush x:Key="myBrush">
      <GradientStop Color="CornflowerBlue" Offset="0" />
      <GradientStop Color="Crimson" Offset="1" />
   </RadialGradientBrush>
</Window.Resources>
```

If you don't intend a resource to be available to the entire *Window,* you can define it in the *Resources* collection of an element in the *Window,* as shown in this example:

```
<Grid>
   <Grid.Resources>
      <RadialGradientBrush x:Key="myBrush">
         <GradientStop Color="CornflowerBlue" Offset="0" />
         <GradientStop Color="Crimson" Offset="1" />
      </RadialGradientBrush>
   </Grid.Resources>
</Grid>
```

The usefulness of this is somewhat limited, and the most common scenario is to define resources in the *Window.Resources* collection. One point to remember is that when using static resources, you must define the resource in the XAML code before you refer to it. Static and dynamic resources are explained later in this section.

Every object declared as a *Resource* must set the *x:Key* property. This is the name that will be used by other WPF elements to access the resource. There is one exception to this rule: *Style* objects that set the *TargetType* property do not need to set the *x:Key* property explicitly because it is set implicitly, behind the scenes. In the previous two examples, the key is set to *myBrush*.

The *x:Key* property does not have to be unique in the application, but it must be unique in the *Resources* collection where it is defined. Thus, you could define one resource in the *Grid*.

Resources collection with a key of *myBrush* and another in the *Window.Resources* collection with the same key. Objects within the visual tree of the *Grid* that refer to a resource with the key *myBrush* refer to the object defined in the *Grid.Resources* collection, and objects that are not in the visual tree of the *Grid* but are within the visual tree of the *Window* refer to the object defined in the *Window.Resources* collection.

Application Resources

In addition to defining resources at the level of the element or *Window,* you can define resources that are accessible by all objects in a particular application. You can create an application resource by opening the App.xaml file (for C# projects) or the Application.xaml file (for Visual Basic projects) and adding the resource to the *Application .Resources* collection, as shown in bold here:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>
       <SolidColorBrush x:Key="appBrush" Color="PapayaWhip" />
    </Application.Resources>
</Application>
```

Accessing a Resource in XAML

You can access a resource in XAML by using the following syntax:

```
{StaticResource myBrush}
```

In this example, the markup declares that a static resource with the key *myBrush* is accessed. Because this resource is a *Brush* object, you can plug that markup into any place that expects a *Brush* object. This example demonstrates how to use a resource in the context of a WPF element:

```
<Grid Background="{StaticResource myBrush}">
</Grid>
```

When a resource is referred to in XAML, the *Resources* collection of the declaring object first is searched for a resource with a matching key. If one is not found, the *Resources* collection of that element's parent is searched, and so on, up to the *Window* that hosts the element and to the application *Resources* collection.

Static and Dynamic Resources

In addition to the syntax described previously, you can refer to a resource with the following syntax:

```
{DynamicResource myBrush}
```

The difference between the syntax of *DynamicResource* and the syntax of *StaticResource* lies in how the resources are retrieved by the referring elements. Resources referred to by the *StaticResource* syntax are retrieved once by the referring element and used for the lifetime of the resource. Resources referred to with the *DynamicResource* syntax, on the other hand, are acquired every time the referred object is used.

It might seem intuitive to think that if you use the *StaticResource* syntax, the referring object does not reflect changes to the underlying resource, but this is not necessarily the case. WPF objects that implement dependency properties automatically incorporate change notification, and changes made to the properties of the resource are picked up by any objects using that resource. Take the following example:

```
<Window x:Class="WpfApplication2.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
   <Window.Resources>
      <SolidColorBrush x:Key="BlueBrush" Color="Blue" />
   </Window.Resources>
   <Grid Background="{StaticResource BlueBrush}">
   </Grid>
</Window>
```

This code renders the *Grid* in the *Window* with a *Blue* background. If the color property of the *SolidColorBrush* object defined in the *Window.Resources* collection were changed in code to *Red,* for instance, the background of the *Grid* would render as *Red* because change notification would notify all objects using that resource that the property had changed.

The difference between static and dynamic resources comes when the underlying object changes. If the *Brush* defined in the *Windows.Resources* collection were accessed in code and set to a different object instance, the *Grid* in the previous example would not detect this change. However, if the *Grid* used the following markup, the change of the object would be detected and the *Grid* would render the background with the new *Brush*:

```
<Grid Background="{DynamicResource BlueBrush}">
</Grid>
```

Accessing resources in code is discussed in the not found "Retrieving Resources in Code" section later in this chapter.

The downside of using dynamic resources is that they tend to decrease application performance. Because the resources are retrieved every time they are used, dynamic resources can reduce the efficiency of an application. The best practice is to use static resources unless there is a specific reason for using a dynamic resource. Examples of when you would want to use a dynamic resource include when you use the *SystemBrushes, SystemFonts,* and *SystemParameters* classes as resources, or any other time when you expect the underlying object of the resource to change.

Creating a Resource Dictionary

A *resource dictionary* is a collection of resources that reside in a separate XAML file and can be imported into your application. They can be useful for organizing your resources in a single place or for sharing resources between multiple projects in a single solution. The following procedure describes how to create a new resource dictionary in your application:

To create a resource dictionary

1. From the Project menu, choose Add Resource Dictionary. The Add New Item dialog box opens. Choose the name for the resource dictionary and click Add. The new resource dictionary is opened in XAML view.
2. Add resources to the new resource dictionary in XAML view. You can add resources to the file in XAML view, as shown in bold here:

```
<ResourceDictionary
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <SolidColorBrush x:Key="appBrush" Color="DarkSalmon" />
</ResourceDictionary>
```

Merging Resource Dictionaries

For objects in your application to access resources in a resource dictionary, you must merge the resource dictionary file with a *Resources* collection that is accessible in your application, such as the *Windows.Resources* or *Application.Resources* collection. You merge resource dictionaries by adding a reference to your resource dictionary file in the *ResourceDictionary.MergedDictionaries* collection. The following example demonstrates how to merge the resources in a *Windows.Resources* collection with the resources in resource dictionary files named Dictionary1.xaml and Dictionary2.xaml:

```
<Window.Resources>
   <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
         <ResourceDictionary Source="Dictionary1.xaml" />
         <ResourceDictionary Source="Dictionary2.xaml" />
      </ResourceDictionary.MergedDictionaries>
      <SolidColorBrush x:Key="BlueBrush" Color="Blue" />
   </ResourceDictionary>
</Window.Resources>
```

Note that if you define additional resources in your *Resources* collection, they must be defined within the bounds of the *<ResourceDictionary>* tags.

Choosing Where to Store a Resource

You have seen several options regarding where resources should be stored. The factors that should be weighed when deciding where to store a resource include ease of accessibility by referring elements, readability and maintainability of the code, and reusability.

For resources to be accessed by all elements in an application, you should store resources in the *Application.Resources* collection. The *Window.Resources* collection makes resources available only to elements in that *Window,* but that is typically sufficient for most purposes. If you need to share individual resources over multiple projects in a solution, your best choice is to store your resources in a resource dictionary that can be shared among different projects.

Readability is important for enabling maintenance of your code by other developers. The best choice for readability is to store resources in the *Window.Resources* collection because this allows developers to read your code in a single file rather than having to refer to other code files.

If making your resources reusable is important, then the ideal method for storing them is to use a resource dictionary. This allows you to reuse resources among different projects and to extract those resources easily for use in other solutions as well.

Designing for Accessibility

The workforce contains a significant number of people with accessibility requirements, requiring applications that meet the broad demands of today's business environment. Accessible applications begin in the design phase. When you plan for accessibility in application design, you can integrate the principles of accessibility into the UI. Some of these principles are:

- Flexibility
- Choice of input and output methods
- Consistency
- Compatibility with accessibility aids

An accessible program requires flexibility. Users must be able to customize the UI to suit their specific needs—for example, by increasing font sizes. A user also should have a choice of input methods, such as keyboard and mouse devices. That is, the application should provide keyboard access for all important features and mouse access for all main features. A choice of output methods also renders an application more accessible, and the user should have the ability to choose among visual cues, sounds, text, and graphics. An accessible application should interact within its own operation and with other applications in a consistent manner, and it should be compatible with existing accessibility aids.

Support Standard System Settings

For your application to be accessible, it must support standard system settings for size, color, font, and input. Adopting this measure will ensure that all users' applications have a consistent UI that conforms to the system settings. Users with accessibility needs can thus configure all their applications by configuring their system settings.

You can implement standard system settings in your application by using the classes that represent the UI options used by the system. Table 2-5 lists the classes that expose the system settings. These classes are found in the *System.Drawing* namespace.

TABLE 2-5 Classes That Expose System Settings

Class	Description
SystemBrushes	Exposes *Brush* objects that can be used to paint in the system colors
SystemColors	Exposes the system colors
SystemFonts	Exposes the fonts used by the system
SystemIcons	Exposes the icons used by the system
SystemPens	Exposes *Pen* objects that can be used to draw in the system colors

These classes monitor changes to the system settings and adjust correspondingly. For example, if you build an application that uses the *SystemFonts* class to determine all the fonts, the fonts in the application will be reset automatically when the system settings are changed.

Ensure Compatibility with the High-Contrast Option

The high-contrast option (which users can set themselves in the Control Panel) sets the Windows color scheme to provide the highest possible level of contrast in the UI. This option is useful for users requiring a high degree of legibility.

By using only system colors and fonts, you can ensure that your application is compatible with the high-contrast settings. You also should avoid the use of background images because they tend to reduce contrast in an application.

Provide Documented Keyboard Access to All Features

Your application should provide keyboard access for all features and comprehensive documentation that describes this access. Shortcut keys for controls and menu items, as well as setting the *Tab* order for controls on the UI, allow you to implement keyboard navigation in your UI. Documentation of these features is likewise important. A user must have some means of discovering keyboard access to features, whether that is through UI cues or actual documentation.

Provide Notification of the Keyboard Focus Location

The location of the keyboard focus is used by accessibility aids such as *Magnifier* and *Narrator*. Thus, it is important that the application and the user have a clear understanding of where the keyboard focus is at all times. For most purposes the .NET Framework provides this functionality, but when designing your program flow, you should incorporate code to set the focus to the first control on a form when the form is initially displayed and the *Tab* order should follow the logical program flow.

Convey No Information by Sound Alone

Although sound is an important cue for many users, an application should never rely on conveying information by using sound alone. When using sound to convey information, you should combine that with a visual notification, such as flashing the form or displaying a message box.

Accessibility Properties of Windows Forms Controls

In addition to properties that affect the visual interface of a control, Windows Forms controls have five properties related to accessibility that determine how the control interacts with accessibility aids. These properties are summarized in Table 2-6.

TABLE 2-6 Accessibility Properties of Windows Controls

Property	Description
AccessibleDefaultActionDescription	Contains a description of the default action of a control. This property cannot be set at design time and must be set in code.
AccessibleDescription	Contains the description that is reported to accessibility aids.
AccessibleName	Contains the name that is reported to accessibility aids.
AccessibleRole	Contains the role that is reported to accessibility aids. This value is a member of the *AccessibleRole* enumeration, and a variety of accessibility aids use it to determine what kind of UI element an object is.
AccessibilityObject	Contains an instance of *AccessibleObject*, which provides information about the control to usability aids. This property is read-only and set by the designer.

These properties provide information to accessibility aids about the role of the control in the application. Accessibility aids then can present this information to the user or make decisions about how to display the control.

Deciding When Custom Controls Are Needed

User controls, custom controls, and templates all allow you to create custom elements with custom appearances. Because each of these methods is so powerful, you might be confused about what technique to use when creating a custom element for your application. The key to making the right decision isn't based on the appearance that you want to create, but rather the functionality that you want to incorporate into your application.

The standard WPF controls provide a great deal of built-in functionality. If the functionality of one of the preset controls, such as a progress bar or a slider, matches the functionality that you want to incorporate, then you should create a new template for that preexisting control to achieve your visual appearance goals. Creating a new template is the lightest solution to creating a custom element, and you should consider that option first.

If the functionality that you want to incorporate into your application can be achieved through a combination of preexisting controls and code, you should consider creating a user control. User controls allow you to bind together multiple preexisting controls in a single interface and add code that determines how they behave.

If no preexisting control or combination of controls can approach the functionality that you want to create, then you should create a custom control. Custom controls allow you to create a completely new template that defines the visual representation of the control and add completely custom code that determines the control's functionality.

Objective Summary

- The conceptual design should be evaluated for structure, simplicity, visibility, feedback, tolerance, and reuse.
- Whenever possible, code should be inherited and reused. You can extend controls and forms in Windows Forms applications and reuse styles and resources in WPF.
- When designing for accessibility, you should support standard system settings, ensure compatibility with the high-contrast option, provide documented keyboard access to all features, provide notification of the keyboard focus location, and convey no information by sound alone.
- User controls should be employed when you want to bind multiple preexisting controls into a single functional unit. Custom controls should be used when no preexisting control or controls incorporate the functionality you desire.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of the chapter.

1. Which of the following is not a best practice for designing for accessibility?
   - **A.** Provide audio for all important information.
   - **B.** Support standard system settings.
   - **C.** Ensure compatibility with the high-contrast mode.
   - **D.** Provide keyboard access to all important functionalities.
2. You have created a series of customized *Brush* objects that will be used to create a common color scheme for every window in each of several applications in your company. The *Brush* objects have been implemented as resources. What is the best place to define these resources?
   - **A.** In the *Resources* collection of each control that needs them
   - **B.** In the *Resources* collection of each *Window* that needs them
   - **C.** In the *Application.Resources* collection
   - **D.** In a separate resource dictionary

THOUGHT EXPERIMENT
Customizing the Appearance of an Application

In the following thought experiment, apply what you've learned about this objective to predict how a theoretical application architecture will perform. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are a consultant for Adventure Works. Adventure Works is designing the UI layout and structure for a new WPF application. Adventure Works wants the application to match the organization's style by using its colors and typography.

Adventure Works has created what they call an Appearance Standards list. This list includes a set of property values that developers must set for any UI elements and window backgrounds. These property values define the colors and shapes of those UI elements so that they are consistent with the organization's standards. To ensure that developers follow these standards, Adventure Works has updated its testing process to verify individual UI properties.

Answer the following questions about the performance of the application:

1. **Are their appearance standards the best way to maintain a common style?**
2. **Will the application be accessible to visually impaired users? If not, what would you do differently?**

Objective 2.3: Design Application Workflow

Some applications require no directed workflow or have very simplistic workflow requirements. Other applications can be very directional or even branching in the workflow design. This section describes how to implement user navigation in Windows Forms and WPF applications.

This objective covers how to:

- Implement user navigation.
- Create navigation applications in WPF.
- Handle navigation events.
- Use the *Hyperlink*, *NavigationService*, *PageFunction*, and *Journal* objects.
- Host pages in frames.

Implementing User Navigation

Simple client interfaces can require no more than a single form, but for more complex interfaces a navigable user experience is frequently required. Both Windows Forms and WPF allow you to incorporate user navigation into your applications.

MDI Forms in Windows Forms

MDI applications follow a parent form/child form model. An MDI application generally has a single parent form that contains and organizes multiple child forms (although it is possible for an application to have multiple parent forms). Microsoft Excel is an example of an MDI application—you can open multiple documents and work with them separately within the parent form. The parent form organizes and arranges all the child documents that are currently open.

Creating an MDI Parent Form

The parent form is the main form of any MDI application. This form contains all child forms that the user interacts with and handles the layout and organization of the child forms as well. It is a simple task to create an MDI parent form in Visual Studio.

To create an MDI parent form

1. Create a new Windows Forms application.
2. In the Properties window for the startup form, set the *IsMDIContainer* property to *True*. This designates the form as an MDI parent form.

Creating MDI Child Forms

MDI child forms are at the center of user interaction in MDI applications. They present the data to the user and generally contain individual documents. Child forms are contained within, and are managed by, a parent form. You can create an MDI child form by setting the *MdiParent* property of the form.

To create an MDI child form

1. Create an MDI parent form, as described previously.
2. In Visual Studio, add a second form to the project and add controls to implement the UI. This is the child form.
3. In a method in the parent form, such as a menu item *Click* event handler, create a new instance of the child form and set its *MdiParent* property, as shown in the following example:

Sample of Visual Basic.NET code

```
' This example takes place in a method in the parent form, and
' assumes a Form called ChildForm
Dim aChildForm As New ChildForm
```

```
' Sets the MdiParent property to the parent form
aChildForm.MdiParent = Me
aChildForm.Show
```

Sample of C# code

```
// This example takes place in a method in the parent form, and
// assumes a Form called ChildForm
ChildForm aChildForm = new ChildForm();
// Sets the MdiParent property to the parent form
aChildForm.MdiParent = this;
aChildForm.Show();
```

Identifying the Active Child Form

At times, you will want to identify the active child form in an MDI application. For example, a common feature of MDI applications is a central menu on the parent form that contains commands that act upon the child form that has the focus. You can use the *ActiveMDIChild* property of the parent form to obtain a reference to the form that was last accessed. The following code example demonstrates how to obtain a reference to the active child form:

Sample of Visual Basic.NET Code

```
' This example demonstrates how to obtain a reference to the active child
' form from a method inside the parent form
Dim aForm As Form
aForm = Me.ActiveMDIChild
```

Sample of C# Code

```
// This example demonstrates how to obtain a reference to the active child
// form from a method inside the parent form
Form aForm;
aForm = this.ActiveMDIChild;
```

Sending Data to the Active Child Form from the Clipboard

Once you have identified the active MDI form, you can use the properties of the form to send data from the Clipboard to an active control on the form. You might use this functionality to implement a *Paste* menu item to paste data from the Clipboard into a control. The following code example demonstrates how to determine if the active control is a text box and paste text from the Clipboard into the text box:

Sample of Visual Basic.NET Code

```
Dim activeForm As Form = Me.ActiveMDIChild
' Checks to see if an active form exists
If Not activeForm Is Nothing Then
   If activeForm.ActiveControl.GetType Is GetType(TextBox) Then
      Dim aTextBox As TextBox = CType(activeForm.ActiveControl, TextBox)
      ' Creates a new instance of the DataObject interface.
      Dim data As IDataObject = Clipboard.GetDataObject()
      ' Checks to see of the data in the data object is text. If it is,
      ' the text of the active Textbox is set to the text in the clipboard.
```

```
      If data.GetDataPresent(DataFormats.Text) Then
         aTextBox.Text = data.GetData(DataFormats.Text).ToString()
      End If
   End If
End If
```

Sample of C# Code

```
Form activeForm = this.ActiveMDIChild;
// Checks to see if an active form exists
if (activeForm != null)
{
    if (activeForm.ActiveControl.GetType() is TextBox)
    {
       TextBox aTextBox = (TextBox)activeForm.ActiveControl;
       // Creates a new instance of the DataObject interface.
       IDataObject data = Clipboard.GetDataObject();
       // Checks to see of the data in the data object is text. If it is,
       // the text of the active Textbox is set to the text in the clipboard.
       if (data.GetDataPresent(DataFormats.Text))
       {
          aTextBox.Text = data.GetData(DataFormats.Text).ToString();
       }
    }
 }
```

Arranging MDI Child Forms

You will commonly want to organize the forms in an MDI application so that they are ordered. The MDI parent form can arrange the child forms that it contains by calling the *LayoutMdi* method. The *LayoutMdi* method takes a parameter that is a member of the *MdiLayout* enumeration. This method causes the forms contained by the parent form to be arranged in the manner specified by the parameter. The members of the *MdiLayout* enumeration are described in Table 2-7.

TABLE 2-7 *MdiLayout* Enumeration Members

Member	Description
ArrangeIcons	All MDI child icons are arranged within the client region of the MDI parent form.
Cascade	All MDI child windows are cascaded within the client region of the MDI parent form.
TileHorizontal	All MDI child windows are tiled horizontally within the client region of the MDI parent form.
TileVertical	All MDI child windows are tiled vertically within the client region of the MDI parent form.

The following example demonstrates the *LayoutMdi* method by causing the contained forms to cascade in the parent form:

Sample of Visual Basic.NET Code

```
' Causes the contained forms to cascade in the parent form
Me.LayoutMdi(System.Windows.Forms.MdiLayout.Cascade)
```

Sample of C# Code

```
// Causes the contained forms to cascade in the parent form
this.LayoutMdi(System.Windows.Forms.MdiLayout.Cascade);
```

Navigation Applications in WPF

Unlike Windows applications, which are based on the WPF Window object and are toolbar-driven and menu-driven, page-based applications are based on the WPF Page object and are navigation-driven, meaning that the flow of the program is driven by navigating through multiple pages rather than interacting with existing windows by using menu and toolbar commands. Although the page-based model is limited in some ways, it lends itself very well to lightweight applications that are focused around a single task, such as a wizard or a shopping cart application. This section will focus primarily on the navigation of page-based applications.

Using Hyperlinks

The most familiar method of page-based navigation is by using hyperlinks. Hyperlinks are displayed as a section of text, usually underlined and in a different color than the surrounding text, which the user can click. When the user clicks a hyperlink, the application navigates to the page indicated by the hyperlink.

Hyperlinks expose a property called *NavigateUri* that indicates the target of the hyperlink. You set the *NavigateUri* property in XAML to indicate the navigation target when the hyperlink is clicked, as shown here:

```
<TextBlock>This is a <Hyperlink NavigateUri="Page2.xaml">hyperlink</Hyperlink>
</TextBlock>
```

Hyperlinks are not controls themselves—rather, they are inline flow elements. That means they must be placed within another element that supports inline flow elements, such as a *TextBlock* element. When a hyperlink pointing to another XAML page is clicked, a new instance of that page is created and the application navigates to that page. A hyperlink also can point to a *PageFunction*, but it is not possible to return a value from a *PageFunction* using a hyperlink. *PageFunction* objects are discussed in greater detail later in this section.

In addition to linking to other WPF pages, hyperlinks can link your application to webpages. You can link an application to a webpage by supplying the Hypertext Transfer Protocol (HTTP) address in the *NavigateUri* property, as shown here:

```
<TextBlock>This is a <Hyperlink NavigateUri="http://www.microsoft.com">hyperlink</
Hyperlink> </TextBlock>
```

You can set the *NavigateUri* property of a hyperlink dynamically in code. This allows you to change the navigation target of a hyperlink in response to program conditions. To change the *NavigateURI* property dynamically, you must set the *Name* property of the hyperlink in XAML, as shown here in bold:

```
<TextBlock>This is a <Hyperlink Name="myLink">hyperlink</Hyperlink></TextBlock>
```

Then you can set the *NavigateUri* property in code, as shown here:

Sample of Visual Basic.NET Code

```
myLink.NavigateUri = New System.Uri("Page2.xaml", System.UriKind.Relative)
```

Sample of C# Code

```
myLink.NavigateUri = new System.Uri("Page2.xaml", System.UriKind.Relative);
```

Using *NavigationService*

Hyperlinks provide a fairly easy way to navigate between pages, but for more complicated navigational models, the *NavigationService* class provides finer control.

You can obtain a reference to the *NavigationService* class by calling the static *GetNavigationService* method, as shown here:

Sample of Visual Basic.NET Code

```
Dim myNav As NavigationService
myNav = NavigationService.GetNavigationService(Me)
```

Sample of C# Code

```
NavigationService myNav; myNav = NavigationService.GetNavigationService(this);
```

The *NavigationService* exposes a method called *Navigate*, which causes the application to navigate to the specified page. The most common way to use the *Navigate* method is to provide an instance of a Uniform Resource Identifier (URI), as shown here:

Sample of Visual Basic.NET Code

```
myNav.Navigate(New System.Uri("Page2.xaml", UriKind.Relative))
```

Sample of C# Code

```
myNav.Navigate(new System.Uri("Page2.xaml", UriKind.Relative));
```

You also can create an instance of a new page in memory and navigate to it with the *Navigate* method, as shown here:

Sample of Visual Basic.NET Code

```
Dim aPage As New Page2()
myNav.Navigate(aPage)
```

Sample of C# Code

```
Page2 aPage = new Page2();
myNav.Navigate(aPage);
```

There are advantages and disadvantages to each method. When passing a URI to the *Navigate* method, the application's journal can maintain the page data without having to maintain the entire page object in memory. Thus, memory overhead is lower using this method. However, it is not possible to pass information between pages using a URI. You can pass information between pages by creating a custom constructor for your page and using it to pass information or by setting properties on the page prior to navigating to it.

You can use the *NavigationService* to refresh your page by calling *NavigationService.Refresh*. *NavigationService* also allows you to navigate forward and backward in the journal by calling *NavigationService.GoForward* and *NavigationService.GoBack*, respectively. These methods are demonstrated here:

Sample of Visual Basic.NET Code

```
myNav.Refresh()
myNav.GoForward()
myNav.GoBack()
```

Sample of C# Code

```
myNav.Refresh();
myNav.GoForward();
myNav.GoBack();
```

The *NavigationService* also exposes two Boolean properties called *CanGoBack* and *CanGoForward*, which you can query to determine if the application can navigate backward or forward. An example is shown here:

Sample of Visual Basic.NET Code

```
If myNav.CanGoBack = True Then
   myNav.GoBack()
End If
```

Sample of C# Code

```
if (myNav.CanGoBack)
   myNav.GoBack();
```

Navigation is asynchronous. Thus, you can cancel navigation before it is completed by calling *NavigationService.StopLoading*, as shown here:

Sample of Visual Basic.NET Code

```
myNav.StopLoading()
```

Sample of C# Code

```
myNav.StopLoading();
```

Hosting Pages in Frames

In addition to hosting a stand-alone Navigation application in a *NavigationWindow*, you also can host a page inside a control called a frame. A *frame* is simply a host for a XAML page or a webpage and is itself hosted inside a page or window. This allows you to incorporate navigation-based sections into a Windows application.

The *Source* property of the *Frame* control indicates the page to be loaded into the frame. The following code demonstrates how to set the source for a frame in XAML:

```
<Frame Margin="66,98,12,64" Name="frame1" Source="Page2.xaml"/>
```

Using the Journal

The journal is a bit of built-in technology in XBAPs and Navigation applications that keeps a list of the pages that have been visited and allows you to navigate this list. This will be familiar to anyone who uses Internet Explorer—the Back button navigates backward in the history to previously visited pages. The *NavigationService* allows you to manipulate the contents of the journal.

REMOVING ITEMS FROM THE JOURNAL

You might want to remove items from the journal. For example, suppose that your application has a complex configuration step that runs through several pages initially but is required only once. After configuration, you might want to remove these journal entries so that the user could navigate the regular pages without reloading the configuration pages. Removing items from the journal is fairly straightforward. *NavigationService* provides a method called *RemoveBackEntry*, which removes the last entry in the journal and returns an instance of *JournalEntry* that describes the instance that was removed. The following example demonstrates how to remove the last item from the journal:

Sample of Visual Basic.NET Code

```
myNav.RemoveBackEntry()
```

Sample of C# Code

```
myNav.RemoveBackEntry();
```

You can use the *CanGoBack* property to remove all the items in the journal, as shown here:

Sample of Visual Basic.NET Code

```
While myNav.CanGoback
   myNav.RemoveBackEntry()
End While
```

Sample of C# Code

```
while (myNav.CanGoBack)
{
   myNav.RemoveBackEntry();
}
```

ADDING ITEMS TO THE JOURNAL

Adding items to the journal is considerably less straightforward than removing them. In general, you want to add items to the journal only when you want to take a "snapshot" of the state of a single page and allow the user to navigate back to previous states. For example, if you were performing a complex configuration task with multiple steps on a single page, you could provide custom journal entries to allow the user to roll back changes before they were committed.

NavigationService provides a method called *AddBackEntry*, but it is more difficult to use than it appears and is considerably more complicated than *RemoveBackEntry*. It takes a single parameter, which is an instance of a class that derives from *CustomContentState*. This class, which you must implement, stores the state information for the page and reconstitutes the page state when the custom entry is navigated to. You also must implement the *IProvideCustomContentState* interface in the page for which you want to provide custom journal entries. Finally, you must add the custom journal entry manually at the point that you want to take the snapshot. The following is a high-level procedure that describes the general protocol for adding custom journal entries.

To add custom journal entries

1. Create a class that inherits *CustomContentState*. (You need a separate class for each page for which you want to add custom entries.) This class also must be marked with the *Serializable* attribute.
2. Add member variables and public properties to this class that hold the state of each control on the page that you want to constitute.
3. Add code to override the *JournalEntryName* property, which indicates the name that will be displayed in the journal. Often you might want to set this value in the constructor for this class, or use a method to determine an automatic name.
4. Override the *Replay* method. This method is called when the application navigates backward or forward in the journal and is used to reconstitute the page. Although there are a few different approaches here, the best method is to create a callback method that executes a method in the page that receives the class instance as a parameter, thereby allowing the page access to the stored data.
5. Create a constructor for this class. The constructor should set the value of all data about the state of the page that needs to be stored. It also should indicate the address of the callback method for the *Replay* method and any other parameters that you need for this instance (such as the *JournalEntryName*).
6. In the page for which a custom journal entry will be created, create a method that handles the callback from the *Replay* method. This method should use the information in the passed parameter to restore the state of the page.
7. Implement the *IProvideCustomContentState* in the page. This involves implementing the method *GetContentState*. *GetContentState* must return a *CustomContentState* object—you return an instance of your class in this method.
8. Add code that calls the *NavigationService.AddBackEntry* method at each point for which you want to create a custom journal entry. Each time you call this method, you must provide a new instance of your class to save the custom state.

Handling Navigation Events

Navigation in WPF applications occurs asynchronously. Thus, the *NavigationService.Navigate* method will return before navigation is complete. *NavigationService* exposes several events that allow your application to react at different points in the navigation process. You can handle these events to provide custom validation, to update navigation progress, or to add any other custom navigation functionality that is required. Table 2-8 summarizes the navigation events exposed by *NavigationService*. The events are listed in the order in which they occur.

TABLE 2-8 Navigation Events Exposed by *NavigationService*

Event	Description
Navigating	The *Navigating* event occurs just as navigation begins.
Navigated	The *Navigated* event occurs after navigation has been initiated but before the target page has been retrieved.
NavigationProgress	The *NavigationProgess* event is raised after every 1 kilobyte (KB) of data has been received from the new page.
LoadCompleted	The *LoadCompleted* event is raised after the page has finished loading but before any of the page events fire.
FragmentNavigation	The *FragmentNavigation* event occurs as the page is about to be scrolled to the target element. This event does not fire if you do not use a URI with a target element.
NavigationStopped	The *NavigationStopped* event fires when the *StopLoading* method is called. Note that this event is not fired if navigation is canceled in the *Navigating* event handler.
NavigationFailed	The *NavigationFailed* event is raised if the requested page cannot be located or downloaded.

Note that the *NavigationService* events fire whether navigation occurs through the *NavigationService* or through hyperlink clicks.

Because *NavigationService* events are regular .NET events, not routed events, you can create event handlers by creating methods with the correct signature and then attaching them to the event with the *AddHandler* operator (in Visual Basic) or the += operator (in C#), as shown here:

Sample of Visual Basic.NET Code

```
Public Sub HandleNavigated(ByVal sender As Object, _
   ByVal e As System.Windows.Navigation.NavigationEventArgs)
  ' Event Handling Code goes here
End Sub
Public Sub HookupEventHandler()
   ' Hookup the event handler
   AddHandler NavigationService.Navigated, AddressOf HandleNavigated
End Sub
```

Sample of C# Code

```
public void HandleNavigated(object sender,
  System.Windows.Navigation.NavigationEventArgse)
{
   // Event handling code goes here
}
public void HookupEventHandler()
{
   NavigationService.Navigated += HandleNavigated;
}
```

PASSING INFORMATION TO NAVIGATION EVENTS

The *NavigationService.Navigate* method exposes overloads that allow you to pass additional information that becomes available when navigation events are being handled. For example, you might pass time stamp information or an object that could be used to validate the page request. To pass additional information to the event handlers, simply call one of the overloads of *NavigationService.Navigate* that takes an additional object parameter, as shown here:

Sample of Visual Basic.NET Code

```
NavigationService.Navigate(New System.Uri("page2.xaml"), "user = Joe")
```

Sample of C# Code

```
NavigationService.Navigate(new System.Uri("page2.xaml"), "user = Joe");
```

The additional information will be available in the *Navigated*, *NavigationStopped*, and *LoadCompleted* events through the *e.ExtraData* property, as shown here:

Sample of Visual Basic.NET Code

```
Public Sub Navigate(ByVal sender As Object,
   ByVal e As _System.Windows.Navigation.NavigationEventArgs)
   If e.ExtraData.ToString = "user=Kilroy" Then
      Trace.WriteLine("Kilroy was here")
   End If
End Sub
```

Sample of C# Code

```
public void Navigate(object sender,
   System.Windows.Navigation.NavigationEventArgs e)
   {
     if (e.ExtraData.ToString() == "user=Kilroy")
     Trace.WriteLine("Kilroy was here");
   }
```

CANCELLING NAVIGATION

You can cancel navigation in the *Navigating* event handler by setting the *e.Cancel* property to *True*, as shown here:

Sample of Visual Basic.NET Code

```
Public Sub NavigatingHandler(ByVal sender As Object, _
   ByVal e As System.Windows.Navigation.NavigatingCancelEventArgs)
   e.Cancel = True
End Sub
```

Sample of C# Code

```
public void NavigatingHandler(object sender,
   System.Windows.Navigation.NavigatingCancelEventArgs e)
   {
     e.Cancel = true;
   }
```

Using *PageFunction* Objects

The *PageFunction* class is very similar to the *Page* class. You can design a *PageFunction* in the designer, you can add controls to a *PageFunction*, and you can navigate to a *PageFunction* through hyperlinks or by using *NavigationService*. The principal difference between *Page* objects and *PageFunction* objects is that *PageFunction* objects can return a value. This allows you to create pages that act in an analogous manner to dialog boxes—they can collect user information and then return that information to the main page.

To add a *PageFunction* object to a project

1. From the Project menu, choose Add New Item to open the Add New Item dialog box.
2. In the Add New Item dialog box, select PageFunction (WPF). Name your *PageFunction* and click Add.

   A *PageFunction* can return any type of .NET object. When you add a *PageFunction* to your project, Visual Studio automatically configures it to return a *String* instance. Although this is frequently useful, you might want to return some other kind of object from a *PageFunction*, such as an integer or an object. Changing the return type of your *PageFunction* is relatively straightforward.

To change the return type of your *PageFunction*

1. In XAML view, locate the line in the *PageFunction* XAML that reads:

   ```
   x:TypeArguments="sys:String"
   ```

   Then change the *TypeArguments* parameter to the type you want, as follows:

   ```
   x:TypeArguments="sys:Object"
   ```

   For Visual Basic, that is all you need to do. For C#, the following additional step is required.
2. In Code view, locate the class declaration and change the type, as shown here:

   ```
   public partial class PageFunction1 : PageFunction<Object>
   ```

   When you are ready for your *PageFunction* to return a value, you should call the *OnReturn* method. The *OnReturn* method takes a parameter of the type specified for the *PageFunction*. You also can return *null* for this parameter if no return value is required. The page that navigated to the *PageFunction* should handle the *Returned* event for that *PageFunction*. The instance of *ReturnEventArgs* returned by that event contains the returned value.

To return a value from a *PageFunction*

1. In the page that navigates to the *PageFunction*, create a method that handles the *Returned* method of that *PageFunction*. An example is shown here:

   Sample of Visual Basic.NET Code

   ```
   Public Sub ReturnHandler(ByVal sender As Object, _
      ByVal e As ReturnEventArgs(Of String))
      myString = e.Result
   End Sub
   ```

   Sample of C# Code

   ```
   public void ReturnHandler(object sender,
      ReturnEventArgs<string> e)
   {
      myString = e.Result;
   }
   ```

2. In the page that navigates to the *PageFunction*, instantiate the *PageFunction* programmatically and add code to hook up the *PageFunction.Returned* event to the new event handler, as shown here:

   Sample of Visual Basic.NET Code

   ```
   Dim myPage As New PageFunction1
   AddHandler myPage.Return, AddressOf ReturnHandler
   ```

   Sample of C# Code

   ```
   PageFunction1 myPage = new PageFunction1();
   myPage.Return += ReturnHandler;
   ```

3. In the *PageFunction*, after the task is completed, call the *OnReturn* method and pass the return value in a new instance of *ReturnEventArgs*, as shown here:

   Sample of Visual Basic.NET Code

   ```
   OnReturn(New ReturnEventArgs(Of String)("Kilroy was here"))
   ```

   Sample of C# Code

   ```
   OnReturn(new ReturnEventArgs<String>("Kilroy was here"));
   ```

Removing *PageFunction* Entries from the Journal

Because *PageFunction* objects are used frequently to collect user input, you might not want to allow the user to return to a *PageFunction* via the journal after the task is completed. The *PageFunction* class exposes a property called *RemoveFromJournal*. When *RemoveFromJournal* is set to *True*, *PageFunction* entries are deleted automatically from the journal once the user is finished with the task.

Simple Navigation and Structured Navigation

Simple navigation is a common design model in lightweight page-based applications. An application with simple navigation has a start, an end, and a series of pages though which the user navigates. There is generally little or no branching, and after a page is visited, it generally

is not returned to unless the user wants to back up. Although this paradigm is well suited to certain types of applications, such as a configuration wizard, other kinds of applications might find it lacking. Consider a shopping cart application. A user might want to add items to a shopping cart, return to shop for more items, add them to the shopping cart, repeat this a few more times, and then check out. Strictly linear navigation would be insufficient in this case.

PageFunction objects allow you to build more structure into your application. With *PageFunction* objects, you can allow your users to leave the main line of execution to perform tasks and then return. Using *PageFunction* objects, you can create execution models with complex flow structures, and by manipulating the journal, you can control how a user is able to navigate back through the application.

Designing for Different Input Types

Different input types require different UI designs. When designing an application for deployment in a public kiosk, keep these factors in mind:

- Design the application to be full-screen, and remove control buttons that might allow a user to move, resize, or minimize a window.
- Users will speak different languages, so rely more on icons and images than text.
- Users will have varying accessibility requirements. Keep buttons and text large, simple, and high-contrast.
- Users will not spend time learning how to use your application. Defaults should be carefully selected, the UI must be as simple as possible, and you should limit the number of choices presented to the user at a time to less than four.
- Even with a touch-screen interface, avoid complex interactions such as dragging, pinching, or using multiple fingers. The only user interaction should be touching the screen or pressing a button.
- Perform extensive usability testing. For more information about testing, refer to Objective 5.2.

When designing a UI for a mobile device, keep these factors in mind:

- Build a task-based UI focused on the most common tasks. Use buttons instead of menus.
- Minimize the amount of typing required. For example, instead of prompting the user to type their state, provide them with a list they can select from.
- Avoid requiring users to scroll. Instead, separate the UI into more pages.
- Plan for users to be interrupted regularly. Mobile application use tends to be interrupted more often.
- Design the UI behavior to be as similar as possible to the operating system's default behavior. For example, support swiping, panning, and pinch-to-zoom (but do not require the user to take advantage of those capabilities).
- Support both horizontal and vertical screen orientations.

Objective Summary

- User navigation can be implemented in Windows Forms application through the use of MDI forms. MDI forms allow multiple forms to be arranged and navigated as part of a single application.
- Navigation applications in WPF provide an easy-to-program navigational experience that allows forward and backward navigation as well as branching. The *NavigationService* object provides the basic functionality required for user navigation.
- The *Journal* object allows you to keep a record of past states of a navigation application in case changes need to be rolled back.
- *PageFunction* objects behave like pages, but they return a value and can be useful for receiving user input.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of the chapter.

1. Which of the following is NOT required to implement custom back entries?
   - **A.** A *Page* or *PageFunction* that implements *IProvideCustomContentState*
   - **B.** Code that calls *NavigationService.AddBackEntry*
   - **C.** An instance of the *JournalEntry* class
   - **D.** A class that inherits from *CustomContentState*
2. Which of the following is the correct firing order for navigation events in a navigation application?
   - **A.** *Navigating*
     NavigationProgress
     Navigated
     FragmentNavigation
     LoadCompleted
   - **B.** *Navigating*
     Navigated
     NavigationProgress
     LoadCompleted
     FragmentNavigation
   - **C.** *Navigating*
     NavigationProgress

Navigated

LoadComplete

FragmentNavigation

D. *Navigating*

Navigated

NavigationProgress

FragmentNavigation

LoadCompleted

THOUGHT EXPERIMENT
Designing a Data Entry Wizard

In the following thought experiment, apply what you've learned about this objective to predict how a theoretical application architecture will perform. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are a consultant for Lucerne Publishing. Lucerne is creating a new WPF application that its data entry team can use to enter newly published books into its database. Because the data entry team is often staffed with temporary personnel, the UI must be intuitive and must minimize the opportunity for making mistakes.

The process of entering information about a new book requires the data entry staff member to enter about 40 different pieces of information. The developers have designed a UI with these attributes:

- **A WPF page-based application.**
- **A series of eight pages with five fields to enter different pieces of information on each.**
- **Next and Cancel buttons in the lower-right corner of the window.**
- **A Back button in the upper-left corner of the window.**

Answer the following questions about the future performance of the application:

1. **Would it be better to design the application using MDI forms? Why or why not?**
2. **The application creates a new record for the book after page 4. This process cannot be changed easily. Is it possible to prevent users from going back to earlier pages after they finish page 4? If so, how?**
3. **We would like to allow the user to click the Back button to undo changes within a single page. Is this possible? If so, how?**

Objective 2.4: Design Data Presentation and Input

Most of what your Presentation layer is doing in a distributed application is either presenting or receiving data. Your Presentation layer will be responsible for validating data, binding data and presenting it to the user, and presenting media to the user. In this section, you will learn how to handle data in the Presentation layer.

This objective covers how to:

- Design data validation.
- Design a data binding strategy.
- Manage data shared between forms.
- Choose media services.

Designing Data Validation

An important part of designing any UI is how to handle data input. Users can be error-prone, and to ensure the integrity of your database, it is vital that data be validated for correctness. You might need to use several different data validation techniques, depending on the needs of your application.

Data Type Validation

The simplest form of data validation is data type validation. In this type of validation, data is simply checked to ensure that it is of the appropriate type. For example, input that should be a string is checked to see if it is a string, and numeric values are parsed to an integer or decimal. This type of validation can usually be accomplished with fairly simple methods in the UI.

Range Checking

An extension of data type validation, range checking ensures not only that data is of the appropriate data type, but also that it falls within an acceptable range of values. For example, a field that asked for the age of an employee might require a value between 18 and 100, or some other range that represented the actual range of working employees within the company. This type of validation is also not complex and usually can be accomplished within the UI.

Lookup Validation

Lookup validation can be thought of as a more specialized form of range checking. In lookup validation, fields are only allowed to be one of a certain set of values, which may or may not be sequential in any way. For example, consider an application that required the serial

number of an item being sold to correspond to a serial number of an item in an inventory database. The typical solution to lookup validation is to create a lookup table and validate against the values contained in that table. Lookup validation can take place against a static set of values or against values that change over time, requiring dynamic generation of the lookup table. In either case, this kind of data validation usually requires a more complex business rule to validate against.

Complex Validation

Your input data might require a more complex type of validation than any of the types described here. For complex validation, separate business rules will be required.

Validating Data at the Client and Server

When validating user input, you should validate it on the client (for immediate responsiveness and to assist data entry) and again at the server (for security). Never trust data validation performed at the client because it is possible for users to bypass client-side validation controls by modifying the application, altering data after it is sent by the application, or creating an entirely different application that connects to the server.

The easiest way to validate data in a Windows Form application is to use the *Validating* event. The *Validating* event occurs before a control loses the focus. This event is raised only when the *CausesValidation* property of the control that is about to receive the focus is set to *True*. Thus, if you want to use the *Validating* event to validate data entered in your control, the *CausesValidation* of the next control in the tab order should be set to true. In order to use *Validating* events, the *CausesValidation* property of the control to be validated must also be set to *True*. By default, the *CausesValidation* property of all controls is set to *True* when controls are created at design time. Controls such as Help buttons are typically the only kind of controls that have *CausesValidation* set to *False*.

The *Validating* event allows you to perform sophisticated validation on your controls. You could, for example, implement an event handler that tested whether the value entered corresponded to a very specific format. Another possible use is an event handler that doesn't allow the focus to leave the control until a value has been entered.

The *Validating* event includes an instance of the *CancelEventArgs* class. This class contains a single property, *Cancel*. If the input in your control does not fall within the required parameters, you can use the *Cancel* property within your event handler to cancel the *Validating* event and return the focus to the control.

The *Validated* event fires after a control has been validated successfully. You can use this event to perform any actions based upon the validated input.

The following example demonstrates a handler for the *Validating* event. This method requires an entry in *TextBox1* before it will allow the focus to move to the next control.

Sample of Visual Basic.NET Code

```
Private Sub TextBox1_Validating(ByVal sender As Object, ByVal e As _
   System.ComponentModel.CancelEventArgs) Handles TextBox1.Validating
```

```
   ' Checks the value of TextBox1
   If TextBox1.Text = "" Then
      ' Resets the focus if there is no entry in TextBox1
      e.Cancel = True
   End If
End Sub
```

Sample of C# Code

```
private void textBox1_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Checks the value of textBox1
   if (textBox1.Text == "")
      // Resets the focus if there is no entry in TextBox1
      e.Cancel = true;
}
```

Validating Data in a WPF Application

WPF allows you to set validation rules that define how your application validates its data. Each *Binding* object exposes a *ValidationRules* collection. You can add new rules to the *ValidationCollection,* as shown in bold in this example:

```
<TextBox>
   <TextBox.Text>
      <Binding Path="CandyBars">
         <Binding.ValidationRules>
            <local:CandyBarValidationRule />
            <local:SweetTreatsValidationRule />
         </Binding.ValidationRules>
      </Binding>
   </TextBox.Text>
</TextBox>
```

In this example, the *CandyBarValidationRule* and *SweetTreatsValidationRule* declarations represent two custom validation rules that have been defined in your application. When a new value is bound, each of the validation rules are evaluated in the order in which they are declared. In this example, the *CandyBarValidationRule* is evaluated first, followed by the *SweetTreatsValidationRule*. If there are no validation problems, the application proceeds normally. If there is a problem that violates a validation rule, however, the following things happen:

- The element with the validation error is outlined in red.
- The attached property *Validation.HasError* is set to *True*.
- A new *ValidationError* object is created and added to the attached *Validation.Errors* collection.
- If the *Binding.NotifyOnValidationError* property is set to *True,* the *Validation.Error*–attached event is raised.
- The data-binding source is not updated with the invalid value and instead remains unchanged.

Implementing Custom Validation Rules

You can create specific validation rules by creating classes that inherit the abstract class *ValidationRule*. The *ValidationRule* class has one virtual method that must be overridden: the *Validate* method. The *Validate* method receives an object parameter, which represents the value that is being evaluated, and returns a *ValidationResult* object, which contains an *IsValid* property and an *ErrorCondition* property. The *IsValid* property represents a Boolean value that indicates whether or not the value is valid, and the *ErrorCondition* property is text that can be set to provide a descriptive error condition. If a *ValidationResult* with an *IsValid* value of *True* is returned, the value is considered to be valid and application execution proceeds normally. If a *ValidationResult* with an *IsValid* result of *False* is returned, a *ValidationError* is created as described previously.

The following example demonstrates a simple implementation of the *ValidationRule* abstract class:

Sample of Visual Basic.NET Code

```
Public Class NoNullStringsValidator
    Inherits ValidationRule

    Public Overrides Function Validate(ByVal value As Object, ByVal _
        cultureInfo As System.Globalization.CultureInfo) As _
        System.Windows.Controls.ValidationResult
        Dim astring As String = value.ToString
        If astring = "" Then
            Return New ValidationResult(False, "String cannot be empty")
        Else
            Return New ValidationResult(True, Nothing)
        End If
    End Function
End Class
```

Sample of C# Code

```
public class NoNullStringsValidator : ValidationRule
{
   public override ValidationResult Validate(object value,
      System.Globalization.CultureInfo cultureinfo)
   {
      string aString = value.ToString();
      if (aString == "")
         return new ValidationResult(false, "String cannot be empty");
      return new ValidationResult(true, null);
   }
}
```

In this example, the string contained in the *value* object is evaluated. If it is a zero-length string, the validation fails; otherwise, the validation succeeds.

Handling Validation Errors

Once validation errors are raised, you must decide how to respond to them. In some cases, the visual cues provided by the validation error are enough—the user can see that the element is surrounded by a red outline and can detect and fix the problem. In other cases,

however, you might need to provide feedback to the user regarding the nature of the validation problem.

When validation is enabled for a binding, the *Validation.Error* event is attached to the bound element. *Validation.Error* includes an instance of *ValidationErrorEventArgs*, which contains two important properties, as described in Table 2-9.

TABLE 2-9 Important Properties of *ValidationErrorEventArgs*

Property	Description
Action	Describes whether the error in question is a new error or an old error that is being cleared
Error	Contains information about the error that occurred, the details of which are described in further detail in Table 5-7

The *Error* object of *ValidationErrorEventArgs* contains a host of useful information regarding the error that occurred. Important properties of the *Error* object are described in Table 2-10.

TABLE 2-10 Important Properties of the *Error* Object

Property	Description
BindingInError	Contains a reference to the *Binding* object that caused the validation error
ErrorContent	Contains the string set by the *ValidationRule* object that returned the validation error
Exception	Contains a reference to the *Exception*, if any, that caused the validation error
RuleInError	Contains a reference to the *ValidationRule* that caused the validation error

The *Validation.Error* event is not fired unless the *NotifyOnValidationError* property of the *Binding* object is specifically set to *True*, as shown in bold here:

```
<Binding NotifyOnValidationError="True" Mode="TwoWay"
   Source="{StaticResource StringCollection}" Path="name">
   <Binding.ValidationRules>
      <local:NoNullStringsValidator/>
   </Binding.ValidationRules>
</Binding>
```

When this property is set to *True*, the *Validation.Error* event is raised anytime any *ValidationRule* in the *ValidationRules* collection of the *Binding* object detects a validation error. The *Validation.Error* event is a bubbling event. It is raised first in the element where the validation error occurs and then in each higher-level element in the visual tree. Thus, you can

create a local error-handling method that specifically handles validation errors from a single element, as shown in bold here:

```
<TextBox Validation.Error="TextBox1_Error" Height="21" Width="100"
    Name="TextBox1" >
```

Alternatively, you can create an error-handling routine that is executed higher in the visual tree to create a more generalized validation error handler, as shown in bold here:

```
<Grid Validation.Error="Grid_Error">
```

The *Validation.Error* event is fired both when a new validation error is detected and when an old validation error is cleared. Thus, it is important to check the *e.Action* property to determine whether the error is being cleared or it is a new error. The following example demonstrates a sample validation error handler that displays the error message to the user when a new error occurs and writes information to *Trace* when a validation error is cleared:

Sample of Visual Basic.NET Code

```
Private Sub Grid_Error(ByVal sender As System.Object, ByVal e As _
   System.Windows.Controls.ValidationErrorEventArgs)
   If e.Action = ValidationErrorEventAction.Added Then
      MessageBox.Show(e.Error.ErrorContent.ToString)
   Else
      Trace.WriteLine("Validation error cleared")
   End If
End Sub
```

Sample of C# Code

```
private void Grid_Error(object sender, ValidationErrorEventArgs e)
{
   if (e.Action == ValidationErrorEventAction.Added)
      MessageBox.Show(e.Error.ErrorContent.ToString());
   else
      System.Diagnostics.Trace.WriteLine("Validation error cleared");
}
```

Design a Data Binding Strategy

Data binding in the Presentation layer typically refers to binding presentation controls to a cached client-side copy of the bound data. This data can be presented to the user and changed or added to as need be, and then updated to or from the central data store when the application requires it. When deciding on a data binding strategy for your Presentation layer, you must decide how you want to store your local data, and then determine what component in the Presentation layer you want to use to connect the presentation logic to the local data.

Data Binding in Windows Forms

Local data in a Windows Forms application is typically held in a *Dataset* object. *Dataset* is a very versatile class that can handle multiple data tables. While you can bind your Presentation layer directly to a *Dataset,* a better strategy is to access the individual tables in a *Dataset* via a *BindingSource* component.

The *BindingSource* component manages data currency and navigation for the underlying data source. Thus, by binding your Presentation layer controls to a *BindingSource* object that refers to the local copy of data, you are able to manage the presentation of the underlying data easily. The *BindingSource* component contains the information that controls need to bind to a *BindingSource* by passing it a reference to a *DataTable* in a *DataSet*. By binding to the *BindingSource* instead of to the *DataSet*, you can redirect your application to another source of data easily without having to redirect all the data binding code to point to the new data source.

The following code shows how to create a *BindingSource* and assign it a reference to the Northwind Customers table in a Northwind database-based dataset named NorthwindDataSet1:

Sample of Visual Basic.NET Code

```
customersBindingSource = New BindingSource(NorthwindDataSet1, "Customers")
```

Sample of C# Code

```
customersBindingSource = new BindingSource(northwindDataSet1, "Customers");
```

Binding to Types Other Than *Dataset*

The *BindingSource* component can be used to bind to several different types of objects and will in most cases expose the underlying data of that object as an *IBindingList* interface. Table 2-11 explains the types that the *BindingSource.DataSource* property can be set to and what the result will be.

TABLE 2-11 Types for the *DataSource* Property of *BindingSource*

***DataSource* Property**	**List Results**
Nothing	An empty *IBindingList* of objects. Adding an item sets the list to the type of the added item.
Nothing with *DataMember* set	Not supported; raises *ArgumentException*.
Non-list type or object of type "T"	Empty *IBindingList* of type "T".
Array instance	*IBindingList* containing the array elements.
IEnumerable instance	An *IBindingList* containing the *IEnumerable* items.
List instance containing type "T"	*IBindingList* instance containing type "T".

Data Binding in WPF

WPF has data binding built in at all levels, and you can bind a WPF Presentation layer easily to a variety of data sources, including datasets, objects, or XML representations in memory.

Binding to collections in WPF is handled in essentially the same way, whether the bound collection is a dataset, datatable, or other in-memory object. For simple displaying of bound

members, you must set the *ItemsSource* property to the collection to which you are binding and set the *DisplayMemberPath* property to the collection member that is to be displayed. The following example demonstrates how to bind a *ListBox* to a static resource named *myList* and a display member called *FirstName:*

```
<ListBox Width="200" ItemsSource="{Binding Source={StaticResource myList}}"
   DisplayMemberPath="FirstName" />
```

A more common scenario when working with bound lists, however, is to bind to an object that is defined and filled with data in code. In this case the best way to bind to the list is to set the *DisplayMemberPath* in the XAML and then set the *DataContext* of the element or its container in code. The following example demonstrates how to bind a *ListBox* to an object called *myCustomers* that already has been created and populated. The *ListBox* displays the entries from the *CustomerName* property:

Sample of Visual Basic.NET Code

```
' Code to initialize and fill myCustomers has been omitted
grid1.DataContext = myCustomers;
```

Sample of Visual C# Code

```
// Code to initialize and fill myCustomers has been omitted
grid1.DataContext = myCustomers;
```

Sample of XAML Code

```
<Grid Name="grid1">
   <ListBox ItemsSource="{Binding}" DisplayMemberPath="CustomerName"
      Margin="92,109,66,53" Name="ListBox1" />
</Grid>
```

Note that in the XAML for this example, the *ItemsSource* property is set to a *Binding* object that has no properties initialized. The *Binding* object binds the *ItemsSource* of the *ListBox*, but because the *Source* property of the *Binding* is not set, WPF searches upward through the visual tree until it finds a *DataContext* that has been set. Because the *DataContext* for *grid1* has been set in code to *myCustomers*, this then becomes the source for the binding.

Navigating Bound Data in WPF

WPF has a built-in navigation mechanism for data and collections. When a collection is bound to by a WPF *Binding,* an *ICollectionView* is created behind the scenes. The *ICollectionView* interface contains members that manage data currency, as well as managing views, grouping, and sorting. You can get a reference to the *ICollectionView* by calling the *CollectionViewSource.GetDefaultView* method, as shown here:

Sample of Visual Basic.NET Code

```
' This example assumes a collection named myCollection
Dim myView As System.ComponentModel.ICollectionView
myView = CollectionViewSource.GetDefaultView (myCollection)
```

Sample of Visual C# Code

```
// This example assumes a collection named myCollection
System.ComponentModel.ICollectionView myView;
myView = CollectionViewSource.GetDefaultView (myCollection);
```

When calling this method, you must specify the collection or list for which to retrieve the view (which is *myCollection* in the previous example). *CollectionViewSource.GetDefaultView* returns an *ICollectionView* object that is actually one of three different classes depending on the class of the source collection.

If the source collection implements *IBindingList,* the view returned is a *BindingListCollectionView* object. If the source collection implements *IList* but not *IBindingList,* the view returned is a *ListCollectionView* object. If the source collection implements *IEnumerable* but not *IList* or *IBindingList,* the view returned is a *CollectionView* object.

Binding to XML in WPF

The *XmlDataProvider* allows you to bind WPF elements to data in the XML format. The following example demonstrates an *XmlDataProvider* providing data from a source file called Items.xml:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items" Source="Items.xml" />
</Window.Resources>
```

You can also provide the XML data inline as an XML data island. In this case you wrap the XML data in *XData* tags, as shown here:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items">
      <x:XData>
         <!--XML Data omitted-->
      </x:XData>
   </XmlDataProvider>
</Window.Resources>
```

You can bind elements to the data provided by an *XmlDataProvider* in the same way that you would bind to any other data source—namely, using a *Binding* object and specifying the *XmlDataProvider* in the *Source* property, as shown here:

```
<ListBox ItemsSource="{Binding Source={StaticResource Items}}"
   DisplayMemberPath="ItemName" Name="listBox1" Width="100" Height="100"
   VerticalAlignment="Top" />
```

Using XPath When Binding to XML

You can use *XPath* expressions to filter the results exposed by the *XmlDataProvider* or to filter the records displayed in the bound controls. By setting the *XPath* property of the *XmlDataProvider* to an *XPath* expression, you can filter the data provided by the source. The following example filters the results exposed by an *XmlDataProvider* object to include only those nodes called <*ExpensiveItems*> in the <*Items*> top-level node:

```
<Window.Resources>
   <XmlDataProvider x:Key="Items" Source="Items.xml"
   XPath="Items/ExpensiveItems" />
</Window.Resources>
```

You also can apply *XPath* expressions in the bound controls. The following example sets the *XPath* property to Diamond (shown in bold), which indicates that only data contained in *<Diamond>* tags will be bound:

```
<ListBox ItemsSource="{Binding Source={StaticResource Items}
   XPath=Diamond}" DisplayMemberPath="ItemName" Name="listBox1" Width="100"
   Height="100" VerticalAlignment="Top" />
```

Using Data Templates in the WPF Presentation Layer

A *data template* is a bit of XAML that describes how bound data is displayed. A data template can contain elements that are bound to a data property, along with additional markup that describes layout, color, and other aspects of appearance. The following example demonstrates a simple data template that describes a *Label* element bound to the *ContactName* property. The *Foreground, Background, BorderBrush,* and *BorderThickness* properties are also set:

```
<DataTemplate>
   <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
      Background="Yellow" BorderThickness="3" Foreground="Blue" />
</DataTemplate>
```

You set the data template on a control by setting one of two properties. For content controls, you set the *ContentTemplate* property, as shown in bold here:

```
<Label Height="23" HorizontalAlignment="Left" Margin="56,0,0,91"
   Name="label1" VerticalAlignment="Bottom" Width="120">
   <Label.ContentTemplate>
      <DataTemplate>
         <!--Actual data template omitted-->
      </DataTemplate>
   </Label.ContentTemplate>
</Label>
```

For item controls, you set the *ItemsTemplate* property, as shown in bold here:

```
<ListBox ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True"
   Margin="18,19,205,148" Name="listBox1">
   <ListBox.ItemTemplate>
      <DataTemplate>
         <!--Actual data template omitted-->
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```

Note that for item controls, the *DisplayMemberPath* and *ItemTemplate* properties are mutually exclusive—you can set one but not the other.

A frequent pattern with data templates is to define them in a resource collection and reference them in your element, rather than defining them inline as shown in the previous

examples. All that is required to reuse a data template in this manner is to define the template in a resource collection and set a *Key* for the template, as shown here:

```
<Window.Resources>
   <DataTemplate x:Key="myTemplate">
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </DataTemplate>
</Window.Resources>
```

Then you can set the template by referring to the resource, as shown in bold here:

```
<ListBox ItemTemplate="{StaticResource myTemplate}"
   Name="ListBox1" />
```

Managing Data Shared Between Forms

If you must share data between forms in the Presentation layer, the best way to do so is by scoping the objects containing the data as application scope variables. This allows members in all forms in an application to access the data.

Creating an Application Variable in Windows Forms with Visual Basic

When programming in Visual Basic .NET, the easiest way to create a variable with application scope is to create a module and declare a public variable within that module. This variable then will be available to all forms in the application.

Creating an Application Variable in Windows Forms with Visual C#

When programming in Visual C#, the best way to create an application-scoped variable is to add a public, static variable to the .cs file that contains the Main sub. In Visual Studio–generated applications, this is usually Program.cs. Variables created in this fashion will be available to all forms in an application, though they will need to be prefaced with the name of the class.

Creating an Application Variable in WPF

When creating an application variable in WPF, the best way is to create an application resource that will be accessible by all objects in a particular application. You can create an application resource by opening the App.xaml file (for C# projects) or the Application.xaml file (for Visual Basic projects) and adding the resource to the *Application.Resources* collection, as shown in bold here:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>
       <SolidColorBrush x:Key="appBrush" Color="PapayaWhip" />
    </Application.Resources>
</Application>
```

Managing Media

The .NET Framework provides tools that allow you to manage sound and video media presentations in your client applications. For simple sound support, the *SoundPlayer* component is provided; and for more complex sound or video media, the *MediaPlayer* and *MediaElement* components can be used.

SoundPlayer

The *SoundPlayer* class was introduced in .NET Framework 2.0 as a managed class to enable audio in Windows applications. It is lightweight and easy to use, but it has significant limitations.

The *SoundPlayer* class can play only uncompressed .wav files. It cannot read compressed .wav files, nor can it read files in other audio formats. Furthermore, the developer has no control over volume, balance, speed of playback, or any other aspects of sound playback.

In spite of its limitations, *SoundPlayer* can be a useful and lightweight way to incorporate sound into your applications. It provides a basic set of members that allow you to load and play uncompressed .wav files easily.

MediaPlayer and *MediaElement*

The *MediaPlayer* and *MediaElement* classes provide deep support for playing audio and video media files in a variety of formats. Both of these classes use the functionality of Windows Media Player 10, so while they are guaranteed to be usable in applications running on Windows Vista and later, which come with Media Player 11 as a standard feature, these classes will not function on Windows XP installations that do not have at least Windows Media Player 10 installed.

The *MediaPlayer* and *MediaElement* classes are very similar and expose many of the same members. The primary difference between the two classes is that although *MediaPlayer* loads and plays both audio and video, it has no visual interface and thus cannot display video in the UI. On the other hand, *MediaElement* is a full-fledged WPF element that can be used to display video in your applications. *MediaElement* wraps a *MediaPlayer* object and provides a visual interface to play video files. Furthermore, *MediaPlayer* cannot be used easily in XAML, whereas *MediaElement* is designed for XAML use.

While *MediaPlayer* and *MediaElement* are designed for use in WPF applications, you can use them in your Windows Forms applications through interoperability, as described earlier in this chapter.

Objective Summary

- Data validation is a common task for the Presentation layer. Depending on the data, any of a number of different types of validation might be necessary. Both WPF and Windows Forms provide technologies to enable validation of user input.

- WPF and Windows Forms both enable binding to datasets and other collections. In Windows Forms, the *BindingSource* object manages data currency and navigation. In WPF, these tasks are managed through the *ICollectionView* interface. WPF also incorporates technology that allows direct binding to XML and the use of *XPath* queries.
- Data can be shared between multiple forms via application variables.
- The *SoundPlayer, MediaPlayer,* and *MediaElement* classes incorporate functionality that enables multimedia presentations.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of the chapter.

1. You are creating a Presentation layer that will validate user input. The fields that need to be validated include an employee age field, which must be an integer between 18 and 85. What is the correct data validation strategy for this scenario?

   A. Data type validation

   B. Range checking

   C. Lookup validation

   D. Complex validation

2. Which of the following code snippets correctly demonstrates a data template that binds the *ContactName* field set in a *ListBox*? Assume that the *DataContext* is set correctly.

   A.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <DataTemplate>
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </DataTemplate>
</ListBox>
```

   B.

```
<ListBox name="ListBox1">
   <ListBox.ItemsSource>
      <DataTemplate>
         <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
            Background="Yellow" BorderThickness="3" Foreground="Blue" />
      </DataTemplate>
   </ListBox.ItemsSource>
</ListBox>
```

C.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <ListBox.ItemTemplate>
      <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
         Background="Yellow" BorderThickness="3" Foreground="Blue" />
   </ListBox.ItemTemplate>
</ListBox>
```

D.

```
<ListBox ItemsSource="{Binding}" name="ListBox1">
   <ListBox.ItemTemplate>
      <DataTemplate>
         <Label Content="{Binding Path=ContactName}" BorderBrush="Black"
            Background="Yellow" BorderThickness="3" Foreground="Blue" />
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```

THOUGHT EXPERIMENT
Designing Data Validation

In the following thought experiment, apply what you've learned about this objective to predict how a theoretical application architecture will perform. You can find answers to these questions in the "Answers" section at the end of this chapter.

Lucerne Publishing has designed the UI for a new page-based WPF application that its data entry team can use to enter newly published books into its database. With the eight pages of the UI designed, the company is ready to begin designing the page functionality.

The process of entering information about a new book requires the data entry staff member to enter about 40 different pieces of information. Once the information has been entered to the database, a copy of it is sent to distributors, making it impossible to change. Therefore, accurate data entry is of paramount importance.

The WPF application retrieves information from and submits information to a Windows Communication Foundation (WCF) web service. The web service, in turn, communicates with the underlying database. In the near future, Lucerne Publishing plans to expose the web service to partner organizations so that it can develop its own client applications.

The developers have designed an application with the following traits:

- Numeric values are validated using WPF range validation as the user populates a field.
- Text data, such as a book's ISBN (a unique identifier for a book) are validated within WPF using regular expressions.

- **Data that refers to existing values in the database are verified by using a custom validation rule. The *ValidationRule*-derived class makes a call to the WCF web service when the user clicks the Next button.**
- **When the elements on a page can be validated, the application's title bar is green. If validation fails for any element on the page, the application changes the color of the title bar to red.**

Answer the following questions about the future performance of the application:

1. **What would you change about the data validation design?**
2. **Will indicating the validation state by changing the title bar work?**

Objective 2.5: Design Presentation Behavior

With the advent of WPF, the developer now has a great deal of control over how the UI interacts with the user. While familiar behaviors such as dragging are still easy to implement, you can also use attached events, triggers, and animation to make the user experience more responsive than ever before.

This objective covers how to:

- Determine which behaviors will be implemented and how.
- Implement drag-and-drop functionality.

Determine Which Behaviors Will Be Implemented and How

WPF provides unprecedented functionality for implementing behaviors in the Presentation layer through attached events, triggers, and animation.

Attached Events

It is possible for a control to define a handler for an event that the control cannot itself raise. These incidents are called *attached events*. For example, consider *Button* controls in a *Grid*. The *Button* class defines a *Click* event, but the *Grid* class does not. However, you still can define a handler for buttons in the grid by attaching the *Click* event of the *Button* control in the XAML code. The following example demonstrates attaching an event handler for a *Button* contained in a *Grid:*

```
<Grid Button.Click="changeGridColor">
   <Button Height="23" Margin="132,80,70,0" Name="button1"
   VerticalAlignment="Top" >Button</Button>
</Grid>
```

Now every time a button contained in the *Grid* shown here is clicked, the *changeGridColor* event handler will handle that event. In this way, the *Grid* can respond to another element's event. Attached events make it possible for any element in your Presentation layer to respond directly to an event raised by other elements.

Triggers

Along with *Setters*, *Triggers* make up the bulk of objects that you use in creating styles in WPF applications. *Triggers* allow you to implement property changes declaratively in response to other property changes that would have required event-handling code in Windows Forms programming. There are five kinds of *Trigger* objects, as listed in Table 2-12.

TABLE 2-12 Types of *Trigger* Objects

Type	Class Name	Description
Property trigger	*Trigger*	Monitors a property and activates when the value of that property matches the *Value* property.
Multi-trigger	*MultiTrigger*	Monitors multiple properties and activates only when all the monitored property values match their corresponding *Value* properties.
Data trigger	*DataTrigger*	Monitors a bound property and activates when the value of the bound property matches the *Value* property.
Multi-data-trigger	*MultiDataTrigger*	Monitors multiple bound properties and activates only when all the monitored bound properties match their corresponding *Value* properties.
Event trigger	*EventTrigger*	Initiates a series of *Actions* when a specified event is raised.

A *Trigger* is active only when it is part of a *Style.Triggers* collection—with one exception. *EventTrigger* objects can be created within a *Control.Triggers* collection outside a *Style*. The *Control.Triggers* collection can accommodate only *EventTriggers*, and any other *Trigger* placed in this collection causes an error. *EventTriggers* are used primarily with animation.

Property Triggers

The most commonly used type of *Trigger* is the property trigger. The property trigger monitors the value of a property specified by *Property*. When the value of the specified property equals the *Value* property, the *Trigger* is activated.

Triggers listen to the property indicated by *Property* and compare that property to the *Value* property. When the referenced property and the *Value* property are equal, the *Trigger* is activated. Any *Setter* objects in the *Setters* collection of the *Trigger* are applied to the style, and any *Actions* in the *EnterActions* collections are initiated. When the referenced property no longer matches the *Value* property, the *Trigger* is inactivated. All *Setter* objects in the *Setters* collection of the *Trigger* are inactivated, and any *Actions* in the *ExitActions* collection are initiated.

The following example demonstrates a simple *Trigger* object that changes the *FontWeight* of a *Button* element to *Bold* when the mouse enters the *Button:*

```
<Style.Triggers>
   <Trigger Property="Button.IsMouseOver" Value="True">
      <Setter Property="Button.FontWeight" Value="Bold" />
   </Trigger>
</Style.Triggers>
```

In this example, the *Trigger* defines one *Setter* in its *Setters* collection. When the *Trigger* is activated, that *Setter* is applied.

Multi-Triggers

Multi-triggers are similar to property triggers in that they monitor the value of properties and activate when those properties meet a specified value. The difference is that multi-triggers are capable of monitoring several properties at a single time and they activate only when all monitored properties equal their corresponding *Value* properties. The properties that are monitored and their corresponding *Value* properties are defined by a collection of *Condition* objects.

The following example demonstrates a *MultiTrigger* that sets the *Button.FontWeight* property to *Bold* only when the *Button* is focused and the mouse has entered the control:

```
<Style.Triggers>
   <MultiTrigger>
      <MultiTrigger.Conditions>
         <Condition Property="Button.IsMouseOver" Value="True" />
         <Condition Property="Button.IsFocused" Value="True" />
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
         <Setter Property="Button.FontWeight" Value="Bold" />
      </MultiTrigger.Setters>
   </MultiTrigger>
</Style.Triggers>
```

Data Triggers and Multi-Data-Triggers

Data triggers are similar to property triggers in that they monitor a property and activate when the property meets a specified value, but they differ in that the property they monitor is a bound property. Instead of *Property*, data triggers expose a *Binding* property that indicates the bound property to listen to.

The following shows a data trigger that changes the *Background* property of a *Label* to *Red* when the bound property *CustomerName* equals "Fabrikam":

```
<Style.Triggers>
   <DataTrigger Binding="{Binding Path=CustomerName}" Value="Fabrikam">
      <Setter Property="Label.Background" Value="Red" />
   </DataTrigger>
</Style.Triggers>
```

Multi-data-triggers are to data triggers as multi-triggers are to property triggers. They contain a collection of *Condition* objects, each of which specifies a bound property via its *Binding* property and a value to compare to that bound property. When all the conditions are satisfied, the *MultiDataTrigger* activates. The following example demonstrates a *MultiDataTrigger* that sets the *Label.Background* property to *Red* when *CustomerName* equals "Fabrikam" and *OrderSize* equals 500:

```
<Style.Triggers>
   <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
         <Condition Binding="{Binding Path=CustomerName}" Value="Fabrikam" />
         <Condition Binding="{Binding Path=OrderSize}" Value="500" />
      </MultiDataTrigger.Conditions>
      <MultiDataTrigger.Setters>
         <Setter Property="Label.Background" Value="Red" />
      </MultiDataTrigger.Setters>
   </MultiDataTrigger>
</Style.Triggers>
```

Event Triggers

Event triggers are different from the other *Trigger* types. Whereas other *Trigger* types monitor the value of a property and compare it to an indicated value, event triggers specify an event and activate when that event is raised. In addition, event triggers do not have a *Setters* collection—rather, they have an *Actions* collection. The following two examples demonstrate the *EventTrigger* class. The first example uses a *SoundPlayerAction* to play a sound when a *Button* is clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <SoundPlayerAction Source="C:\myFile.wav" />
</EventTrigger>
```

The second example demonstrates a simple animation that causes the *Button* to grow in height by 200 units when clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <EventTrigger.Actions>
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation Duration="0:0:5"
               Storyboard.TargetProperty="Height" To="200" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger.Actions>
</EventTrigger>
```

Animation

The term *animation* brings to mind hand-drawn anthropomorphic animals performing amusing antics in video media, but in WPF, animation has a far simpler meaning. Generally speaking, an animation in WPF refers to an automated property change over a set period

of time. You can animate an element's size, location, color, or virtually any other property or properties associated with an element. You can use the *Animation* classes to implement these changes.

The *Animation* classes are a large group of classes designed to implement these automated property changes. There are 42 *Animation* classes in the *System.Windows.Media.Animation* namespace, and each one has a specific data type that they are designed to animate. *Animation* classes fall into three basic groups: linear animations, key frame–based animations, and path-based animations.

Linear animations, which automate a property change in a linear way, are named in the format *<TypeName>Animation,* where *<TypeName>* is the name of the type being animated. *DoubleAnimation* is an example of a linear animation class, and that is the animation class you are likely to use the most.

Key frame–based animations perform their animation on the basis of several waypoints, called *key frames*. The flow of a key-frame animation starts at the beginning and then progresses to each of the key frames before ending. The progression is usually linear. Key-frame animations are named in the format *<TypeName>AnimationUsingKeyFrames,* where *<TypeName>* is the name of the *Type* being animated. An example is *StringAnimationUsingKeyFrames*.

Path-based animations use a *Path* object to guide the animation. They are used most often to animate properties that relate to the movement of visual objects along a complex course. Path-based animations are named in the format *<TypeName>AnimationUsingPath,* where *<TypeName>* is the name of the type being animated. There are currently only three path-based *Animation* classes—*PointAnimationUsingPath, DoubleAnimationUsingPath,* and *MatrixAnimationUsingPath*.

Creating Attached Behaviors

You also can create custom attached behaviors that change or extend the behavior of standard classes. Usually, developers do this to change the way a UI element interacts with the user. For example, you might create a custom attached behavior to allow a button to respond to right-clicking in an M-V-VM application, or to add drag-and-drop behavior to a class that does not support it natively.

Custom attached behaviors are simpler to implement than deriving a new class from a standard control, and they do not require you to violate the presentation pattern (as discussed in Objective 2.1) by writing behavior logic in the ViewModel.

To implement an attached behavior, follow these high-level steps:

1. Create a new static class that represents the behavior.
2. Within the class, expose any attached properties as public dependency objects. The attached property will be used to enable your attached behavior.

3. Within the change event for your attached property, write the code that implements your attached behavior.
4. In the XAML, apply the attached behavior to the element.

For example, imagine that you wanted to extend the standard text box behavior with auto-complete functionality: whenever the user typed a character, the view would send the typed phrase to a web service that returns a list of options the user can select from. You could do this by handling the appropriate text box event in the ViewModel and then populating the list from the event handler. However, this technique requires writing UI logic in your ViewModel, breaking the M-V-VM presentation pattern. It also requires writing event handlers for every text box that you want to add this behavior to.

You also could do this by creating an attached behavior. By creating an attached behavior, the behavior code is contained within the view, which fits the presentation pattern better. In addition, the attached behavior uses less, more maintainable code.

MORE INFO **CREATING ATTACHED BEHAVIORS**

For detailed information, read "Introduction to Attached Behaviors in WPF" at *http://www.codeproject.com/KB/WPF/AttachedBehaviors.aspx,* and "Attached Behavior" at *http://eladm.wordpress.com/2009/04/02/attached-behavior/.*

Implementing Drag-and-Drop Functionality

Drag-and-drop functionality is ubiquitous in Windows programming. It refers to allowing the user to grab data—such as text, an image, or another object—with the mouse and drag it to another control. When the mouse button is released over the other control, the data that is being dragged is dropped onto the control, and a variety of effects can then occur.

The drag-and-drop operation is similar to cutting and pasting. The mouse pointer is positioned over a control and the mouse button is pressed. Data is copied from a source control; when the mouse button is released, the action is completed. All code for copying the data from the source control and any actions taken on the target control must be coded explicitly.

Drag-and-drop operations are very similar in WPF and Windows Forms applications. The primary difference is that in Windows Forms, drag-and-drop methods and events are exposed on individual controls, and in WPF, the methods are exposed on a static class called *DragDrop*, which also provides attached events to WPF controls to facilitate drag-and-drop operations.

The drag-and-drop process is primarily an event-driven process. There are events that occur on the source control and events that occur on the target control. The drag-and-drop events for the source control are described in Table 2-13. The drag-and-drop events for the target control are described in Table 2-14.

TABLE 2-13 Source Control Events Involved in Implementing Drag-and-Drop Operations

Event	Description
MouseDown	Occurs when the mouse button is pressed while the pointer is over the control. In general, the *DoDragDrop* method is called in the method that handles this event. In WPF applications, this is a bubbling event.
GiveFeedBack	Provides an opportunity for the user to set a custom mouse pointer. In WPF applications, this is a bubbling event.
QueryContinueDrag	Enables the drag source to determine whether a drag event should be canceled. In WPF applications, this is a bubbling event.
PreviewMouseDown	WPF only. The tunneling version of the *MouseDown* event.
PreviewGiveFeedBack	WPF only. The tunneling version of the *GiveFeedback* event.
PreviewQueryContinueDrag	WPF only. The tunneling version of the *QueryContinueDrag* event.

TABLE 2-14 Target Control Events Involved in Implementing Drag-and-Drop Operations

Event	Description
DragEnter	Occurs when an object is dragged within a control's bounds. The handler for this event receives a *DragEventArgs* object. In WPF, this is a bubbling event.
DragOver	Occurs when an object is dragged over a target control. The handler for this event receives a *DragEventArgs* object. In WPF, this is a bubbling event.
DragDrop	Occurs when the mouse button is released over a target control. The handler for this event receives a *DragEventArgs* object. In WPF, this is a bubbling event.
DragLeave	Occurs when an object is dragged out of the control's bounds. In WPF, this is a bubbling event.
PreviewDragEnter	WPF only. The tunneling version of *DragEnter*.
PreviewDragOver	WPF only. The tunneling version of *DragOver*.
PreviewDragDrop	WPF only. The tunneling version of *DragDrop*.
PreviewDragLeave	WPF only. The tunneling version of *DragLeave*.

In addition, the *DoDragDrop* method on the source control is required to initiate the drag-and-drop process in Windows Forms, and the *DoDragDrop* method of the *DragDrop* class is required for WPF. Furthermore, the target control must have the *AllowDrop* property set to *True*.

If you are creating an XBAP application, you must run the application with full trust to take advantage of true drag-and-drop; partial trust allows only a simulated drag-and-drop using limited mouse events. For more information about partial trust, refer to Objective 1.3,

"Design the Security Implementation," in Chapter 1, "Designing the Layers of a Solution." For more information about deploying ClickOnce applications with full trust, refer to Objective 4.1, "Define a Client Deployment Strategy," in Chapter 4, "Planning a Solution Deployment."

MORE INFO **PARTIAL TRUST DRAG-AND-DROP OPERATIONS**

For detailed instructions about how to implement drag-and-drop functionality in a partial trust environment, read "Implementing Drag Drop Operations for Browser Based WPF Applications (XBAP)" at *http://www.codeproject.com/KB/WPF/XBAPDragDrop.aspx*.

The General Sequence of a Drag-and-Drop Operation

The general sequence of events that takes place in a drag-and-drop operation is as follows:

1. The drag-and-drop operation is initiated by calling the *DoDragDrop* method on the source control (for Windows Forms) or the *DragDrop.DoDragDrop* method for WPF applications. This is usually done in the *MouseDown* event handler. *DoDragDrop* copies the desired data from the source control to a new instance of *DataObject* and sets flags that specify which effects are allowed with this data.
2. The *GiveFeedBack* and *QueryContinueDrag* events are raised at this point. The *GiveFeedback* event handler can set the mouse pointer to a custom shape, and the *QueryContinueDrag* event handler can be used to determine if the drag operation should be continued or aborted.
3. The mouse pointer is dragged over a target control. Any control that has the *AllowDrop* property set to *True* is a potential drop target. When the mouse pointer enters a control with the *AllowDrop* property set to *True*, the *DragEnter* event for that control is raised. The *DragEventArgs* object that the event handler receives can be examined to determine if data appropriate for the target control is present. If so, the *Effect* property of the *DragEventArgs* object then can be set to an appropriate value.
4. The user releases the mouse button over a valid target control, raising the *DragDrop* event. The code in the *DragDrop* event handler then obtains the dragged data and takes whatever action is appropriate in the target control.

The *DragDropEffects* Enumeration

To complete a drag-and-drop operation, the drag effect specified in the *DoDragDrop* method must match the value of the *Effect* parameter of the *DragEventArgs* object associated with the drag-and-drop event, which is generally set in the *DragEnter* handler. The *Effect* property is an instance of the *DragDropEffects* enumeration. The members of the *DragDropEffects* enumeration are described in Table 2-15.

TABLE 2-15 *DragDropEffects* Enumeration Members

Member	Explanation
All	Data is copied, removed from the drag source, and scrolled in the target.
Copy	The data is copied to the target.
Link	The data is linked to the target.
Move	The data is moved to the target.
None	The target does not accept the data.
Scroll	Scrolling is about to start or is currently occurring in the target.

Note that the main function of the *Effect* parameter is to change the mouse cursor when it is over the target control. The value of the *Effect* parameter has no actual effect on the action that is executed except that when the *Effect* parameter is set to *None*, no drop can take place on that control because the *DragDrop* event will not be raised.

Initiating the Drag-and-Drop Operation in Windows Forms Applications

The drag-and-drop operation is initiated by calling the *DoDragDrop* method on the source control. The *DoDragDrop* method takes two parameters: an *Object*, which represents the data to be copied to the *DataObject*, and an instance of *DragDropEffects*, which specifies what drag effects will be allowed with this data. The following example demonstrates how to copy the text from a text box and set the allowed effects to *Copy* or *Move*:

Sample of Visual Basic.NET Code

```
Private Sub TextBox1_MouseDown(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles TextBox1.MouseDown
    TextBox1.DoDragDrop(TextBox1.Text, DragDropEffects.Copy Or DragDropEffects.Move)
End Sub
```

Sample of Visual C# Code

```
private void textBox1_MouseDown(object sender, MouseEventArgs e)
{
    textBox1.DoDragDrop(textBox1.Text, DragDropEffects.Copy | DragDropEffects.Move);
}
```

Note that you can use the *Or* operator (Visual Basic) or the | operator (C#) to combine members of the *DragDropEffects* enumeration to indicate multiple effects.

Initiating the Drag-and-Drop Operation in WPF Applications

In WPF applications, you initiate the drag-and-drop operation by calling *DragDrop.Do DragDrop*. This method takes three parameters: a *DependencyObject* that represents the source control for the drag operation, an *Object* that represents that data that will be copied

to the *DataObject*, and an instance of *DragDropEffects*, which specifies what drag effects will be allowed with this data. The following example demonstrates how to copy the text from a text box and set the allowed effects to *Copy* or *Move*:

Sample of Visual Basic.NET Code

```
Private Sub TextBox1_MouseDown(ByVal sender As System.Object, _
    ByVal e As System.Windows.Input.MouseButtonEventArgs) _
    Handles TextBox1.MouseDown
    DragDrop.DoDragDrop(TextBox1, TextBox1.Text, DragDropEffects.Copy Or
DragDropEffects.Move)
End Sub
```

Sample of Visual C# Code

```
private void textBox1_MouseDown(object sender, MouseButtonEventArgs e)
{
    DragDrop.DoDragDrop(textBox1, textBox1.Text, DragDropEffects.Copy | DragDropEffects.
Move);
}
```

Handling the *DragEnter* Event

The *DragEnter* event should be handled for every target control. This event occurs when a drag-and-drop operation is in progress and the mouse pointer enters the control. This event passes a *DragEventArgs* object to the method that handles it, and you can use the *DragEventArgs* object to query the *DataObject* associated with the drag-and-drop operation. If the data is appropriate for the target control, you can set the *Effect* property to an appropriate value for the control. The following example demonstrates how to examine the data format of the *DataObject* and set the *Effect* property:

Sample of Visual Basic.NET Code

```
 ' This is a Windows Forms example
Private Sub TextBox2_DragEnter(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.DragEventArgs) _
    Handles TextBox2.DragEnter
    If e.Data.GetDataPresent(DataFormats.Text) = True Then
        e.Effect = DragDropEffects.Copy
    End If
End Sub
' This is a WPF example
Private Sub TextBox2_DragEnter(ByVal sender As System.Object, _
    ByVal e As System.Window.DragEventArgs) _
    Handles TextBox2.DragEnter
    If e.Data.GetDataPresent(DataFormats.Text) = True Then
        e.Effect = DragDropEffects.Copy
    End If
End Sub
```

Sample of Visual C# Code

```
// The Windows Forms and WPF examples look the same in C#
private void textBox2_DragEnter (object sender, DragEventArgs e)
```

```
{
    if (e.Data.GetDataPresent(DataFormats.Text))
    {
        e.Effect = DragDropEffects.Copy;
    }
}
```

Note that in WPF applications, this is an attached event that is based off of the *DragDrop* class. You can attach this event in the Events pane of the Properties window in Visual Studio.

Handling the *DragDrop* Event

When the mouse button is released over a target control during a drag-and-drop operation, the *DragDrop* event is raised. In the method that handles the *DragDrop* event, you can use the *GetData* method of the *DataObject* to retrieve the copied data from the *DataObject* and take whatever action is appropriate for the control. The following example demonstrates how to drop a *String* into a *TextBox*:

Sample of Visual Basic.NET Code

```
 ' This is a Windows Forms example
Private Sub TextBox2_DragDrop(ByVal sender As System.Object, ByVal e As _
   System.Windows.Forms.DragEventArgs) Handles TextBox2.DragDrop
   TextBox2.Text = TryCast(e.Data.GetData(DataFormats.Text), String)
End Sub
' This is a WPF example
Private Sub TextBox2_DragDrop(ByVal sender As System.Object, ByVal e As _
   System.Windows.DragEventArgs) Handles TextBox2.DragDrop
   TextBox2.Text = TryCast(e.Data.GetData(DataFormats.Text), String)
End Sub
```

Sample of Visual C# Code

```
// The Windows Forms and WPF examples look the same in C#
private void textBox2_DragDrop(object sender, DragEventArgs e)
{
   textBox2.Text = (string)e.Data.GetData(DataFormats.Text);
}
```

Note that in WPF applications, this is an attached event that is based off the *DragDrop* class. You can attach this event in the Events pane of the Properties window in Visual Studio.

Implementing Drag-and-Drop Operations Between Applications

The system intrinsically supports drag-and-drop operations between .NET Framework applications. You don't need to take any additional steps to enable drag-and-drop operations that take place between applications. The only conditions that must be satisfied to enable a drag-and-drop operation between applications are:

- The target control must allow one of the drag effects specified in the *DoDragDrop* method call.

- The target control must accept data in the format that was set in the *DoDragDrop* method call.

Objective Summary

- Attached events allow you to enable any WPF element to respond to events raised by any other WPF element.
- Triggers allow WPF UIs to respond dynamically to user input or actions.
- Animation allows you to alter the appearance of WPF elements in real time.
- Drag-and-drop operations are supported in both Windows Forms and WPF UIs.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of the chapter.

1. Look at the following XAML snippet:

```
<Window.Resources>
   <Style x:Key="Style1">
      <Style.Triggers>
         <MultiTrigger>
            <MultiTrigger.Conditions>
               <Condition Property="TextBox.IsMouseOver"
                  Value="True" />
               <Condition Property="TextBox.IsFocused"
                  Value="True" />
            </MultiTrigger.Conditions>
            <Setter Property="TextBox.Background"
               Value="Red" />
         </MultiTrigger>
      </Style.Triggers>
   </Style>
</Window.Resources>
<Grid>
   <TextBox Style="{StaticResource Style1}" Height="21"
      Margin="75,0,83,108" Name="TextBox1"
      VerticalAlignment="Bottom" />
</Grid>
```

When will *TextBox1* appear with a red background?

A. When the mouse is over *TextBox1*

B. When *TextBox1* is focused

C. When *TextBox1* is focused and the mouse is over *TextBox1*

D. All of the above

E. Never

2. Which of the following events must be handled to execute a drag-and-drop operation? (Choose all that apply.)
   A. *MouseDown*
   B. *MouseUp*
   C. *DragLeave*
   D. *DragDrop*

THOUGHT EXPERIMENT
Designing UI Behavior

In the following thought experiment, apply what you've learned about this objective to predict how a theoretical application architecture will perform. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are a consultant for the Graphic Design Institute. The Graphic Design Institute is creating a new XBAP WPF application to allow users to upload image files. They are designing the application to be extremely user-friendly; the UI design highlights every selection they need to make.

The developers have designed a UI with these attributes:

- **Users will drag a file from Windows Explorer to the WPF window. The file name will be displayed in a text box.**
- **If the selected file has one of the valid image extensions, the button will animate.**
- **Buttons on other pages will use the same animation when the form is ready to be submitted.**

Answer the following questions about how to implement the application:

1. **How can the developers initiate the button animation?**
2. **How can they implement the animation itself?**
3. **Will the drag-and-drop operation work as expected? If not, what can the developers do?**

Objective 2.6: Design for UI Responsiveness

While the primary function of the Presentation layer is to interact with the user, you also can use the computer power of the client by handling computing tasks that do not require communication with the server or data layers. Using multithreaded techniques enables you to use the processing power of the client while maintaining responsiveness in the UI.

This objective covers how to:

- Offload operations from the UI thread.
- Report progress.
- Avoid unnecessary screen refreshes.
- Determine whether to sort and filter data on the client or server.

Offloading Operations from the UI Thread and Reporting Progress

The *BackgroundWorker* component is designed to allow you to execute time-consuming operations on a separate, dedicated thread. This allows you to run operations that take a lot of time, such as file downloads and database transactions asynchronously and allow the UI to remain responsive.

The key method of the *BackgroundWorker* component is the *RunWorkerAsync* method. When this method is called, the *BackgroundWorker* component raises the *DoWork* event. The code in the *DoWork* event handler is executed on a separate, dedicated thread, allowing the UI to remain responsive.

Announcing the Completion of a Background Process

When the background process terminates, whether because the process is completed or because the process is cancelled, the *RunWorkerCompleted* event is raised. You can alert the user to the completion of a background process by handling the *RunWorkerCompleted* event.

Returning a Value from a Background Process

You might want to return a value from a background process. For example, if your process is a complex calculation, you would want to return the end result. You can return a value by setting the *Result* property of *DoWorkEventArgs* in *DoWorkEventHandler*. This value will then be available in the *RunWorkerCompleted* event handler as the *Result* property of the *RunWorkerCompletedEventArgs* parameter.

Cancelling a Background Process

You might want to implement the ability to cancel a background process. *BackgroundWorker* supports the ability to cancel a background process, but you must implement most of the cancellation code yourself. The *WorkerSupportsCancellation* property of the *BackgroundWorker* component indicates whether the component supports cancellation. You can call the *CancelAsync* method to attempt to cancel the operation; doing so sets the *CancellationPending* property of the *BackgroundWorker* component to *True*. By polling the *CancellationPending* property of the *BackgroundWorker* component, you can determine whether to cancel the operation.

Reporting Progress of a Background Process with *BackgroundWorker*

For particularly time-consuming operations, you might want to report progress back to the primary thread. You can report progress of the background process by calling the *ReportProgress* method. This method raises the *BackgroundWorker.ProgressChanged* event and allows you to pass a parameter that indicates the percentage of progress that has been completed to the methods that handle that event. The following example demonstrates how to call the *ReportProgress* method from within the *BackgroundWorker.DoWork* event handler and then to update a *ProgressBar* control in the *BackgroundWorker.ProgressChanged* event handler:

Sample of Visual Basic.NET Code

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, _
   ByVal e As System.ComponentModel.DoWorkEventArgs) _
   Handles BackgroundWorker1.DoWork
   For i As Integer = 1 to 10
      RunTimeConsumingProcess()
      ' Calls the Report Progress method, indicating the percentage
      ' complete
      BackgroundWorker1.ReportProgress(i*10)
   Next
End Sub
Private Sub BackgroundWorker1_ProgressChanged( _
   ByVal sender As System.Object, _
   ByVal e As System.ComponentModel.ProgressChangedEventArgs) _
   Handles BackgroundWorker1.ProgressChanged
   ProgressBar1.Value = e.ProgressPercentage
End Sub
```

Sample of C# Code

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
   for (int i = 1;i < 11; i++)
   {
      RunTimeConsumingProcess();
      // Calls the Report Progress method, indicating the percentage
      // complete
      backgroundWorker1.ReportProgress(i*10);
   }

}
private void backgroundWorker1_ProgressChanged(object sender,
   ProgressChangedEventArgs e)
{
   progressBar1.Value = e.ProgressPercentage;
}
```

Note that in order to report progress with the *BackgroundWorker* component, you must set the *WorkerReportsProgress* property to *True*.

Requesting the Status of a Background Process

You can determine if a *BackgroundWorker* component is executing a background process by reading the *IsBusy* property. The *IsBusy* property returns a Boolean value. If *True*, the *BackgroundWorker* component is currently running a background process. If *False*, the *BackgroundWorker* component is idle.

Creating Process Threads

For applications that require more precise control over multiple threads, you can create new threads with the *Thread* object. The *Thread* object represents a separate thread of execution that runs concurrently with other threads. You can create as many *Thread* objects as you like, but the more threads there are, the greater the impact on performance and the greater the possibility of adverse threading conditions, such as deadlocks.

Creating and Starting a New Thread

The *Thread* object requires a delegate to the method that will serve as the starting point for the thread. This method must be a *Sub* (*void* in C#) method and must take either no parameters or a single *Object* parameter. In the latter case, the *Object* parameter is used to pass any required parameters to the method that starts the thread. Once a thread is created, you can start it by calling the *Thread.Start* method. The following example demonstrates how to create and start a new thread:

Sample of Visual Basic.NET Code

```
Dim aThread As New System.Threading.Thread(Addressof aMethod)
aThread.Start()
```

Sample of C# Code

```
System.Threading.Thread aThread = new
   System.Threading.Thread(aMethod);
aThread.Start();
```

For threads that accept a parameter, the procedure is similar, except that the starting method can take a single *Object* as a parameter and that object must be specified as the parameter in the *Thread.Start* method. An example is shown below:

Sample of Visual Basic.NET Code

```
Dim aThread As New System.Threading.Thread(Addressof aMethod)
aThread.Start(anObject)
```

Sample of C# Code

```
System.Threading.Thread aThread = new
   System.Threading.Thread(aMethod);
aThread.Start(anObject);
```

Destroying Threads

You can destroy a *Thread* object by calling the *Thread.Abort* method. This method causes the thread on which it is called to cease its current operation and to raise a *ThreadAbortException*. If there is a *Catch* block that is capable of handling the exception, it will execute along with any *Finally* blocks. The thread then is destroyed and cannot be restarted.

Sample of Visual Basic.NET Code

```
aThread.Abort()
```

Sample of C# Code

```
aThread.Abort();
```

Synchronizing Threads

Two of the most common difficulties involved in multithread programming are deadlocks and race conditions. A deadlock occurs when one thread has exclusive access to a particular variable and then attempts to gain exclusive access to a second variable at the same time that a second thread has exclusive access to the second variable and attempts to gain exclusive access to the variable that is locked by the first thread. The result is that both threads wait indefinitely for the other to release the variables and they cease operating.

A race condition occurs when two threads attempt to access the same variable at the same time. For example, consider two threads that access the same collection. The first thread might add an *object* to the collection. The second thread then might remove an object from the collection based on the index of the object. The first thread then might attempt to access the object in the collection to find that it had been removed. Race conditions can lead to unpredictable effects that can destabilize your application.

The best way to avoid race conditions and deadlocks is by careful programming and judicious use of thread synchronization. You can use the *SyncLock* keyword in Visual Basic and the *lock* keyword in C# to obtain an exclusive lock on an object. This allows the thread that has the lock on the object to perform operations on that object without allowing any other threads to access it. Note that if any other threads attempt to access a locked object, those threads will pause until the lock is released. The following example demonstrates how to obtain a lock on an object:

Sample of Visual Basic.NET Code

```
SyncLock anObject
   ' Perform some operation
End SyncLock
```

Sample of C# Code

```
lock (anObject)
{
   // Perform some operation
}
```

Some objects, such as collections, implement a synchronization object that should be used to synchronize access to the greater object. The following example demonstrates how to obtain a lock on the *SyncRoot* object of an *ArrayList* object:

Sample of Visual Basic.NET Code

```
Dim anArrayList As New System.Collections.ArrayList
SyncLock anArrayList.SyncRoot
   ' Perform some operation on the ArrayList
End SyncLock
```

Sample of C# Code

```
System.Collections.Arraylist anArrayList = new System.Collections.ArrayList();
lock (anArrayList.SyncRoot)
{
   // Perform some operation on the ArrayList
}
```

It is generally good practice when creating classes that will be accessed by multiple threads to include a synchronization object that is used for synchronized access by threads. This allows the system to lock only the synchronization object, thus conserving resources by not having to lock every single object contained in the class. A synchronization object is simply an instance of *Object* and does not need to have any functionality except to be available for locking. The following example demonstrates a class that exposes a synchronization object:

Sample of Visual Basic.NET Code

```
Public Class aClass
   Public SynchronizationObject As New Object()
   ' Insert additional functionality here
End Class
```

Sample of C# Code

```
public class aClass
{
    public object SynchronizationObject = new Object();
   // Insert additional functionality here
}
```

Special Considerations When Working with Controls

Because controls are always owned by the UI thread, it is generally unsafe to make calls to controls from a different thread. In WPF applications, you can use the *Dispatcher* object, discussed later in this section, to make safe function calls to the UI thread. In Windows Forms applications, you can use the *Control.InvokeRequired* property to determine if it is safe to make a call to a control from another thread. If *InvokeRequired* returns *False*, it is safe to make the call to the control. If *InvokeRequired* returns *True*, however, you should use the *Control.Invoke* method on the owning form to supply a delegate to a method to access the control. Using *Control.Invoke* allows the control to be accessed in a thread-safe manner. The following example demonstrates setting the *Text* property of a *TextBox* control named *Text1*:

Sample of Visual Basic.NET Code

```
Public Delegate Sub SetTextDelegate(ByVal t As String)
Public Sub SetText(ByVal t As String)
   If TextBox1.InvokeRequired = True Then
      Dim del As New SetTextDelegate(AddressOf SetText)
      Me.Invoke(del, New Object() {t})
   Else
      TextBox1.Text = t
   End If
End Sub
```

Sample of C# Code

```
public delegate void SetTextDelegate(string t);
public void SetText(string t)
{
   if (textBox1.InvokeRequired)
   {
      SetTextDelegate del = new SetTextDelegate(SetText);
      this.Invoke(del, new object[]{t});
   }
   else
   {
      textBox1.Text = t;
   }
}
```

In the preceding example, the method tests *InvokeRequired* to determine if it is dangerous to access the control directly. In general, this will return *True* if the control is being accessed from a separate thread. If *InvokeRequired* does return *True*, the method creates a new instance of a delegate that refers to itself and calls *Control.Invoke* to set the *Text* property in a thread-safe manner.

Using *Dispatcher* to Access Controls Safely on Another Thread in WPF

At times, you might want to change the UI from a worker thread. For example, you might want to enable or disable buttons based on the status of the worker thread, or provide more detailed progress reporting than is allowed by the *ReportProgess* method. The WPF threading model provides the *Dispatcher* class for cross-thread calls. Using *Dispatcher*, you can update your UI safely from worker threads.

You can retrieve a reference to the *Dispatcher* object for a UI element from its *Dispatcher* property, as shown here:

Sample of Visual Basic.NET Code

```
Dim aDisp As System.Windows.Threading.Dispatcher
aDisp = Button1.Dispatcher
```

Sample of C# Code

```
System.Windows.Threading.Dispatcher aDisp;
aDisp = button1.Dispatcher;
```

Dispatcher provides two principal methods that you will use: *BeginInvoke* and *Invoke*. Both methods allow you to call a method safely on the UI thread. The *BeginInvoke* method allows you to call a method asynchronously, and the *Invoke* method allows you to call a method synchronously. Thus, a call to *Dispatcher.Invoke* will block execution on the thread on which it is called until the method returns, whereas a call to *Dispatcher.BeginInvoke* will not block execution.

Both the *BeginInvoke* and *Invoke* methods require you to specify a delegate that points to a method to be executed. You also can supply a single parameter or an array of parameters for the delegate, depending on the requirements of the delegate. You also are required to set the *DispatcherPriority* property, which determines the priority with which the delegate is executed. In addition, the *Dispatcher.Invoke* method allows you to set a period of time for the *Dispatcher* to wait before abandoning the invocation. The following example demonstrates how to invoke a delegate named *MyMethod* using *BeginInvoke* and *Invoke*:

Sample of Visual Basic.NET Code

```
Dim aDisp As System.Windows.Threading.Dispatcher = Button1.Dispatcher
' Invokes the delegate synchronously
aDisp.Invoke(System.Windows.Threading.DispatcherPriority.Normal, MyMethod)
' Invokes the delegate asynchronously
aDisp.BeginInvoke(System.Windows.Threading.DispatcherPriority.Normal, MyMethod)
```

Sample of C# Code

```
System.Windows.Threading.Dispatcher aDisp = button1.Dispatcher;
// Invokes the delegate synchronously
aDisp.Invoke(System.Windows.Threading.DispatcherPriority.Normal, MyMethod);
// Invokes the delegate asynchronously
aDisp.BeginInvoke(System.Windows.Threading.DispatcherPriority.Normal, MyMethod);
```

Avoiding Unnecessary Screen Refreshes

Client applications refresh the screen when the visible contents of a control are updated. Refreshing the screen is processor-intensive, and can negatively affect the performance of applications such as Remote Desktop, which transfer the UI across the network. While most Windows applications will work fine without any special planning, there are several best practices you can follow to avoid unnecessary screen refreshes:

- Reduce the default animation rate of 60 frames per second (fps). The following code sample sets it to 20 fps:

  Sample of Visual Basic.NET Code

  ```
  Timeline.DesiredFrameRateProperty.OverrideMetadata( _
      GetType(Timeline), _
      New FrameworkPropertyMetadata() With {Key.DefaultValue = 20})
  ```

  Sample of C# Code

  ```
  Timeline.DesiredFrameRateProperty.OverrideMetadata(
      typeof(Timeline),
      new FrameworkPropertyMetadata { DefaultValue = 20 }
  );
  ```

- Reduce the animation rate for a *Storyboard* or individual objects within a *Storyboard*. By default, animations run at 60 fps. The lower the value of the *Timeline.DesiredFrame Rate* attached property, the better your performance will be.
- WPF will refresh controls automatically when you specify a *DependencyProperty* or implement *INotifyPropertyChanged*. Avoid rapidly updating values that will initiate a refresh; for example, by updating the value within a loop. Instead, update the value once after the loop has completed.

Determining Whether to Sort and Filter Data on the Client or Server

Many tasks can be performed on either the client or the server. The most important considerations are as follows:

- **Security** Security-related tasks, such as authentication, authorization, auditing, and data validation, always must occur on the server. Client applications should never be trusted.
- **Responsiveness** Perform tasks on the client to provide the best responsiveness. For example, imagine a WPF application that retrieves a list of products from a WCF web service. If the user wants to sort the list differently, the WPF application could re-sort the list in memory, or it could send a second request to the web service to retrieve the list in a different order. Sending the request to the web service incurs a delay, however, because the request and response must be sent across the network. The higher the network latency, the longer the delay.
- **Client and server load** Performing tasks on the client increases the load on the client, and performing tasks on the server increases the load on the server. For clients with sufficient processing power, perform the task on the client to improve server scalability. If your client computers do not have the processing power necessary to perform a task in a reasonable time, perform the task on the server.
- **Network utilization** Every task that you perform on the server requires data to be sent from the client to the server. This increases network utilization. While it might be difficult to notice the impact on a higher-performance local area network (LAN), wireless and wide area network (WAN) links can be saturated much more easily, slowing the performance of every application running on the network.

As a general rule, process non-security Presentation layer tasks on the client, and process Business Logic layer tasks on the server. If a particular Business Logic layer task would be much faster to process on the client, then create a separate client-side Business Logic layer assembly, and perform that processing on the client (but validate the data on the server).

Addressing UI Memory Issues

UI elements can consume a great deal of memory if not used carefully. Follow these best practices to avoid common UI-related memory leaks in WPF:

- Un-register event handlers from a child window to a parent window when you no longer need them. Otherwise, the child window will remain in memory until the parent window is closed, even if the user closes the child window.
- Un-register event handlers to static objects when you no longer need them. Static objects always stay alive in memory.
- Stop *Timer* objects when you no longer need them.
- Set the *TextBox.UndoLimit* property if you plan to update a *TextBox* repeatedly.
- If a data-binding target refers back to the class that is bound to it, manually remove the binding before closing the window. For detailed information, refer to Microsoft Knowledge Base article 938416 at *http://support.microsoft.com/kb/938416/*.
- Freeze objects whenever possible. Freezing an object disables change notifications, improving performance and reducing memory usage. For example, if you create a brush to set the background color of an object, the .NET Framework must monitor that brush for changes so that the changes can be reflected in any objects that use the brush. Freezing the brush would spare the .NET Framework from having to do that task. Freezable objects derive from the *Freezable* class and are usually graphics-related, including bitmaps, brushes, pens, and animations. Before freezing an object, verify that the value of the *CanFreeze* property is true. For detailed information, read "Freezable Objects Overview" at *http://msdn.microsoft.com/library/ms750509.aspx*.
- Use the CLR Profiler, described in Objective 5.3 in Chapter 5, to identify memory leaks. If you examine the working set using Task Manager or Performance Monitor, you will see an exaggerated memory size. To get a more realistic size, minimize your .NET Framework application. The CLR Profiler is available as a free download at *http://www.microsoft.com/download/en/details.aspx?id=16273*.

MORE INFO **ADDRESSING MEMORY LEAKS**

For more information, read "Finding Memory Leaks in WPF-based applications" at *http://blogs.msdn.com/b/jgoldb/archive/2008/02/04/finding-memory-leaks-in-wpf-based-applications.aspx*, "Memory Leaks in WPF applications" at *http://svetoslavsavov.blogspot.com/2010/05/memory-leaks-in-wpf-applications.html*, and "Top 11 WPF Performance Tips" at *http://www.wpftutorial.net/10PerformanceTips.html*. For information about why examining the working set using Task Manager or Performance Monitor is misleading, read "How much memory does my .NET application use?" at *http://www.itwriting.com/dotnetmem.php*.

Objective Summary

- The *BackgroundWorker* component encapsulates a worker thread and provides methods to report progress from that thread.
- You can create new threads directly using the *Thread* object.
- When using threads that communicate directly with the UI thread, you must take care to avoid cross-thread function calls. In Windows Forms interfaces, query the *Control.InvokeRequired* property to determine if *Invoke* is required. In WPF, use the *Dispatcher* object to communicate safely with the UI thread.

Objective Review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the chapter.

1. Which of the following are required to start a background process with the *BackgroundWorker* component? (Choose all that apply.)
   - **A.** Calling the *RunWorkerAsync* method
   - **B.** Handling the *DoWork* event
   - **C.** Handling the *ProgressChanged* event
   - **D.** Setting the *WorkerSupportsCancellation* property to *True*
2. Which of the following are good strategies for updating the UI from the worker thread? (Choose all that apply.)
   - **A.** Use *Dispatcher.BeginInvoke* to execute a delegate to a method that updates the UI.
   - **B.** Invoke a delegate to a method that updates the UI.
   - **C.** Set the *WorkerReportsProgress* property to *True*, call the *ReportProgress* method in the background thread, and handle the *ProgressChanged* event in the main thread.
   - **D.** Call a method that updates the UI from the background thread.

THOUGHT EXPERIMENT
Designing a Responsive User Interface

In the following thought experiment, apply what you've learned about this objective to predict how a theoretical application architecture will perform. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are a consultant for Margie's Travel. Margie's Travel recently began developing a new WPF application that its travel agents will use to book flights, hotels, and activities. The priorities for the application are:

- To remain responsive when the application looks up information using public web services
- To run well on low-powered computers with limited memory
- To guide agents through the UI using animations to notify them of the progress of web services and to draw their attention to the next step

The developers have designed an application with these attributes:

- All web service requests occur in background processes.
- The agent can sort lists by price, time, or company by clicking column headings. Sorting occurs at the client.
- Pictures and video related to hotels and activities are loaded only when the user clicks a button.

Answer the following questions about the future performance of the application:

1. Will using a background process allow the application to remain responsive while web service requests occur? If so, what would you recommend the developers do differently?
2. Is it the right choice to sort lists on the client, or should the sorting occur on the server? Why?
3. Might the animations affect the performance of lower-powered computers? If so, what would you recommend the developers do differently?
4. Are there any free tools you could recommend to catch memory leaks in the application?

Chapter Summary

- There are two dominant technologies for Presentation layer development—Windows Forms and WPF. Both have inherent advantages and disadvantages. Windows Forms has a strong and dedicated developer base and offers superior globalization and localization technology. WPF is relatively new and has not been adopted as strongly by developers, but it offers substantial improvements in the interactivity of the UI. You can use Windows Forms and WPF controls in the other application type through interoperability.
- Your UI should be designed with the user in mind. The principles of good UI include:
  - Structure
  - Simplicity
  - Visibility
  - Feedback
  - Tolerance
  - Reuse
- You should design for inheritance and code reuse whenever possible. Use of resources in WPF and extended controls in Windows Forms are examples of code reuse.
- Your UI should be designed for accessibility, and thus it should support standard system settings, ensure compatibility with a high-contrast mode, provide keyboard access to all important functionality, provide notification of the keyboard focus location, and convey no important information by sound alone.
- Application workflow can be simple or complex. For applications that require navigation, Navigation applications in WPF provide a range of options from simple navigation to highly structured branching. Windows Forms applications can use MDI technology for navigation.
- Data validation is most commonly handled in the Presentation layer. Both Windows Forms and WPF provide technology to create and implement validation rules in your UI. Management of data binding is usually handled through an intermediary class—the *DataSource* in Windows Forms and *ICollectionView* in WPF. In addition, WPF provides technology that enables direct binding to XML data.
- WPF provides a great deal of technology to implement different behaviors in the UI, including attached events, triggers, and animation. Drag-and-drop functionality is very similar in both Windows Forms and WPF.
- The *BackgroundWorker* object encapsulates a background task and contains functionality to enable stopping and starting the task, as well as reporting to the main thread. For finer control, *Thread* objects can be created directly. In Windows Forms, you must query the *Control.InvokeRequired* property of a control to ensure safe cross-thread access. In WPF, you can use the *Dispatcher* object for safe cross-thread access.

Answers

This section contains the answers to the Object Reviews and the Thought Experiments.

Objective 2.1: Review

1. **Correct Answer:** B
   A. **Incorrect:** The *WindowsFormsHost* allows you to host a Windows Forms control in a WPF application.
   B. **Correct:** The *ElementHost* allows you to host a WPF control in a Windows Forms application.
   C. **Incorrect:** The *Grid* element is a container that hosts other controls in a WPF application, but it cannot be used independently in Windows Forms applications.
   D. **Incorrect:** The *Form* class is the base class for Windows Forms application, but it cannot host WPF controls independently.
2. **Correct Answer:** D
   A. **Incorrect:** Windows Forms lack the required responsiveness to styles for this application.
   B. **Incorrect:** WPF does not have an inherent *MaskedTextBox* control.
   C. **Incorrect:** Windows Forms lack the required responsiveness to styles, and added WPF controls would not be helpful in this instance.
   D. **Correct:** A WPF application will be responsive to styles, but you can use the Windows Forms *MaskedTextBox* control for the specialized input.

Objective 2.1: Thought Experiment

1. For this application, WPF is the best choice for implementing the Presentation layer. While there are some globalization and localization concerns for which Windows Forms would offer a superior development experience, they are fairly minor, requiring no changes in the presentation language of the UI, and these concerns are outweighed by the WPF advantages in the use of Styles and multimedia presentation.
2. User responsiveness can be maintained by using the *BackgroundWorker* component to process the background tasks while the UI remains responsive. When the UI needs to be updated from the background thread, this can be done safely using this *Dispatcher* object.

Objective 2.2: Review

1. **Correct Answer:** C
   A. **Incorrect:** It is not necessary to provide audio cues for all important information, and it is important to convey no information by sound alone.

B. **Incorrect:** An accessible application should supply support for standard system settings.

C. **Correct:** An accessible application should ensure compatibility with the high-contrast mode.

D. **Incorrect:** An accessible application should provide documented keyboard access to all important functionalities.

2. **Correct Answer:** D

A. **Incorrect:** Because these *Brush* objects need to be used in many different elements, defining them at the individual element level would require a great deal of redundancy.

B. **Incorrect:** Because these *Brush* objects need to be used in each window, defining them at the window level would be redundant.

C. **Incorrect:** Although defining these resources at the application level would enable use across the entire application, it would not facilitate reuse across multiple applications.

D. **Correct:** By defining your resources in a resource dictionary, you can share the file between applications, as well as import the resources contained therein at the window or application level as needed.

Objective 2.2: Thought Experiment

1. It's a good start. However, they could simplify development and improve consistency by using styles instead.

2. Maybe. However, for accessibility, it's better to use standard system settings for size, color, and font. When you use standard system settings, any changes the user makes to the operating system to better support accessibility needs will be reflected in the application. Since the organization is concerned about it, it should perform accessibility testing, as described in Objective 5.2, "Evaluate and Recommend a Test Strategy," in Chapter 5, "Designing for Stability and Maintenance."

Objective 2.3: Review

1. **Correct Answer:** C

A. **Incorrect:** The *Page* or *PageFunction* for the custom journal entry must implement *IProvideCustomContent*.

B. **Incorrect:** You must call *NavigationService.AddBackEntry* to create the custom back entry.

C. **Correct:** You do not need an instance of *JournalEntry* to create a custom back entry.

D. **Incorrect:** You must create a class that inherits from *CustomContentState* to store the state of the page.

2. **Correct Answer:** B

   A. **Incorrect:** *FragmentNavigation* occurs after *LoadCompleted,* and *Navigation-Progress* occurs after *Navigated*.

   B. **Correct:** This is the correct order in which navigation events fire.

   C. **Incorrect:** *NavigationProgress* occurs after *Navigated*.

   D. **Incorrect:** *FragmentNavigation* occurs after *LoadCompleted*.

Objective 2.3: Thought Experiment

1. No, a WPF page-based application is the best choice. Page-based applications are perfect for completing straightforward, linear tasks such as data entry.
2. Yes. You can remove pages from the journal by calling *NavigationService.Remove BackEntry* repeatedly.
3. Yes (though it might be confusing to the user). To accomplish this, call *NavigationService.AddBackEntry* and provide an instance of a class that derives from *CustomContentState* and contains the state information for the page that the user should return to when she clicks Back.

Objective 2.4: Review

1. **Correct Answer:** B

   A. **Incorrect:** Although data type validation will validate that the input is an integer, it will not validate the allowed range of values correctly.

   B. **Correct:** Range checking will validate both data type and range.

   C. **Incorrect:** Lookup validation is not required because the expected data falls within a consistent range.

   D. **Incorrect:** Complex validation is not required because the expected data falls within a consistent range.

2. **Correct Answer:** D

   A. **Incorrect:** The <*DataTemplate*> tags must be enclosed in <*ListBox. Item- Template*> tags to set the data template to the *ItemTemplate* property.

   B. **Incorrect:** The data template must be set to the *ItemTemplate* property, not the *ItemsSource* property.

   C. **Incorrect:** The data template must be enclosed in <*DataTemplate*> tags.

   D. **Correct:** The data template is set correctly.

Objective 2.4: Thought Experiment

1. Overall, the design is solid. However, it has one major flaw: all data validation occurs on the client. Client data validation is useful for improving the responsiveness of the application; however, for security and data integrity, you also must implement data validation on the server. This is particularly important because Lucerne plans to expose the web service to other clients, which might not implement proper data validation.
2. It might work, but it's not a good UI design choice. First, it fails to indicate which element on a page failed to validate. Second, it provides poor accessibility because it would not be meaningful to users who could not distinguish the colors red and green.

Objective 2.5: Review

1. **Correct Answer:** C
   - A. **Incorrect:** A multi-trigger requires that all conditions be met before activating its *Setters*. In this case, Textbox1 is not focused.
   - B. **Incorrect:** A multi-trigger requires that all conditions be met before activating its *Setters*. In this case, the mouse is not over *Textbox1*.
   - C. **Correct:** A multi-trigger is active when all conditions defined in it are met.
   - D. **Incorrect:** A multi-trigger requires that all conditions be met before activating its *Setters*, and will only activate when all conditions are met.
   - E. **Incorrect:** When all the conditions are met, the multi-trigger activates.
2. **Correct Answer:** D
   - A. **Incorrect:** Although most drag-and-drop operations begin in the *MouseDown* event on the source control, it is not required that they begin there.
   - B. **Incorrect:** Although it is recommended that the *DragEnter* event handler be used to examine the data object and set the *Effect* property as appropriate, it is not required.
   - C. **Incorrect:** The *DragLeave* event is used to execute code when data is dragged out of a control, but it is not necessary for the drag-and-drop operation.
   - D. **Correct:** The *DragDrop* event is the only event that must be handled to complete a drag-and-drop operation.

Objective 2.5: Thought Experiment

1. There are many ways to initiate the animation. The simplest would be to use validation and validate that the file name ends in one of the acceptable image file extensions.
2. There are also many ways to implement the animation. However, an attached behavior would simplify adding the behavior to multiple buttons.

3. Not if it's running in a partial trust environment, which most XBAP applications do. They either need to configure the application for full trust (which supports true drag-and-drop functionality) or they need to simulate drag-and-drop functionality in the partial trust environment.

Objective 2.6: Review

1. **Correct Answers:** A and B
   - A. **Correct:** *RunWorkerAsync* raises the *DoWork* event, which must be handled to run code on the background thread.
   - B. **Correct:** The *DoWork* event handler contains the code that will be run on the background thread.
   - C. **Incorrect:** Handling the *ProgressChanged* event is not required.
   - D. **Incorrect:** Setting *WorkerSupportsCancellation* to *True* is not required.
2. **Correct Answers:** A and C
   - A. **Correct:** The *Dispatcher.BeginInvoke* method will execute a method asynchronously and safely on the main thread.
   - B. **Incorrect:** Direct access of the UI from a background thread is not allowed.
   - C. **Correct:** Using the built-in mechanism of *BackgroundWorker* for reporting is the preferred way to report progress that can be expressed numerically.
   - D. **Incorrect:** Direct access of the UI from a background thread is not allowed.

Objective 2.6: Thought Experiment

1. Yes.
2. Yes, the sorting should occur on the client. Sorting by these values should happen quickly, even on lower-powered clients. In addition, because the application is connecting to public web services, the application might not have the ability to request the list to be sorted differently by the server.
3. Yes. They could lower the frame rate of the animations to reduce the performance impact.
4. Yes, the CLR Profiler is an excellent tool for identifying memory leaks, and it can be downloaded for free from Microsoft.

Index

A

acceptance testing, 277
accessibility requirements
  design considerations, 109
  high-contrast option, 110, 280
  keyboard focus location, 110
  keyboard navigation, 110
  notification considerations, 110
  standard system settings, 109
  Windows Forms controls, 111
accessibility testing, 280
accessing data. *See* databases
Access (Microsoft), 178
AD DS (Active Directory Domain Services)
  delegation, 38
  federated security, 42
administrative privileges, 31
ADO.NET
  about, 174
  flat file support, 174, 178
ADO.NET data services, 176–177
ADO.NET Sync Services, 195
Animation classes, 147–148
API testing, 278
App.config file, 243
AppDomain class, 34
AppDomainSetup class, 34
application domains, 34
application life cycle, 225
Application Logic layer. *See* Business Logic layer
application resources, 106
application variables, 139–140
application workflow
  design considerations, 113
  designing for different input types, 126
  implementing user navigation, 114–117
  Navigation applications in WPF, 117–124
  PageFunction class, 124–125
  simple navigation, 125
  structured navigation, 125
assemblies
  accessing from unmanaged code, 52
  version number considerations, 26
Assembly Registration Tool (RegAsm.exe), 53
asynchronous services, 24
Atom feeds, 198
attached events, 143
authentication
  claims-based, 43
  designing trusted subsystems, 39–41
  hashed passwords, 47
  HTTPS support, 46
  impersonation and delegation considerations, 37
  planning role management, 44–45
authorization
  designing model, 51
  federated security, 41–43
autonomous services, 3

B

background processes
  announcing completion of, 156
  cancelling, 156
  reporting progress of, 157
  requesting status of, 158
  returning values from, 156
BackgroundWorker component, 156–161
backward compatibility, 25–26
basicHttpBinding binding type, 16
BehaviorExtensionElement class, 23

binding data
choosing binding types, 16–17
developing design strategy, 134–139
long-running processes and, 58–59
BindingSource class, 134–135
BizTalk Server, 10–11
black box testing
about, 276
acceptance testing and, 277
range testing, 278
unit testing and, 278
"black hat" hackers, 279
buffer overflow attacks, 279
Business Logic layer. *See also* WCF web services
about, 4
choosing between Presentation layer and, 6–7
determining component installation order, 256
exception handling, 270
sample implementations, 5, 254–255
Button class, 57, 104

C

CA (certification authority), 48, 228
caching data
design considerations, 189–190
web services, 190–191
CancelEventArgs class, 130
capacity testing, 281
CAS (code access security), 31–34
CDNs (content delivery networks), 47
central deployment, 243
centralized logging, 292
certificates, deploying, 228
certification authority (CA), 48, 228
CertMgr.exe tool, 228
change management strategies for schemas, 184
chatty application communications, 64
chunky application communications, 64
claims-based authentication, 43
C# language, 159
ClickOnce
deployment strategies, 226–228, 231
update strategies, 247–248
client-side tasks. *See also* Presentation layer
choosing between server-side tasks and, 6–7
data validation, 130
defining deployment strategy, 226–236
designing for long-running processes, 57
exception handling, 272
logging events, 292
sorting/filtering data considerations, 163–164
cloud computing, 63
CLR (Common Language Runtime)
MSL support, 186
partial trust and, 32–33
CLR Profiler, 164, 294
code access security (CAS), 31–34
code coverage criteria (testing), 282
cohesion, defined, 14–15
CollectionViewSource class, 136
COM (Component Object Model)
about, 52
accessing assemblies from unmanaged code, 52
accessing COM objects, 53–54
Common Language Runtime (CLR)
MSL support, 186
partial trust and, 32–33
communicational cohesion, 15
COM objects
accessing, 53–54
deploying, 234
Component Object Model (COM)
about, 52
accessing assemblies from unmanaged code, 52
accessing COM objects, 53–54
Computer Management console
Event Trace Sessions snap-in, 286
Event Viewer snap-in, 285
Performance Monitor tool, 294
WMI support, 290
Conceptual Schema Definition Language (CSDL), 186
concurrency
cross-tier distributed transactions, 207
defined, 203
design considerations, 203
designing for web services, 206
planning for multiuser conflicts, 203–205
concurrency issue, defined, 204
condition coverage (testing criteria), 282
connection objects, 174
Constantine, Larry, 101
constrained delegation, 38–39
content-based routing, 8–10
content delivery networks (CDNs), 47
context-based routing, 8

contracts
data, 19–22
message, 19
services sharing, 3
controls. *See* Windows Forms controls; WPF controls
conversational exchanges, 25
Coordinated Universal Time (UTC), 72
credit card processing services, 64
cross-thread function calls, 161–162
cross-tier distributed transactions, 207–208
cryptography
about, 45
digital signatures, 48–49
federated security and, 42
IPsec, 46
SSL, 46–47
storing passwords, 47–48
Crystal Reports reporting tool, 292
CSDL (Conceptual Schema Definition Language), 186
CSV format, 178
CsvReader library, 178
CultureAndRegionInfoBuilder class, 71
CultureInfo class, 74
CurrentCulture class, 70, 74
CurrentUICulture class, 70

D

DAC (data-tier projects), 239
.dacpac format, 237, 239
Database Publishing Wizard, 240–241
databases
about, 174
choosing access strategies, 174–181
concurrent multiuser access, 203–210
deploying privately, 242
designing data caching, 189–193
designing data object model, 181–188
designing offline storage, 194–197
designing synchronization, 198–201
embedded, 242
planning deployment, 237–246
publishing from Server Explorer, 240–241
publishing with WCF web services, 241–242
synchronizing schemas between, 238
data binding
choosing binding types, 16–17
developing design strategy, 134–139
long-running processes and, 58–59
data caching
design considerations, 189–190
web services, 190–191
Data Collector Sets, 288, 294
DataContract attribute, 19, 183
data contracts, 20–88
DataContractSerializer class, 183
data integrity, 24
Data layer
about, 4, 173
analyzing data services for optimization, 210–215
choosing appropriate access strategy, 174–181
designing concurrent multiuser environment, 203–210
designing data caching, 189–193
designing data object model, 181–188
designing data synchronization, 194–203
designing offline storage, 194–203
determining component installation order, 256
sample implementations, 6–7, 254–255
DataMember attribute, 20, 183
DataObject class, 150–153
data object model
abstracting from service layer, 185–186
defined, 181
mapping to persistent storage, 182–184
schema change management strategies, 184
DataReader class, 174
DataService class, 177
DataSet class, 134, 174
DataTable class, 174
data templates, 138
data-tier projects (DAC), 239
Data Transfer Objects (DTOs), 210, 213
data triggers, 145
data validation
client-side, 130
complex validation, 130
data type validation, 129
error handling, 132–134
implementing custom rules, 132
lookup validation, 129
range checking, 129
server-side, 130
trust boundaries and, 24
WPF applications, 131

DateTime class, 72
DateTimeOffset class, 73
.dbschema files, 237, 238, 240
DbServerSyncProvider class, 196, 198
deadlock conflicts, 205
decision coverage (testing criteria), 282
deferred loading, 212–213
delegation
  authentication and, 39
  constrained, 38–39
  recommended usage, 36
DELETE verb (HTTP), 18
demilitarized zone (DMZ), 254
denial-of-service (DoS) attacks, 279, 293
deployment, solution
  defining client strategy, 226–236
  designing update strategies, 246–253
  planning considerations, 225
  planning for databases, 237–246
  planning for n-tier applications, 253–258
deserialization, file mapping and, 183
diagnostics and monitoring
  centralized logging, 292
  design considerations, 285, 293–294
  distributed logging, 292
  profiling technique, 294–295
  providing monitoring information, 285–291
  usage reporting, 288, 291–292
dialog boxes
  creating custom, 102
  file, 93
  modal, 103
  modeless, 103
  in WPF applications, 93
DialogResult class, 103
Dispatcher class, 160–161
DispatcherUnhandledException event, 273
distributed transactions, 207–208
distribution points, defined, 229
DMZ (demilitarized zone), 254
DoS (denial-of-service) attacks, 279, 293
DoubleAnimationUsingPath class, 147
drag-and-drop operations
  about, 148
  DragDropEffects enumeration, 150–151
  DragDrop event, 153
  DragEnter event, 152
  general sequence, 150
  implementing between applications, 153
  initiating in Windows Forms, 151
  source control events, 149
  target control events, 149
  WPF applications, 151
DragDropEffects enumeration, 150–151
DragDrop event, 150, 153
DragEnter event, 150, 152
DragEventArgs object, 150
dstributed logging, 292
DTOs (Data Transfer Objects), 210, 213
duration testing, 282
dynamic resources, 106–107

E

eager loading, 212–213
ElementHost control, 92
embedded databases, deploying privately, 242
encryption, defined, 45
endurance testing, 282
Entity Framework
  about, 174–175
  abstracting data model, 185–186
  choosing as data access strategy, 179
  database mapping, 183
  lazy loading, 212
  optimistic locking, 204
  ORM and, 211–212
  SQL Profiler tool, 295
error handling
  across tiers, 267–270
  best practices, 266–267
  collecting user feedback, 270–272
  creating custom exception classes, 272
  data validation, 132–134
  designing strategies, 266–267
  processing unhandled exceptions, 272
  stability testing and, 277
Error object, 133
evaluating
  conceptual design, 100–172
  test strategies, 275–285
Event Forwarding, 292, 294
event handling
  attached events, 143
  drag-and-drop operations, 148–154

exceptions, 273
Navigation applications, 122–124
Windows Forms controls, 96
EventLog class, 288
Event Log (Windows)
about, 285, 286–288
centralized logging, 292
collecting user feedback, 270
distributed logging, 292
term usage for event, 286
events
defined, 286
logging client-side, 292
monitoring via Event Log, 285–288
EventSetter class, 104
event setters, 104
Event Tracing For Windows, 286, 289
event triggers, 146
Event Viewer snap-in, 285
Exception class, 266–267
exception handling
across tiers, 267–270
best practices, 266–267
collecting user feedback, 270–272
creating custom exception classes, 272
data validation, 132–134
designing strategies, 266–267
processing unhandled exceptions, 272
stability testing and, 277
explicit service boundaries, 2
extended controls, 101
Extensible Application Markup Language (XAML)
accessing resources, 106
<EventSetter> tag, 104
<Setter> tag, 104
WPF support, 91
external systems, interoperability with
accessing assemblies from unmanaged code, 52
accessing COM objects, 53–54
typical methods, 52

F

FaultException class, 268–269
federated security, 41–43
feedback, collecting for error handling, 270–272
file dialog boxes, 93–94
FileHelpers library, 178
FileIOPermission class, 34
file sharing, 52
file version number, 26
filtering data, 163–164
Finally clause, 267
firewalls, binding types and, 16
flat files
ADO.NET support, 174, 178
defined, 178
formatting strings, 71–72
frames
defined, 119
hosting pages in, 119
full-mesh topology, 200
functional cohesion, 15
functional tests
acceptance testing, 277
API testing, 278
integration testing, 278
range testing, 278
security testing, 278
stability testing, 277
stress testing, 277, 281
unit testing, 278
function coverage (testing criteria), 282

G

GAC (Global Assembly Cache), 26, 272
GET verb (HTTP), 18
GiveFeedback event, 150
Global Assembly Cache (GAC), 26, 272
globalization and localization
comparing data, 73–74
design considerations, 69, 74
formatting text for differing cultures, 71–88
setting web page culture, 70–71
testing considerations, 279
translating applications, 72
working with time, 72
GPOs (Group Policy Objects), 228
gray box testing, 276
group membership roles
authorization and, 42, 43–44
managing, 44–45
Group Policy Objects (GPOs), 228

H

hashing
  defined, 45
  storing passwords, 47–48
high-contrast option, 110, 280
HTTP (Hypertext Transfer Protocol)
  choosing binding types, 16–18
  hyperlink support, 117
  RESTful web services and, 18
HTTPS (Hypertext Transfer Protocol Secure), 46
hub-and-spoke topology, 198–199
hyperlinks
  about, 117
  NavigateUri property, 117–118

I

IBindingList interface, 135–136
IClientMessageInspector interface, 23
ICollectionView interface, 136
IcuTest tool, 277
IDispatchMessageInspector interface, 23
IEndPointBehavior interface, 23
IEnumerable interface, 176
IExtensibleDataObject interface, 250
IIS (Internet Information Services)
  about, 8
  deadlock conflicts, 205
  hosting within, 27
  performance counters and, 288
  process identity and, 35–36
IList interface, 137
impersonation
  authentication and, 39
  example demonstrating, 37–38
  recommended usage, 36
INotifyPropertyChanged interface, 163
installation methods
  choosing, 231–232
  ClickOnce, 226–228, 231
  Windows Installer, 228–231
  XCopy, 230–231
instancing, defined, 206
integration testing, 278
internalizationalization testing, 279
Internet Information Services (IIS)
  about, 8
  deadlock conflicts, 205
  hosting within, 27
  performance counters and, 288
  process identity and, 35–36
Internet Protocol Security (IPsec), 46
interoperability with external systems
  accessing assemblies from unmanaged code, 52
  accessing COM objects, 53–54
  typical methods, 52
interoperating between Windows Forms and WPF, 92–97
IProvideCustomContentState interface, 121–122
IPsec (Internet Protocol Security), 46

J

journals
  adding items, 120–124
  defined, 120
  removing items, 120
  removing PageFunction entries, 125
JSON format, 177

K

Kerberos ticket, 43
keyboard navigation, 110
key frame-based animations, 147

L

latency, minimizing, 64
layered applications, creating, 6
layers vs. tiers, 4
lazy loading, 212–213
least privilege
  application domains, 34
  code access security, 31–34
  database design considerations, 200
  defined, 31
  partial trust concept, 32–33
linear animations, 147
LINQ to Entities, 175

LINQ to Objects, 176
LINQ to SQL
about, 175
abstracting data model, 186–187
choosing as data access strategy, 179
database mapping, 182–183
lazy loading, 212
ORM comparison, 211
SQL Profiler tool, 295
LINQ to XML, 178
load-balancing mechanisms, 61–62
load testing, 281
localization and globalization. *See* globalization and localization
localization testing, 279
locking records, 204–206
lock keyword (C#), 159
Lockwood, Lucy, 101
logical resources
declaring, 105
defined, 104
x:Key property, 105
Logman tool, 286
lookup validation, 129
loop coverage (testing criteria), 282
loosely coupled layered architecture
BizTalk Server, 10–11
defined, 2
presentation and business logic comparison, 6–7
separation of concern, 4
service-oriented architecture, 2–3
system topology, 4–6
WCF routing, 8–10

M

Machine.config file, 206
Magnifier accessibility aid, 110
maintenance and stability
designing diagnostics strategy, 285–297
designing monitoring strategy, 285–297
error handling, 132–134, 265–275
evaluating test strategies, 275–285
recommending test strategies, 275–285
Mapping Schema Language (MSL), 186
mapping to persistent storage, 182–184
MaskedTextBox control, 93
MatrixAnimationUsingPath class, 147
MdiLayout enumeration, 116–117
MDI (Multiple Document Interface) Forms
about, 90
parent form/child form model, 114–117
MediaElement class, 140
media management, 140
MediaPlayer class, 140
MemoryCache class, 189–190
memory leaks, UI-related, 164
Merge Module Project template, 248
merge modules, 248–249
merging resource dictionaries, 108
MessageContract attribute, 19
message contracts, 19
message exchange patterns, choosing, 25
MessageHeader class, 22–23
MessageHeaders class, 23–24
method granularity, 16
Microsoft Access, 178
Microsoft Message Queuing (MSMQ), 64
Microsoft.ServiceBus namespace, 65
Microsoft SQL Server
database projects, 239–240
deploying embedded databases privately, 242
editions supported, 177
synchronization considerations, 198
three-layer architecture and, 6
Microsoft Systems Center Operations Manager (SCOM), 287
modal dialog boxes, 103
modeless dialog boxes, 103
Model-View-Controller (MVC), 4, 97
Model-View-Presenter (MVP), 98
Model-View-ViewModel (MVVM), 4, 97
monitoring and diagnostics. *See* diagnostics and monitoring
MouseDown event handler, 150
.msi files, 248
MSL (Mapping Schema Language), 186
.msm files, 248
MSMQ (Microsoft Message Queuing), 64
multi-data-triggers, 146
multiple coverage (testing criteria), 282
Multiple Document Interface (MDI) Forms
about, 90
parent form/child form model, 114–117
multi-triggers, 145

MVC (Model-View-Controller), 4, 97
MVP (Model-View-Presenter), 98
MVVM (Model-View-ViewModel), 4, 97

N

namespaces. *See* specific namespaces
Narrator accessibility aid, 110
NAT (Network Address Translation), 16
navigation
  bound data in WPF, 136
  designing for different input types, 126
  implementing, 114–117
  keyboard, 110
  Navigation applications in WPF, 117–124
  PageFunction class, 124–125
  simple, 125
  structured, 125
  in Windows Forms, 90
Navigation applications
  defined, 91
  event handling, 122–124
  hosting pages in frames, 119
  hyperlink usage, 117–118
  journal usage, 120–121
  NavigationService class, 118–119
NavigationService class
  about, 118–119
  event handling, 122–124
  manipulating journals, 120–121
NavigationWindow class, 119
.NET Framework, deploying, 232–234
NetMsmqBinding binding type, 17
NetNamedPipeBinding binding type, 17, 58
NetPeerTcpBinding binding type, 17, 58
netTcpBinding binding type, 17, 18, 58
NetTcpContextBinding binding type, 58
Network Address Translation (NAT), 16
Network Load Balancing (NLB), 62
Network Service account, 35–88
NLB (Network Load Balancing), 62
n-tier architecture
  about, 4
  designing physical topology, 254–256
  determining component installation order, 256
  planning deployment, 253

O

ObjectCache class, 189
object-oriented programming, 175
Object Relational Mapping (ORM), 211–212
OData (Open Data Protocol) service, 176
offline storage, 194–198
Open Data Protocol (OData) service, 176
OpenFileDialog class, 93
OperationBehaviorAttribute class, 38
OperationContract attribute, 21
optimal processing
  cloud computing, 63
  design considerations, 56
  minimizing latency, 64
  planning for long-running processes, 56–60
  queuing, 63–64
  round-trip time, 213–214
  scaling applications, 60–63
  service bus, 65
  synchronization performance and, 200
optimistic locking, 204–206
ORM (Object Relational Mapping), 211–212

P

Package/Publish SQL tool, 241
packaging shared components, 248–249
PageFunction class, 117, 124–125
Parallel LINQ (PLINQ), 176
partial trust, 32–33, 228
passwords, storing, 47–48
path-based animations, 147
performance counters, 286, 288
performance monitoring tools
  Event Log, 285, 286–288
  Event Tracing For Windows, 286, 289
  performance counters, 286, 288
  Windows Management Instrumentation, 286, 290–291
Performance Monitor tool, 294
performance tests
  about, 281
  capacity testing, 281
  duration testing, 282
  endurance testing, 282
  performance testing, 281
  scalability testing, 281

persistent storage, mapping to, 182–184
pessimistic locking, 204–206
PIA (Primary Interop Assembly), 54
PLINQ (Parallel LINQ), 176
PointAnimationUsingPath class, 147
PollingDuplexHttpBinding class, 58
polling long-running processes, 58
POST verb (HTTP), 18
PowerShell scripts, 290
Presentation layer
  about, 4, 89
  choosing appropriate Windows technology, 90–100
  choosing between Business Logic layer and, 6–7
  choosing presentation patterns, 97–99
  data templates, 138
  designing application workflow, 113–128
  designing data presentation and input, 129–143
  designing presentation behavior, 143–155
  designing UI layout and structure, 100–113
  designing for UI responsiveness, 155–166
  determining component installation order, 256
  exception handling, 270
  sample implementations, 5, 254–255
presentation patterns, choosing, 97–99
Primary Interop Assembly (PIA), 54
PrincipalPermission attribute, 43–44
Print Management console, 36
private deployment, 243
private keys, 48
process identity, 35–88
processing. *See* background processes; optimal processing
profiling technique, 294–295
PropertyGrid control, 93
property setters, 104
property triggers, 144
protocols, binding types and, 16–18
public keys, 48
publishing databases
  from Server Explorer, 240–241
  with WCF web services, 241–242
Publish Web tool, 231
PUT verb (HTTP), 18

Q

Query Analyzer tool, 238
QueryContinueDrag event, 150
querying XML files, 178
queuing
  optimal processing and, 63–64
  scalability testing and, 282

R

race coverage (testing criteria), 282
range checking, 129
range testing, 278
RegAsm.exe (Assembly Registration Tool), 53
regression testing, 276
RegSvr32.exe tool, 234
relational databases. *See* databases
reporting, usage, 288, 291–292
Representational State Transfer (REST), 18–19, 176
resource dictionaries
  creating, 108
  defined, 108
  merging, 108
resource-level security, 35
resources
  accessing in XAML, 106
  application, 106
  dynamic, 106–107
  logical, 104–105
  static, 106–107
  storing, 108
Resources collection, 105–106
REST (Representational State Transfer), 18–19, 176
RngCryptoServiceProvider class, 58
RoboCopy tool, 231–264
roles, group membership
  authorization and, 42, 43–44
  managing, 44–45
round-robin DNS, 25, 61
round-trip time (RTT), 213–214
routing (WCF), 8–10
RSS feeds, 198
RTT (round-trip time), 213–214
RunWorkerCompleted event, 156

S

SAML (Security Assertion Markup Language) token, 43
sandboxes, defined, 34
SaveFileDialog class, 93
scalability testing, 281
scaling applications, 60–63
schemas
  designing change management strategies, 184
  services sharing, 3
  synchronizing between databases, 238
SCOM (Systems Center Operations Manager), 287, 294
screen refreshes, avoiding unnecessary, 162–163
scripts
  PowerShell, 290
  SQL, 237–238
Secure Sockets Layer (SSL), 46–47
security
  authorization, 41–44
  ClickOnce deployment, 228
  cryptography, 45–49
  database deployment conflicts, 242
  error handling considerations, 265
  impersonation and delegation, 36–41
  least privilege, 31–34, 200
  process identity, 35–88
  resource-level, 35
  role management, 44–45
  typical elements, 30
  User Account Control, 31
Security Assertion Markup Language (SAML) token, 43
security classes, 45
security testing, 278
semantic compatability, SOA service and, 3
separation of concern (SoC), 4
sequential cohesion, 15
serialization, file mapping and, 183
Server Explorer, publishing databases from, 240–241
Server Manager console
  Event Viewer snap-in, 285
  Performance Monitor tool, 294
  WMI support, 290
Server Name Indication (SNI), 47
server-side tasks. *See also* Business Logic layer
  choosing between client-side tasks and, 6–7
  data validation, 130
  designing for long-running processes, 58–59
  sorting/filtering data considerations, 163–164
ServiceBehavior attribute, 206
service bus, 65
ServiceContract attribute, 21
service granularity, 14–88
service-oriented architecture (SOA)
  designing, 2–3
  exception handling, 270
services. *See* web services
ServiceSecurityContext class, 37
Setter class, 104
Setup projects, 234, 243
shared components
  packaging, 248–249
  updating, 249–250
shared data between forms, 139
Silverlight
  PollingDuplexHttpBinding class, 58
  Presentation layer and, 4
simple navigation, 125
SNI (Server Name Indication), 47
SOAP headers, custom, 22–24
SOA (service-oriented architecture)
  designing, 2–3
  exception handling, 270
SoC (separation of concern), 4
solution deployment
  defining client strategy, 226–236
  designing update strategies, 246–253
  planning considerations, 225
  planning for databases, 237–246
  planning for n-tier applications, 253–258
solution layers, designing
  about, 1
  design service interaction, 13–30
  globalization and localization, 69–77
  interoperability with external systems, 52–56
  loosely coupled layered architecture, 2–13
  optimal processing, 56–69
  security implementation, 30–52
sorting data, 163–164
SoundPlayer class, 140
SpeedTrace Pro tool, 295
spoofing attacks, 279
SqlCeClientSyncProvider class, 198
SqlCeSyncProvider class, 196
SQL injection attacks, 279, 293
SqlMetal.exe code generation tool, 186

SQL Profiler, 295
SQL scripts, 237–238
SqlServerAdapterBuilder class, 196
SQL Server Data-tier Application template, 239
SQL Server Management Studio, 238, 239
SQL Server (Microsoft)
  database projects, 239–240
  deploying embedded databases privately, 242
  editions supported, 177
  synchronization considerations, 198
  three-layer architecture and, 6
SqlSyncProvider class, 196
SSDL (Store Schema Definition Language), 185
SSL (Secure Sockets Layer), 46–47
stability and maintenance. *See* maintenance and stability
stability testing, 277
Startup event handler, 243
stateful service exchanges, 25
stateless service exchanges, 25
statement coverage (testing criteria), 282
static resources, 106–107
storage
  database considerations, 177–178
  offline, 194–198
  password considerations, 47–48
  persistent, 182–184
  resource considerations, 108
Store Schema Definition Language (SSDL), 185
stress testing, 277, 281
String.Format() method, 71–72
strings, formatting, 71–72
structural compatability, SOA service and, 3
structured navigation, 125
Style class, 92, 103–104
styles
  about, 103
  WPF support, 103–105
SyncAdapter class, 196
Sync Framework
  designing synchronization, 198–201
  offline data storage and, 194–198
synchronization
  accessing files offline, 197
  database considerations, 198–201
  schemas between databases, 238
  thread considerations, 159
synchronous services, 24
SyncLock keyword (Visual Basic), 159
Sync Services For File Systems, 197
SyncTable class, 196
SystemBrushes class, 107
System.Diagnostics namespace, 288
System.Drawing namespace, 109
SystemFonts class, 107, 110
System.Globalization namespace, 71
System.Management.Instrumentation namespace, 286
System.Messaging namespace, 64
SystemParameters class, 107
System.Runtime.Caching namespace, 189
Systems Center Operations Manager (SCOM), 287, 294
System.Security.Cryptography namespace, 228
System.Security.Principal namespace, 37
System.ServiceModel.Channels namespace, 23
system settings, 109
System.Threading namespace, 70, 71
system topology, designing, 4–6
System.VisualBasic namespace, 178
System.Web.Caching namespace, 189
System.Web namespace, 189
System.Windows.Forms.Integration namespace, 95
System.Windows.Forms namespace, 96

T

TCP (Transfer Control Protocol), 8
Test Automation FX tool, 277
testing
  black box, 276, 277, 278
  code coverage considerations, 282
  evaluating strategies, 275
  functional, 277–279
  gray box, 276
  load, 281
  performance, 281–282
  recommending strategies, 275
  regression, 276
  schema changes, 185
  UI, 279–280
  white box, 276, 278, 282
Thread object, 158–159

threads
accessing controls, 161
creating, 158
cross-thread function calls, 161–162
deadlock conflicts and, 206
destroying, 159
starting, 158
synchronizing, 159–160
testing criteria coverage, 282
three-layer architecture
designing topology, 4–6, 254–255
exception handling, 270
tiers vs. layers, 4
tiger team, defined, 279
TimeZone class, 73
TimeZoneInfo class, 73
tokens, authorization and, 42–44
Tracefmt tool, 286
Tracelog tool, 286
Tracerpt tool, 286
TransactionScope class, 207
transactions, distributed, 207–208
Transfer Control Protocol (TCP), 8
translating applications, 72
Trigger objects, 144–147
trust boundaries, managing data integrity, 24–88
Trusted Application Deployment, 228
trusted subsystems, designing, 40–41
Type Library Importer tool, 53

U

UAC (User Account Control), 31
UDP (User Datagram Protocol), 8
UI layout and structure. *See also* Presentation layer
choosing presentation pattern, 97–99
creating resource dictionary, 108–109
custom control considerations, 111
designing for accessbility, 109–111
designing for inheritance, 101–107
dialog boxes and, 93, 103
evaluating conceptual design, 100–101
reusing visual elements, 101–107
UI responsiveness
addressing memory issues, 164
avoiding unnecessary screen refreshes, 162–163
cross-thread function calls, 161–162
design considerations, 155
offloading operations from UI thread, 156–161
sort/filter considerations, 163–164
UI tests
about, 279
accessibility testing, 280
globalization testing, 279
internalizationalization testing, 279
localization testing, 279
usability testing, 279
UnhandledException event, 273
unit testing, 278
unmanaged code, accessing assemblies from, 52–53
updating applications
checking for updates, 249
ClickOnce considerations, 247–248
design considerations, 246–247
designing web services for updates, 249–250
packaging shared components, 248–249
updating shared components, 249–250
Windows Installer support, 248
usability testing, 279
usage reporting
about, 291–292
performance counters and, 288
User Account Control (UAC), 31
User Datagram Protocol (UDP), 8
user feedback, collecting for error handling, 270–272
user interface. *See* UI layout and structure; UI responsiveness
User.IsInRole method, 44
user name/password token, 43
user navigation. *See* navigation
UTC (Coordinated Universal Time), 72

V

Validated event, 130
validating data
client-side, 130
complex validation, 130
data type validation, 129
error handling, 132–134
implementing custom rules, 132
lookup validation, 129
range checking, 129
server-side, 130
trust boundaries and, 24
WPF applications, 131

Validating event, 130
ValidationError class, 132
ValidationErrorEventArgs class, 133
ValidationRule class, 132
ValidationRules class, 131
versioning
  schema change managment strategies, 185
  for services, 25–27
Vsdbcmd.exe tool, 238

W

WAS (Windows Process Activation Service), 27
WCF data services
  about, 176–177
  choosing as data access strategy, 179
  creating, 176
WCF routing, 8–10
WCF web services
  Business Logic layer and, 4, 5
  choosing as data access strategy, 179
  deadlock conflicts, 205
  designing, 12
  designing concurrency, 206
  exception handling, 267
  hosting, 27
  logging events, 293
  performance counters and, 288
  publishing databases, 241–242
  REST support, 19
  WCF data services and, 176
web browsers, ClickOnce deployments, 227
Web.config file, 9
webHttpBinding binding type, 17
web services. *See also* specific web services
  caching, 190–191
  custom SOAP headers, 22–24
  data integrity and, 24
  design considerations, 13, 29
  designing concurrency, 206
  designing for long-running processes, 58–59
  designing for updates, 249–250
  exception handling, 267
  hosting WCF services, 27
  interoperability with external systems, 52
  message and data contracts, 19–22
  message exchange patterns, 25
  protocols and binding types, 16–18
  round-trip time, 213
  service and method granularity, 14–16
  synchronous vs. asynchronous, 24
  versioning considerations, 25–27
WF (Windows Workflow Foundation), 59–60
white box testing
  about, 276
  code coverage considerations, 282
  unit testing and, 278
"white hat" hackers, 279
White tool, 277
Windows applications, defined, 91
Windows Azure, 65
Windows Event Log
  about, 285, 286–288
  centralized logging, 292
  collecting user feedback, 270
  distributed logging, 292
  term usage for event, 286
Windows Forms
  about, 4, 90
  choosing between WPF and, 92
  ClickOnce deployments, 226
  cross-thread function calls, 161–162
  data binding, 134
  data validation, 130
  drag-and-drop operations, 151
  extending, 102
  implementing navigation, 114–117
  interoperating between WPF and, 92–97
  managing shared data, 139
  navigation in, 90
Windows Forms controls
  accessibility properties, 111
  adding to WPF applications, 95
  creating extended controls, 101
  Name property, 97
  obtaining references to, 97
  setting event handlers, 96
  setting properties in, 96
  WPF controls and, 93
WindowsFormsHost class, 95–96
WindowsFormsIntegration assembly, 95
WindowsIdentity class, 38
Windows Identity Foundation, 43
WindowsImpersonationContext class, 37

Windows Installer
checking for updates, 249
deployment strategies, 228–231
update strategies, 248
Windows.Management
.Instrumentation namespace, 290
Windows Management Instrumentation (WMI), 286, 290–291
Windows PowerShell scripts, 290
Windows Presentation Foundation. *See* WPF (Windows Presentation Foundation)
Windows Process Activation Service (WAS), 27
Windows Workflow Foundation (WF), 59–60
WMI (Windows Management Instrumentation), 286, 290–291
WPF (Windows Presentation Foundation)
about, 4, 5
application types supported, 90–91
choosing between Windows Forms and, 92
ClickOnce deployments, 226
creating application variables, 139
cross-thread function calls, 161–162
data binding, 135–137
data validation, 131
drag-and-drop operations, 151
implementing navigation, 114–117
interoperating between Windows Forms and, 92–97
managing shared data, 139
memory leaks, 164
Navigation applications, 91, 117–124
resource usage, 104–107
style usage, 103–105
WPF controls
Child property, 92
custom controls and, 111
Windows Forms controls and, 93
WSDualHttpBinding binding type, 16, 58
wsHttpBinding binding type, 16, 18

X

XAML (Extensible Application Markup Language)
accessing resources, 106
data templates, 138
<EventSetter> tag, 104
<Setter> tag, 104
WPF support, 91
XBAPs (XAML Browser Applications)
about, 32, 91
journal support, 120–121
XCopy deployment, 230–231
XmlDataProvider class, 137–138
XML files
binding WPF elements to data in, 137
mapping, 182, 183
querying, 178
XPath expressions, 137

About the Authors

TONY NORTHRUP, MCPD, MCITP, MCSE, and CISSP, is a consultant and author living in Waterford, Connecticut. Tony started C++ and assembly programming long before Microsoft Windows 1.0 was released, but he has focused on Windows development and administration for the last 18 years. He has created about 30 books and several video training courses covering Windows development, networking, and security. Among other titles, Tony is coauthor of the *MCTS Self-Paced Training Kit (Exam 70-515): Web Applications Development with Microsoft .NET Framework 4* (Microsoft Press, 2010) and *MCTS Self-Paced Training Kit (Exam 70-536): Microsoft® .NET Framework Application Development Foundation* (Microsoft Press, 2008). You can learn more about Tony by friending him on Facebook at *http://facebook.com/tony.northrup*, visiting his personal website at *http://www.northrup.org*, and reading his technical blog at *http://vistaclues.com*.

MATTHEW A. STOECKER, MCP, has written numerous books and articles on Microsoft Visual Basic, Visual C#, Windows Forms, and Windows Presentation Foundation, including *MCTS Self-Paced Training Kit (Exam 70-502), MCTS Self-Paced Training Kit (Exam 70-505),* and *MCTS Self-Paced Training Kit (Exam 70-511).*