Praise for

designing silverlight® business applications

“Jeremy’s writing style and approach to this broad subject has produced a very readable book. Beginners won’t be lost or discouraged, and there’s plenty of serious content for the experienced developer. This book will be open on your desk for reference while you’re coding.”

—Dave Campbell, SilverlightCream.com

“I strongly recommend this book to anyone seriously interested in developing applications with Silverlight. The book is written in an easy-to-read style and answers those tough questions that you would normally spend hours searching the Net for.”

—Michael Crump, Microsoft MVP, michaelcrump.net

“Jeremy explores Silverlight well beyond the basics, while maintaining crystal clarity about each subject. Should be on every Silverlight developer’s bookshelf.”

—Jesse Liberty, Developer Evangelist, Telerik

“What’s cooking in Jeremy’s kitchen? Silverlight delicacies that will benefit every Silverlight developer, regardless of experience level. And worth the price for the information on MVVM and MEF alone.”

—Jeff Prosise, Cofounder, Wintellect

“This book is a must-read for anyone writing business applications using Silverlight. Jeremy has combined his clear and precise writing style with great code examples in a book that is both instructive and enjoyable to read.”

—Beatrix Stollnitz, President, Zag Studio

“This book is a great companion for any Silverlight developer building or looking to build enterprise applications. Jeremy does a great job covering Silverlight concepts and techniques, but the application to real business scenarios based on Jeremy’s extensive experience is where the book really shines.”

—Daniel Vaughan, Cofounder Outcoder, MVP Client App Dev, danielvaughan.org

“If you’re already familiar with Silverlight, get this book to understand when and how to use proven best practices in your Silverlight line-of-business applications.”

—Jim Wooley, author, LINQ in Action
Designing Silverlight®
Business Applications
Designing Silverlight Business Applications

Jeremy Likness

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City
To my family—Gordon, Lizzie, and Doreen—
for all of the support, love, encouragement,
and patience you’ve given me to pursue my dreams.
This page intentionally left blank
Contents

Foreword .......................................................... xviii
Preface .............................................................. xx
Acknowledgments .................................................. xxx
About the Author .................................................. xxxii

Chapter 1  Silverlight ................................................. 1
  The Common Language Runtime (CLR) ......................... 1
  The Base Class Library (BCL) .................................. 2
  The Presentation Core .......................................... 3
  The Presentation Framework .................................. 3
  Communications ................................................. 4
  Data .............................................................. 4
Key Differences from the Full CLR ................................ 5
Silverlight First ..................................................... 6
WPF and Silverlight ................................................ 7
HTML5 ............................................................. 8
  HTML5 Is Not Ready for Line of Business ..................... 10
  HTML5 Won’t Keep Pace ....................................... 11
  HTML5 Isn’t Just Markup ...................................... 12
  HTML5 Isn’t Native ............................................. 12
  HTML5 Is the Perfect Technology ............................. 14
Which Client Technology Is Right for You? ........................ 14
  Use What You Know .......................................... 15
  Listen to the Customer ........................................ 15
Consider the Development Team .................................................. 16
Analyze Third-party Dependencies .............................................. 17
LOB Applications ........................................................................ 18
Summary ..................................................................................... 19

Chapter 2  Getting Started .............................................................. 21
Setting Up Your Environment ....................................................... 21
  Silverlight 5 SDK and Visual Studio Tools .................................. 22
  Expression Blend SDK ................................................................. 22
  Silverlight Toolkit ...................................................................... 24
  Open Source Projects .................................................................. 25
Hello, Silverlight ........................................................................... 26
  Creating the Silverlight Application .......................................... 27
  Class Libraries ............................................................................ 29
  Application Services ................................................................... 32
  Creating the Extension XAP ....................................................... 41
  Sharing between Silverlight and the Core Framework ............... 48
  What about Those Templates? ................................................... 53
The Standard Solution ................................................................... 55
  Web Host .................................................................................... 55
  The Silverlight Project ............................................................... 59
  Anatomy of a XAP File ............................................................. 61
Summary ..................................................................................... 64

Chapter 3  Extensible Application Markup Language (Xaml) .... 67
Markup and Class Instantiation ..................................................... 68
Dependency Objects and Properties ............................................. 70
  Dependency Properties ............................................................. 71
  Attached Properties ................................................................... 75
  Value Precedence ...................................................................... 76
Markup Extensions ........................................................................ 76
Type Converters .......................................................................... 81
Value Converters .......................................................................... 84
Styles ......................................................................................... 86
| Storyboards                                     | 91 |
| Layout                                         | 95 |
| Measure and Arrange                            | 96 |
| Canvas                                         | 96 |
| Grid                                           | 97 |
| StackPanel                                     | 101 |
| VirtualizingPanel and VirtualizingStackPanel   | 103 |
| Containers                                     | 104 |
| ContentControl                                 | 104 |
| ItemsControl                                   | 105 |
| ScrollViewer                                   | 105 |
| ViewBox                                        | 106 |
| Basic Controls                                 | 108 |
| Summary                                        | 111 |

### Chapter 4 Advanced Xaml

| Working with Text                              | 113 |
| Rich Text                                      | 114 |
| Character Spacing                              | 120 |
| Line Height                                    | 121 |
| Parts, States, and Templates                  | 123 |
| Parts                                          | 123 |
| States                                         | 127 |
| Data Templates                                 | 128 |
| Design-time Extensions                         | 132 |
| Interactivity with Behaviors and Triggers      | 141 |
| Behaviors                                      | 141 |
| Triggers                                       | 147 |
| Natural User Interface (NUI)                   | 149 |
| Resource Dictionaries and Isolating Themes     | 151 |
| Embedding and Distributing Fonts               | 155 |
| Tips for XAP Extensions                        | 157 |
| Summary                                        | 159 |
Chapter 5  The Visual State Manager  
Introduction to the VSM  
  Groups  
  States  
  Transitions  
The Visual State Manager Workflow  
Advanced Troubleshooting and Events  
Custom Visual State Managers  
The Visual State Manager in Blend  
The Visual State Aggregator  
Summary  

Chapter 6  Data-Binding  
Data-Binding Basics  
Data-Binding Debugging  
Data-Binding within Styles  
Synchronizing Lists  
Commands  
Validation  
  Validation with Exceptions  
  Validation Using Data Error Info  
  Asynchronous Validation  
  Fluent Validation  
Summary  

Chapter 7  Model-View-ViewModel (MVVM)  
UI Design Patterns  
The Model-View-ViewModel Pattern  
  The Model  
  The View  
  The View Model  
Binding the View Model to the View  
  View Model Locators  
  Controllers
### Chapter 8 The Managed Extensibility Framework (MEF)

- **Design-Time View Models** ........................................... 271
- **Custom Markup Extensions** ........................................ 272
- **View-Model-First Approach Versus View-First Approach** ........ 274
- **Lists and Data Elements** ........................................... 275
- **Summary** ............................................................. 276

#### Discovery

- **Imports** .............................................................. 281
- **Exports** .............................................................. 284
- **Parts** ................................................................. 286
- **Catalogs** .............................................................. 287
- **Containers** ............................................................ 288
- **Composition Initializer and Host** ................................ 289

#### Lifetime Management ................................................. 291

#### Extensibility ............................................................ 295

- **Recomposition** ....................................................... 297
- **Deployment Catalog** .................................................. 301
- **Discovering XAP Files** ............................................. 303
- **Offline Catalog** ....................................................... 305

#### Metadata ............................................................... 306

- **Weakly Typed Metadata** ........................................... 306
- **Strongly Typed Metadata** ........................................... 309
- **Lazy<T,TMetadata>** .................................................. 312

#### Troubleshooting ....................................................... 313

- **Understanding Stable Composition** ................................ 314
- **Import Parameters** ................................................... 315
- **The MEF Debugger** ................................................... 316

#### Jounce, an MVVM with MEF Framework .......................... 318

#### Summary ............................................................... 319

### Chapter 9 Testing ...................................................... 321

- **Why Test?** ............................................................ 322

- **Testing Eliminates Assumptions** .................................. 323
- **Testing Kills Bugs at the Source** .................................. 324
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Helps Document Code</td>
<td>324</td>
</tr>
<tr>
<td>Testing Makes Extending and Maintaining Applications Easier</td>
<td>325</td>
</tr>
<tr>
<td>Testing Improves Architecture and Design</td>
<td>326</td>
</tr>
<tr>
<td>Testing Makes Better Developers</td>
<td>326</td>
</tr>
<tr>
<td>Conclusion: You Should Test</td>
<td>327</td>
</tr>
<tr>
<td>Unit Tests</td>
<td>327</td>
</tr>
<tr>
<td>Silverlight Unit Testing Framework</td>
<td>332</td>
</tr>
<tr>
<td>Linked Classes and Shared Testing</td>
<td>338</td>
</tr>
<tr>
<td>Automated Testing</td>
<td>343</td>
</tr>
<tr>
<td>Mocking and MEF</td>
<td>345</td>
</tr>
<tr>
<td>View Model Tests</td>
<td>352</td>
</tr>
<tr>
<td>Testing Xaml</td>
<td>359</td>
</tr>
<tr>
<td>Coded UI Tests</td>
<td>364</td>
</tr>
<tr>
<td>Challenges</td>
<td>366</td>
</tr>
<tr>
<td>Automation Peers</td>
<td>367</td>
</tr>
<tr>
<td>Summary</td>
<td>371</td>
</tr>
</tbody>
</table>

Chapter 10  **Navigation**  

The Silverlight Navigation Framework  

Basics of Navigation Using the Framework  

Choosing Page-Based Navigation  

Custom Navigation  

Manual Navigation  

Containers  

Navigation Events  

Region Management  

Summary  

Chapter 11  **The Service Layer**  

Domain Data and Behaviors  

Strategies for Sharing Domain Objects between the Client and Server
## Contents

DOM Interop and ASP.NET Callbacks ........................................ 402
  The Updated To-Do List ............................................... 402
  DOM Interoperability .................................................. 404
  Another Example: Callbacks ........................................... 407

Communication ................................................................. 412
  Representational State Transfer (REST) ................................ 412
  Plain Old XML (POX) and JavaScript Object Notation (JSON) .... 423
  Windows Communication Foundation (WCF) .......................... 424
  WCF RIA Services ....................................................... 431
  Local Messages .......................................................... 435
  Sockets ................................................................... 436

Mapping and Transformation ................................................ 443
  Mapping Versus Exposing the Model .................................. 443
  Dynamic Types ........................................................... 448
  The Custom Type Provider .............................................. 450

Asynchronous Techniques .................................................... 452
  Events Versus the Asynchronous Programming Model (APM) ..... 452
  Lambda Expressions and Method Chaining .......................... 455
  Action and Callbacks .................................................... 456
  Reactive Extensions (Rx) ............................................... 459
  IWorkflow ................................................................. 460
  Tasks and await .......................................................... 463

Summary ................................................................. 465

Chapter 12  Persistence and State Management ............................ 467

The To-Do List Application ......................................................... 468
  WCF RIA Services ....................................................... 468
  Filters and Sorts .......................................................... 471
  Navigation Parameters ................................................... 472

Persisting Preferences with Settings ....................................... 473

Folders and Files in Isolated Storage ....................................... 476
  Managing Isolated Storage Access and Quotas ....................... 478
  Accessing Folders and Files ............................................ 482
  Finding Isolated Storage ................................................ 484
**Contents**

*Iterating the File System* ........................................ 485
*Signing Files* ......................................................... 491
*Encrypting Files in Silverlight* .................................. 494
Sterling in Silverlight .................................................. 498
Summary ................................................................. 514

Chapter 13  **Out of Browser Applications** ......................... **515**
*Getting Started* ....................................................... 516
  *Checking for Updates* ............................................. 523
*Elevated Trust* .......................................................... 526
  *Application Signing* ............................................... 528
  *File System Access* ............................................... 530
  *Toast Notifications* .............................................. 531
  *Child Windows* .................................................... 535
  *COM Interop and Script Host* ................................... 540
  *Native Silverlight Extensions* ................................... 546
  *p/Invoke* ................................................................... 547
  *Elevated Trust in the Browser* ................................... 552
*Distribution and Installation* ....................................... 553
  *Installation* .......................................................... 553
  *Execution* .............................................................. 554
  *Uninstalling Your OOB Application* ............................. 555
Summary ................................................................. 556

Chapter 14  **Line of Business Features** ............................ **557**
*Designer/Developer Workflow* ...................................... 559
*Printing* ................................................................... 561
*Localization* ............................................................. 571
*Modularity and Extensibility* ......................................... 576
*Scalability* ............................................................... 578
  *Extremely Large Data Sets* ......................................... 578
  *Concurrency* ........................................................... 590
  *Synchronization* ....................................................... 593
*Proof of Concept* ....................................................... 594
Summary ................................................................. 596
Chapter 15 Debugging and Performance Optimization 597

Debugging Silverlight Applications 598
  Debug Symbols 600
  Debugging Tips 602
  Debugging Applications Already Launched 608
  WinDbg 611

Logging and Tracing 616
  Client Logging and Tracing 617
  WCF Tracing 619

Profiling Applications with Visual Studio 622
Fiddler 625
Silverlight Spy 626
Summary 627

Glossary 629
Index 653
When I was first approached to write the foreword for Jeremy’s book, I thought of many snarky things to say…but at the end of the day I think it is most accurate to say that I was honored. As someone who has talked, written, and taught a lot about Silverlight, I’ve been watching Jeremy’s love of the entire XAML stack.

As he and I share a love of both XAML and a geographic area, it was hard not to run into his presentations on a variety of subjects germane to this book. He has that key pairing of passion for the subject and technical prowess to see the big picture of best practices. These skills are obvious to anyone who has seen him talk or used his excellent Silverlight open source community contributions. It seems like a natural next step for him to tackle a book. And completely unsurprisingly, he didn’t take a light subject but tackled the difficult problem of architecture and Silverlight. And I suspect you wouldn’t be surprised to know he handled it with much aplomb.

Silverlight is a natural solution for the enterprise space as it combines a web-delivered deployment story and marries it with a rich-client, easy-to-develop solution. The challenge is that creating large, scalable, and maintainable applications using Silverlight requires forethought. While it’s expected that any Silverlight book will cover crucial topics like XAML, data binding, and such, to build enterprise applications, other skills are required. These include architectural patterns (for example, MVVM), composition patterns (for example, inversion of control), testability, and the difficult problem of data. Jeremy tackles these topics by ensuring that you, the
reader, know how important it is to think of these topics as more than afterthoughts, but the core of designing mission-critical, line-of-business applications. I, for one, think he has succeeded.

It was 2006 when Silverlight made its debut with the confusing acronym WPF/E, for Windows Presentation Foundation/Everywhere. Like many other developers at the time, my first reaction was to dismiss the technology as yet another plug-in. Silverlight struggled in its early days to prove its identity as something more powerful than just another media player. Through an extremely rapid release cycle, Version 3.0 became available in 2009 and .NET developers began to take notice.

At the time I was working on an extremely large web-based enterprise application that provided a desktop-like experience using HTML and JavaScript. Our developers had to become experts not only in multiple languages—switching between XML, JavaScript, C#, and CSS—but also with the nuances of the multiple browser implementations that existed at the time. We were spending more time testing for cross-browser compatibility and hacking JavaScript and CSS than we were building core business functionality.

I decided to take Silverlight for a spin and started with a simple proof of concept. I found that because Silverlight used a smaller version of the core .NET Framework and could be written in my language of choice (C#), it was very easy to learn. I was extremely impressed with the Xaml layout engine and was able to produce a prototype quickly to show our CEO. He gave me the green light to use Silverlight in our project and we started right away.

It was soon apparent to the team that Silverlight was an easier way to build web-based applications. We were able to provide an amazing, rich
user experience with complex features in a very short span of time and were able to provide a consistent application across multiple browsers and platforms. I estimated that our team was able to produce software about four times faster using Silverlight than traditional web-based technologies.

Since then I’ve used Silverlight to build enterprise applications for a variety of customers across multiple verticals. From a backend server health monitoring system to online cable grid listings, a SharePoint-based risk management system to a large and complex social media monitoring application, Silverlight has made writing usable and scalable software an amazing and rewarding experience.

I’ve spent the past decade focused on highly scalable web-based enterprise applications, with the last several years working almost exclusively with Silverlight. This experience has helped me design, code, test, debug, and implement Line of Business (LOB) solutions for dozens of customers using Silverlight. I’ve done my best to take those various experiences and use them to provide you with powerful and reusable solutions in this book.

**What This Book Is About**

The purpose of this book is to explain how to write Silverlight applications that target enterprises and commercial customers as opposed to end users and consumers. This is the heart of what LOB is about. I intend to focus on advanced features of Silverlight that directly relate to the specific business problems that the framework is capable of solving in that context. I will concentrate on the areas and themes that have come up time and time again from my own experience.

This book is not an introduction to Silverlight. I’m assuming you either have prior Silverlight experience or have experience building LOB applications using other client technologies such as Windows Presentation Foundation or ASP.NET. I’m also assuming you are at least familiar with the concept of design patterns and the notion of decoupled code. Both of these ideas have been core to the success of the applications I’ve helped build and will be used as the foundations for the concepts presented in this book.
Whether you’re a Silverlight LOB developer looking to improve an existing application, or an experienced client technologies developer who is transitioning to Silverlight for the first time, this book will give you the guidance and proven patterns and practices you’ll need to build scalable, maintainable, and highly professional applications that run equally well on multiple platforms and browsers.

This version of the book specifically addresses Silverlight 5 using Visual Studio 2010 SP1. Silverlight runs perfectly well as a standalone client technology with no dependencies on a specific backend. It can be hosted from any type of web server, including a Unix-based Apache server. Examples in this book that involve the server are assumed to be running .NET Framework 4.0, where applicable. This book does not address the Windows Phone version of Silverlight, although many of the concepts are shared between the different versions of the runtime.

**How to Use This Book**

The aim of this book is to enable you to write Silverlight applications that target the enterprise. Each chapter is designed to help you discover what features are available in that area of the framework and how they are applied in a LOB setting. Code examples are provided that demonstrate the features and best practices for programming them using C#. Although different chapters may relate to various parts of a comprehensive project, the individual code samples are designed to stand on their own.

Each chapter is similarly structured. The chapters begin with an introduction to a topic and an inventory of the capabilities that topic provides. This is followed by relevant case studies from existing projects and includes code samples to demonstrate its application. The code samples are explained in detail and the topic is summarized to highlight the specific information that is most important for you to consider.

I suggest you start by reading the book from start to finish, regardless of your existing situation. Inexperienced Silverlight developers will find that their understanding grows as they read each chapter and concepts are
introduced, reinforced, and tied together. Experienced Silverlight developers will gain insights into areas they might not have considered or had to deal with in the past, or simply didn’t factor into their software lifecycles. After you’ve read the book in its entirety, you will then be able to keep it as a reference guide and refer to specific chapters any time you require clarification about a particular topic.

About the Author

My first computer program was written in BASIC on a TI-99/4A. From there I programmed assembly language for the Commodore 64, learned C and C++ on Unix-based systems, and later wrote supply chain management software on the midrange AS/400 computer (now known as iSeries). For the past 15 years, my primary focus has been developing scalable, highly concurrent web-based enterprise applications.

I started my work with Silverlight right before the 3.0 release. At the time, I led a team of 12 developers working on an ASP.NET mobile device management platform that relied heavily on AJAX to provide a desktop-like user experience. When it was evident that the team was spending more time learning various web technologies, such as CSS and JavaScript, and testing the application on multiple browsers and platforms than focusing on the core business value, I began researching alternative solutions and determined that Silverlight was the key our team was looking for.

Since that transition I have worked almost exclusively on Silverlight applications in the enterprise. In addition to the mobile device management software, I helped build the health monitoring system for the backend data centers that provided video streams (live and on demand) during the 2010 Vancouver Winter Olympics. I worked on a major social media analytics project that used Silverlight to present data that was mined from social networks and analyzed to provide brand sentiment. I worked with a team that built a slate-based sales interface for field agents to close sales and integrate with their point of sale system. I was on the team that produced the
Silverlight version of a major e-book reading platform designed for accessibility and customized to provide interactive experiences and audio for children.

All of this work has been with the company Wintellect, founded by well-known .NET luminaries Jeffrey Richter, Jeff Prosise, and John Robbins. All three have produced countless books about the Microsoft stack, .NET Framework, and Core Language Runtime (CLR). They have trained thousands of Microsoft employees (some teams at Microsoft are required to take their courses as a prerequisite to working on their projects) and contributed to the runtime itself by writing and designing portions of the framework. The company has provided me with unique access to industry leaders and architects and their best practices and solutions for creating successful enterprise applications.

I am a certified Microsoft Silverlight developer (MCTS) and was recognized as a Microsoft Most Valuable Professional (MVP) for Silverlight in July of 2010. This was due mostly to my efforts to blog, tweet, and speak about Silverlight at various user group meetings and conferences around the country. It is this depth of experience working with Silverlight, understanding how to build server and web-based software, and migrating existing applications to Silverlight that has provided me with valuable insights into how Silverlight works within the enterprise.

### About Silverlight

Silverlight is a unique platform that has transformed over the years. Originally intended to be a vehicle to provide Xaml technologies everywhere—from Windows to OS X, Linux, and even smartphones—it has matured to become a choice platform for writing LOB applications. Microsoft summarizes Silverlight on their website with this statement:

"Silverlight is a powerful development platform for creating engaging, interactive user experiences for Web, desktop, and mobile applications when online or offline. (Source: http://www.silverlight.net)"
That simple statement covers a lot of ground. Silverlight is a development platform that should be familiar to most shops that already produce .NET code, regardless of whether that code is written in C#, Visual Basic, IronPython, or any other language that can sit on top of the Common Language Runtime (CLR). The layout and rendering engine, along with animations, provide the engaging and interactive experience, whereas the reach of the platform allows for the web, desktop, and mobile targets (although right now the mobile is mostly limited to Windows Phone 7).

An important aspect of Silverlight that sets it apart from other web plug-ins is the ability to run offline, or with an Out of Browser (OOB) experience, even when the user is disconnected from the Internet. This experience is consistent between Windows and OS X (Macintosh) platforms when a specific subset of features is used. The ability to easily deliver this experience over the Web is why many customers have chosen to adopt Silverlight.

If you asked me to summarize Silverlight in a single statement, it would be this:

_Silverlight is a cross-browser, cross-platform, Xaml-based runtime that allows developers to leverage the power of the Visual Studio and Expression Blend tools and a choice of .NET languages to provide rich, interactive experiences that integrate easily with existing data and services._

This statement captures the essence of what I believe Silverlight offers. It is familiar to C# and VB.NET developers because it uses familiar tools and is also accessible to those who prefer F#, IronPython, and other languages. Although Silverlight can be developed using completely free development tools, the addition of the licensed Blend product provides a design experience that can be used to create incredible user interfaces.

Another major benefit of Silverlight as a web-based tool is the ability to integrate directly with existing web services and objects. It enables you to share data and business logic between type definitions that exist on both the client and the server. Silverlight readily consumes WCF-based services as well as REST endpoints and can serialize and deserialize data in binary, XML, ATOM, RSS, and JSON formats. The ability to seamlessly integrate with existing services, data structures, and even class behaviors makes it ideal for LOB applications.
Brief History

Silverlight was first released in 2007. The first version was Xaml based and exposed the programming interface through an API that relied on JavaScript in the browser. The layout engine used HTML controls and DOM interoperability with the Silverlight runtime to function. Silverlight also provided media features such as video and audio playback. This led many developers to believe it was a direct challenge to Flash and designed as “another movie player plug-in.”

The second version, initially 1.1 and renamed to 2.0, was released just a year later in 2008. This version really opened the door for serious developers because it provided a full runtime in the form of a stripped-down version of .NET Framework 3.0. This allowed programs for Silverlight to be compiled in any language supported by .NET. The version also provided a lightweight Base Class Library (BCL) that included canned controls, network APIs, and even access to Language Integrated Query, or LINQ.

Another well-known feature added at this time came from the project code-named Seadragon that was released as DeepZoom. The technology provides a way to take extremely large images (or collections of images) and scale them to allow the user to pan and zoom without having to wait for the entire image to download. It pixelates the zoomed image and slowly increases the resolution as more information is available because the image is turned into slices and tiles of varying resolutions. Many companies used this feature to provide collages of images that could be easily browsed but then zoomed to full resolution. You can view an example of DeepZoom at http://www.wintellect.com/silverlight/deepzoom/.

In November of 2008, the Silverlight Toolkit was originally released. The toolkit is designed to provide the developer community with new components and functionality for product development. It is released out of band with the runtime and includes features such as drag-and-drop behaviors, enhanced validation, and special input controls. Some of the toolkit controls, such as the popular Viewbox for automatically sizing content, were eventually migrated into the core runtime.

Silverlight 3.0 was released during the summer in 2009. It added quite a few controls to the toolbox that developers could use to build their
applications, including the popular DataGrid, which is capable of automatically generating columns and rows from a data source. It provided a rich navigation framework that allows for deep-linking into the application using URLs. The media support was extended significantly.

The most significant features, however, were the catalyst for heavy adoption of Silverlight in the enterprise. The ability to provide Silverlight as an OOB application was extremely significant because it allowed an offline experience for the first time. Users could now use the application even when disconnected from the Internet. This was unique because it enabled companies to provide an application that could run disconnected, was delivered easily over the Internet, yet would run on Mac OS X as well as Windows-based machines.

In addition, a local communications API was provided to allow Silverlight applications running on the same machine to message each other. An update mechanism was introduced that allows applications to check the server from which they were installed for updates and then to update automatically when one is available. For the first time, Silverlight could provide a full application experience that could be delivered and managed from a central location with an easy installation process over the Web.

With heavier adoption in the enterprise, Silverlight users began to complain about LOB features that were missing. These included the following:

- Integration of a web camera and microphone for video and audio capture
- Printing support
- Advanced features users of the Windows Presentation Foundation (WPF) frequently utilized

This all changed with the release of Version 4.0 in 2010, which closed a critical gap for LOB applications by providing raster-based printing support directly from the Silverlight application. The user interface improved dramatically with the ability to send toasts or notifications that could be displayed by the host operating system (whether Windows, OS X, or others). Applications could take advantage of online content by hosting web controls capable of rendering HTML. Silverlight was extended to provide
access to COM components on Windows machines. The Managed Extensibility Framework (MEF) was integrated with the runtime to provide discovery, lifetime management, extensibility, and metadata services. This release represented huge strides forward.

While Silverlight continued to gain momentum in the enterprise, it also encountered its first major hurdles between 2009 and 2010. Silverlight was originally thought of as a development platform that would eventually run everywhere, but the iPhone presented an obstacle too steep to overcome. Apple simply would not allow the Silverlight runtime to exist on their phone. At the same time, the iPad was released and became a literal overnight success. Using the same operating system as the phones, it, too, would not allow the Silverlight plug-in. This presented a tangible barrier and caused executives to begin to question what technology made more sense and had farther reach. Many heads turned to the rapidly emerging HTML5 specification as the holy grail of write once, run anywhere.

The result of this was a shift in focus from the idea that Silverlight would run anywhere to a revised notion that it was ideal for rich desktop-based applications (Windows and OS X) but not something that would end up on all smartphones. One smartphone was released that embraced Silverlight out of the box: the Windows Phone 7 series. Using an enhanced 3.0 version of Silverlight with extensions specific to the phone, it introduced a whole new wave of developers to the platform while existing Silverlight developers suddenly found themselves with unprecedented access to the mobile market. The current 7.1 version of the SDK supports an enhanced Silverlight 4.0 runtime on the Windows Phone 7.5 OS that is code-named Mango.

The general development community also reacted strongly when news of the Windows 8 operating system included demonstrations of a programming experience based on HTML5 and JavaScript. None of the announcements spoke to Silverlight or WPF. This caused concern over whether support for the two popular Xaml-based technologies would even exist in the next version. Where would Silverlight fit into this scenario?

Fortunately, Version 5.0 was announced in late 2010 and the beta released in May of 2011. The release of the version at the end of 2011
demonstrates Microsoft’s commitment to Silverlight as a platform, and the features show how it continues to power LOB experiences. This significant release managed to pack some of the most important features to date, including the following:

- Various text enhancements allowing for more control over text layout and flow, character spacing, pixel-snapping, and ClearType technology
- Performance enhancements ranging from an improved networking stack to the introduction of a composition thread that allows animations to run in the background without blocking (or being blocked by) the primary UI thread
- Click counting and text type-ahead for improved user interaction and use of mouse, touch, and the keyboard
- Vector (PostScript) based printing for improved fidelity and improved performance with printing
- Enhanced data-binding that has more parity with WPF and enables data-binding debugging
- Custom markup extensions to provide more control over Xaml
- Improved trust scenarios, expanded access to the file system, and child windows for OOB applications
- The introduction of Platform Invoke, or p/Invoke (a Windows-specific feature), to access unmanaged code directly from Silverlight applications
- A powerful XNA-based 3D engine that renders vertices with texture-mapped surfaces, shading, lighting, backface culling, and bump maps as well as provides dynamic camera angles

These features are likely to drive increased Silverlight adoption by removing previous barriers to entry and providing a new set of tools for LOB developers to quickly and efficiently build enterprise applications.
No book is possible without a team of passionate individuals who work together to deliver a quality product. I am grateful for my superhuman editor, Joan Murray, who helped pull so many things together in the time we had and continuously provided her support and guidance throughout the process. Chris Zahn provided me with excellent technical support. I had the pleasure of collaborating with an incredible team of development editors, Christopher Cleveland and Eleanor Bru, who have done an amazing job of helping me transform my initial thoughts and ramblings into a polished, professional piece of work, along with the copyedit team of Anne Goebel and Bart Reed.

I also had the privilege of working with an amazing team of technical editors. Thank you, Dave Baskin and John Garland, for the countless hours you put into pouring over my initial drafts. Dave and John not only gave me valuable insights into better ways to organize and introduce concepts, but also closely inspected the veracity of the text, provided valuable references and corrections when needed, and made many recommendations and suggestions that helped this book better convey what it means to build quality Silverlight applications.

Many thanks to Wintellect founders Jeff Prosise, Jeffrey Richter, John Robbins, and Lewis Frazer for their inspiration, encouragement, and incredible advice based on their own experience producing some of the best known books in the .NET world as well as building the top training and consulting practice in the field. I appreciate the support of my colleagues
Todd Fine and Steve Porter, who understood the effort this would take and worked with me to balance those demands with my daily responsibilities.

A special note goes to Charles, Tom, Dan, Adam, and John in Portland, Oregon for their unique contributions to Chapter 13.

Thanks to my wife and daughter who provided their constant support and encouragement. I appreciate their patience and understanding whenever I asked to retreat into my office to disappear for a few thousand words.

I am thankful for the global Silverlight community of individuals who blog, tweet, speak, and write about this amazing platform that has transformed so many businesses. It was this community that encouraged me to write this book.

Finally, thank you! I appreciate my readers.
About the Author

Jeremy Likness is a multiyear Microsoft MVP for Silverlight. A Senior Consultant and Technical Project Manager for Wintellect with 15 years of experience developing enterprise applications, he has worked with software in multiple verticals, ranging from insurance, health and wellness, supply chain management, and mobility. His primary focus for the past decade has been building highly scalable web-based solutions using the Microsoft technology stack. Jeremy has been building enterprise Line of Business applications with Silverlight since version 2.0.

Prior to Wintellect, Jeremy was Director of Information Technology and served as development manager and architect for AirWatch, LLC, where he helped the company grow and solidify its position as one of the leading wireless technology solution providers in the United States by managing the development of their product portfolio, which includes public hot-spot solutions and a management console for enterprise-grade wireless networks, mobile devices, and their consumers. A fluent Spanish speaker, Jeremy served as Director of Information Technology for Hispanicare, where he architected a multilingual content management system for the company’s Hispanic-focused online diet program. Jeremy accepted his role there after serving as Development Manager for Manhattan Associates, a software company that provides supply chain management solutions.
Model-View-ViewModel (MVVM)

Model-View-ViewModel is an elegant way to simplify Silverlight development, making it fast and easy; unfortunately, many developers mistakenly believe it is an incredibly complex pattern. A discrepancy exists because developers can’t seem to agree on what MVVM is, often confusing frameworks that utilize MVVM with the pattern itself. Add to the mix over-engineered and overly complex applications, and you have the ingredients for a controversial soup of opinions about MVVM.

In my experience, the proper use of MVVM makes it easier to build applications, especially when you have larger teams or separate teams of designers and developers. The ability to incorporate unit tests also helps reduce the rate of customer-initiated incidents because bugs are caught earlier in the process. Unit tests make it easier to extend and refactor applications, and the MVVM pattern itself allows for what I call refactoring isolation, or the ability to make modifications to areas of the application without having to visit and update every module as a side effect of the change.

In this chapter, you learn about design patterns and why they are important. I share with you a brief history of patterns, what their authors intended, and how this led to the creation of the MVVM pattern. Sections
cover each element of the MVVM triad, followed by some of the key features and benefits that MVVM provides. MVVM itself is not a framework, although there are many frameworks that provide implementations of the pattern; MVVM is a UI design pattern.

**UI Design Patterns**

In medieval times, guilds were groups of individuals with common goals. There were different types of guilds, including the craft guilds, whose members were artisans of specific occupations such as baking and stone-cutting. Guilds might have to provide a stamp of approval for items before they were sold to the common market in order to maintain the quality and integrity of the product. More importantly, guilds would identify the master craftsmen or experts who would then take on apprentices. An apprentice wouldn’t have to figure out everything on his own; instead his master would share the “tricks of the trade” and provide the best practices to get the job done.

In the guild of software development, the master craftsmen use design patterns as their tools of choice. These are simply repeatable solutions for recurring problems that have evolved over time. Sometimes the solutions were discovered by specific individuals, but more often than not, patterns were established independently to solve similar problems and emerged as a common solution when groups of developers shared their ideas. Although every software application is unique, it is often composed of distinct sets of challenges that have already been solved.

There’s a good chance you’ve worked with established patterns even if you didn’t call them by name. Have you created a method that uses an existing object as the template to create a new object? That’s called the *prototype* pattern. Have you written a data-access layer that uses a generic interface with load, save, and delete methods and then maps those actions to more specific APIs exposed by ADO.NET, LINQ-to-SQL, or some other data provider? That’s called an *adapter*. Have you ever used a *foreach* loop in C#? That’s an *iterator*.

As you can see, there are many existing patterns to solve common problems. Learning patterns isn’t an exercise in hypothetical programming and
doesn’t automatically make you a software architect, but it does provide a vocabulary you can use to construct software applications. Just like learning more words helps us communicate better, learning patterns will make it easier to solve existing problems by tapping into proven solutions that have already been tested in the field. This is extremely important in LOB applications when the development teams are larger and members come and go. Ramping up should involve focusing on the business domain more than the general software itself.

Some user interface (UI) design patterns have evolved to solve the problem of maintaining the presentation layer of your application independently of the underlying business logic, services, and data. Some of the problems being solved include the following:

- **Fluidity of the user interface**—Often there can be significant changes to look, feel, and interaction over time. A well-defined and properly implemented UI design pattern can help insulate those changes to minimize impact on the core business logic and data concerns.

- **Parallel development and design**—Often the design team is separate from the development team, with different skillsets and involving multiple designers. UI design patterns can maximize the efficiency of this workflow by providing the separation necessary to allow the developers and designers to work in parallel with minimal conflicts.

- **Decoupling of presentation logic**—There are common patterns in the presentation layer, such as providing a list of items and allowing the user to select a single item, that can be solved in multiple ways (combo box, grid, list box, and so on). UI design patterns help decouple data elements from the presentation implementation so the core functionality can remain the same regardless of how the pattern is presented.

- **View-logic testing**—Complex view logic is often the domain of developers. Testing the logic is made easier by not requiring a full-blown UI. An example is dependent or cascading lists: Selecting an
item in the first list determines the content of the second list. Ideally, you should be able to implement this behavior and test it without having to draw a corresponding combo box control and process click events.

The problem of effectively developing, testing, and integrating the presentation layer has been around since the earliest days of computer programming. The first UI design patterns can be traced back to the late 1970s when a scientist named Trygve Reenskaug visited Xerox’s Palo Alto Research Center (PARC) and wrote a series of reports in an effort to “simplify the problem of users controlling a large and complex data set.” (http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf).

The solution he described eventually came to be referred to as Model-View-Controller, or MVC for short. The same pattern is popular and used today for web-based applications (Microsoft even named their latest framework for web development after the pattern). It defined a separation of concerns between the visual element on the display, the logic to interact with that element, and the rest of the application that represents its perception of the real world through a domain model.

The initial title of the first paper was actually “thing-model-view-editor,” because out of the model comes a thing that needs to be viewed and edited. The essence of the pattern was a separation that looked something like Figure 7.1.

The model represents everything in the system that doesn’t involve user input or presenting any type of view to the user. The controller works with the model and coordinates views as well as processes input from the user. The idea is that the view doesn’t ever interact directly with input, but simply receives commands from the controller. The view can also observe the model and present information from the model.

The key to the pattern lies in his description of the controller: “a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands” (http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf).
To summarize, a few simple rules drive the MVC pattern:

- The model is completely ignorant of the UI (view/controller).
- The controller handles the user input.
- The view handles the output by inspecting the model.
- Together, the view and controller provide a reusable control with encapsulated behavior, as shown in Figure 7.1.

Although the MVC pattern achieved a certain level of separation, it does have some limitations. In the original pattern, the view would display values based on observations of the model. The model might represent data in the form of integers. The view, however, might display a bubble chart with various diameters based on the value. In that case, something has to be responsible for taking an integer value and determining what diameter it maps to. The model really shouldn’t be concerned with that task because it is supposed to be ignorant of the UI and the concept of a circle with a specific diameter.

The solution is to introduce a special type of model referred to as the presentation model. This is part of the application tasked specifically with translating information of the model and formatting it into the correct values for the view. It is the presentation model that can take an integer and
map it to a diameter. It is still independent of the view in the sense that it
does not know how to draw a circle, but it forms a sort of buffer between
the pure model and the pure view to provide what is referred to as view
logic or presentation logic.

The pattern that uses the presentation model is called the Model-View-
Presenter (MVP) pattern. The term was popularized in a paper published
in 1996 by Mike Potel (http://www.wildcrest.com/Potel/Portfolio/
mvp.pdf). In this pattern, the dependencies and flow were more formal, as
visualized in Figure 7.2.

![Model-View-Presenter](image)

**Figure 7.2:** The Model-View-Presenter (MVP) pattern

The MVP pattern introduced a few significant changes. First, the inputs
(referred to as *interactions*) are handled by the view. The view observes the
model directly to display information, but when the user provides inputs,
those inputs are raised as events to the presenter. The presenter then
processes those events and sends them as commands to the model.

Although there are several variations of both the MVC and MVP pat-
terns, they are widely recognized patterns that have been used for decades
to separate presentation logic from the internal business logic and data that
drives applications. It is these patterns that also laid the foundation for the
pattern you will learn about in this chapter, the Model-View-ViewModel
pattern. This pattern was created and popularized for Windows Presenta-
tion Foundation (WPF) but was quickly carried over to Silverlight.
The Model-View-ViewModel Pattern

In 2005, a developer named John Gossman working on WPF—which at the
time was code-named Avalon—published a blog post that would ultimately
introduce the MVVM pattern to the world (http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx). In his post he described
“a variation of Model/View/Controller (MVC) that is tailored for modern
UI development platforms where the View is the responsibility of a
designer rather than a classic developer.”

The introduction to his post provides valuable insight into one of the
original motivations for the pattern: the designer/developer workflow. His
post further explains that the view is defined declaratively (a reference to
Xaml) and is responsible for inputs, keyboard shortcuts, visual elements,
and more. The view model is responsible for tasks that are too specific for
the general model to handle (such as complex UI operations), for main-
taining the view state, and for projecting the model to the view, especially
when the model contains data types that won’t map directly to controls.

Although MVVM is most often compared to MVC, it actually makes
more sense to think of it as a specialized flavor of MVP that uses data-
binding and the Visual State Manager (VSM). Instead of raising events, the
view drives the view model through data-binding—whether it is by updat-
ing a value that in turn synchronizes to a property on the view model, or by
mapping an event to a command that fires on the view model. The input
aspects of the controller have been absorbed into the view, while the presen-
tation logic is distributed between the view model and the view. Presen-
tation logic in the view takes the form of behaviors, triggers, visual states,
and value converters.

Like other patterns, MVVM is a solution to common problems. When
implemented correctly, it should make the job of building a Silverlight
application easier. Unfortunately, the pattern can be abused and end up
slowing down projects and making them more complex than necessary.
I’ve built dozens of large enterprise Silverlight applications. Not all
applications followed the MVVM pattern, and in many cases my company
was called in to rescue a failing project by refactoring the application to use
MVVM. Although the pattern is now over six years old, there are many lingering misconceptions. Table 7.1 lists some of these misconceptions and the truth that addresses them.

**TABLE 7.1: Common MVVM Misconceptions**

<table>
<thead>
<tr>
<th>Misconception</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVVM is extremely complex.</td>
<td>MVVM can be incredibly simple when implemented correctly.</td>
</tr>
<tr>
<td>Code-behind isn’t allowed in MVVM.</td>
<td>Code-behind is simply an extension of the declarative Xaml for the view. The view is responsible for managing the user interface, including user inputs, and there is no reason the code-behind cannot deal with events and interactions.</td>
</tr>
<tr>
<td>MVVM is hard to implement.</td>
<td>Many frameworks exist that can enable you to have your project up and running in minutes. I recorded a video to demonstrate building a feed reader from scratch using MVVM in just 30 minutes (<a href="http://vimeo.com/17926625">http://vimeo.com/17926625</a>).</td>
</tr>
<tr>
<td>MVVM eliminates the need for value converters.</td>
<td>Value converters are reusable, testable pieces of code that can map data from the model to the view, and there is no reason to eliminate them when using the MVVM pattern.</td>
</tr>
<tr>
<td>MVVM reduces the performance of the application.</td>
<td>The improper implementation of any pattern can create performance issues. Proper use of MVVM facilitates unit tests that can help tweak and improve performance.</td>
</tr>
<tr>
<td>MVVM is only good for very large projects.</td>
<td>A good MVVM framework coupled with solid understanding is just as suitable for small projects as it is big ones.</td>
</tr>
<tr>
<td>MVVM is about commands and messaging systems.</td>
<td>MVVM simply specifies the responsibilities of various modules within the code. Commands, messaging frameworks, and other constructs are just helpers and building blocks.</td>
</tr>
<tr>
<td>MVVM is hard to understand.</td>
<td>MVVM is no more difficult to understand than data-binding and the Visual State Manager, because it really is just a pattern that describes how best to use these features.</td>
</tr>
</tbody>
</table>

In the rest of this chapter you’ll learn about the various parts of MVVM and how to apply it. If you’ve read the previous chapters in this book and followed the examples, you already have an understanding of MVVM because you’ve created specific classes that implement property-change
notification to facilitate data-binding. Those classes can actually be thought of as your view models.

Contrary to the misconceptions about MVVM, there are many advantages the pattern provides above and beyond the separation of design from development. In my experience, these are the top ten benefits you may receive by using MVVM in your applications:

1. **A clean separation of concerns (decoupling)**—MVVM follows best practices for software architecture.
2. **Designer/developer workflow**—MVVM enables parallel development and design by multiple team members working on the same project.
3. **Unit testing**—You will learn more about testing in Chapter 9, “Testing.”
4. **Use of data-binding**—MVVM takes direct advantage of the rich and powerful data-binding system in Silverlight.
5. **Improved code reuse**—View models can be used to power multiple views, and various helpers and scaffolding can be reused throughout your project and across various products in your organization.
6. **Modularity**—MVVM encourages a modular design that makes it easy to modify parts of the application independently of each other.
7. **Refactoring containment**—Through the clean separation of concerns, MVVM minimizes the impact to other areas of the application from refactoring.
8. **Extensibility**—A well-designed MVVM framework makes it easy to extend the application by adding new screens, modules, and plug-ins.
9. **Tools support**—Various tools, such as Expression Blend and the designer, are built in to Visual Studio that can take direct advantage of MVVM.
10. **Pattern vocabulary**

The final item, pattern vocabulary, requires some additional explanation. When you are learning how to read, there is a strong correlation
between the size of your vocabulary and your ability to comprehend what you are reading. This should not be surprising because vocabulary provides the building blocks for the text you are trying to comprehend, and not understanding those blocks can lead to confusing conclusions and misinterpretations of the text. Although there is a strong correlation, vocabulary certainly doesn’t guarantee comprehension because you must be able to piece the words together and derive their meaning as a whole.

Developing software is an exercise that also involves a vocabulary. You start with the vocabulary of the language you are developing in. Programs have their own syntax and grammar, and comprehension relies on your ability to interpret the keywords correctly and understand them in context. Patterns provide a higher level vocabulary that can describe entire subroutines and components within the system. As with vocabulary, knowing a pattern isn’t the same thing as comprehending how it best fits into a software application (or whether it belongs at all).

The more you are able to understand and integrate patterns, the more you will be able to build your vocabulary and better comprehend complex software systems. Although I am not aware of any scientific studies that definitively support this conclusion, it is based on years of working in the software industry with individuals of varying degrees of skill and experience. I’ve found the developers who are involved in the most successful projects and who have tackled the most complex systems also tend to have a strong pattern vocabulary. They are not only aware of many patterns that exist in software development, but also understand when and where they make sense.

I believe MVVM is popular because it has been so successful at providing the benefits listed earlier when implemented correctly. MVVM is an important pattern to learn and understand for Silverlight LOB applications—even if only to articulate why it doesn’t make sense for a particular project. Like all patterns, it is a tool and must be used for the right job. In the next few sections I’ll cover MVVM in more detail to help you learn the pattern and determine when it makes sense to use it in your applications. I’ll start by examining the components that make up MVVM.
The Model

The model is often confused with a “data model,” which is far too specific. A better definition is the application’s model of the world. It is the model that encompasses everything that must happen in order to solve the business problem without defining a specific user interface or presentation of data. Some like to call this the domain model, but a domain model is a conceptual representation, whereas the model in MVVM is an actual implementation.

To provide a simple example, a banking system might contain customers and accounts. The representations of customers and accounts are part of the model. The model describes how they are related: A customer has one or many accounts. It describes state (an account is open or closed) and provides behaviors (an account accrues interest). To make the model work requires implementations of classes with properties, a database to store the information, and loads of APIs to fetch data, transfer it, and apply various algorithms.

When built correctly, the model should expose only the parts needed by the application. For example, the presentation layer shouldn’t have to worry about how the data is stored (is it in a database or in an XML file?) or how the data is retrieved (was it parsed and passed as a binary object over a TCP/IP socket or sent over a REST service?). A model that is too open will create unnecessary dependencies and overcomplicate the code.

Regardless of whether you are a junior developer learning how to build applications or a seasoned architect who has worked on massive enterprise software projects, I believe it is important to know some key fundamentals of software design. As I stated before, I can’t point to a scientific study that states you must follow these principles to build quality software, but I can certainly point to my experience that developers who follow these principles tend to write code that ships with fewer defects, is easier to understand, can be readily maintained, and is produced more quickly than code written by developers who ignore them. These principles will assist you with building the model portion of your MVVM application correctly.
Don’t Repeat Yourself
The first principle to follow is called D.R.Y., which stands for Don’t Repeat Yourself. This is an easy principle to follow. As you are writing your software, you will find there are certain blocks of code and algorithms that repeat themselves. An experienced developer is able to quickly identify those patterns and refactor them into a single class or method. Not only does this save time by not repeating the pattern over and over, it also places the pattern in a single location to make it easier to change the algorithm when it becomes necessary. Finally, it protects other developers by giving them less opportunity to write bad code—it’s a lot easier to introduce bugs in a ten-line block of code than it is calling a simple API.

I recently came across an example of D.R.Y. for a customer project. I was writing a web service interface and quickly noticed a common pattern. Consider the following method that represents a completed web service call:

```csharp
private static void ClientGetWidgetCompleted(object sender,
                                            GetWidgetCompletedEventArgs e)
{
    if (e.Error != null)
    {
        throw e.Error;
    }

    var callback = e.UserState as Action<Widget>;

    if (callback != null)
    {
        callback(e.Result);
    }
}
```

Now look at a similar method that returns a list of related widget items:

```csharp
private static void ClientGetWidgetItemsCompleted(object sender,
                                                GetWidgetItemsCompletedEventArgs e)
{
    if (e.Error != null)
    {
        throw e.Error;
    }
```
The second method introduced what is referred to as code smell or the symptom of a deeper problem. In this case, the problem wasn’t a bug or defect with how the application runs, but instead a level of complexity that just isn’t necessary. Each time the method is implemented, the developer must remember to check for an error and throw it, then cast the state to get the callback, and finally invoke the callback. Forgetting just one simple step, such as invoking the callback, could result in application defects—in this case, one that would be hard to find because the code will simply wait for a result that never comes.

By looking at the methods and deciding that I did not want to repeat myself, I was able to create an extension method to simplify things. The method uses generics to abstract the part of the formula that changes—in this case the return type—while implementing the rest of the pattern in a single place.

```csharp
public static class ServiceExtensions
{
    public static Action<T> GetCallback<T>(
        this AsyncCompletedEventArgs args)
    {
        if (args.Error != null)
        {
            throw args.Error;
        }

        var callback = 
            args.UserState as Action<T> ??
            (obj =>
            {
                throw new Exception();
            });

        return callback;
    }
}
```
Now the previous service calls can be simplified to the following—notice how the code does not repeat the pattern, but instead focuses on the parts that are unique to each method:

```csharp
private static void ClientGetWidgetCompleted(object sender,
    GetWidgetCompletedEventArgs e)
{
    e.GetCallback<Widget>()(e.Result);
}

private static void ClientGetWidgetItemsCompleted(object sender,
    GetWidgetItemsCompletedEventArgs e)
{
    e.GetCallback<IEnumerable<WidgetItem>>()(e.Result);
}
```

The pattern here worked for the specific application, but I wouldn’t necessarily build it upfront in every project. There is always the possibility that another project will handle service calls completely differently and not use the same type of mechanism to abstract calls. This brings me to the next principle.

**You Aren’t Going to Need It**

You Aren’t Going to Need It—or Y.A.G.N.I—is one of the most difficult principles to follow. Some software architects tend to dislike this principle because it keeps them from working on really fun, complex systems by introducing all sorts of frameworks and patterns. Instead, it forces them to design simple, straightforward solutions that are written so cleverly that adding additional features when they are actually needed is not a problem. I’ve watched many systems collapse under the burden of features that were piled on “just in case” and then never used. The understanding of what may be needed often fails to meet the future requirement, and results in having to rip out the initial “guess” and extra refactoring to supply the more appropriate solution.

One example of this is the Enterprise Library, a library of implementations that is released by Microsoft’s own Patterns and Practices Team. I was asked by someone how excited I was when a build of this was released for Silverlight, and this person was surprised when I said I may not use it.
Don’t get me wrong—there are a lot of great features in the library and I’ve used it in many projects. The key, however, is to identify the parts you’ll need and use rather than plugging in the entire system and using only a fraction of it. A really common exercise is to load the Exception Handling Application Block because it allows dynamic configuration of your exception policies. Want to change where they are logged? Great—it handles that for you. Want to write certain exceptions to the database, swallow other exceptions, and write yet a third class to a rolling trace file? Not a problem! It’s a very flexible system.

What often ends up happening is the system is put in place, the configuration files are tinkered with until they are working, and then the application goes to production… and that’s it. The policies are never tweaked, there is no swapping of targets or sources, and often there is not a segregation of exception types. Essentially, a big piece of plumbing is put into place when a simple logging mechanism would have sufficed. It gets worse because then when someone does want to log or trace a certain set of events, they have to wade through reams of configuration to finally get the output they are looking for.

It is a much more straightforward approach to assume you aren’t going to need it, but build the code so that it is easy to add later on. In the case of exceptions, you can always provide a consistent pattern of exception blocks that are processed by a common interface, and start with an implementation that sends them to the event log. If you find a real need to enforce a specific set of policies, you can implement the Exception Handling Application Block in your single class and introduce the policies without having to touch the rest of the application.

Another example of Y.A.G.N.I. is caches. Despite there being no metrics to support it, many developers feel that a cache is needed. I’ve often heard, “We’re moving data, we need a cache.” But usually the overhead of synchronizing the cache for thread safety and applying various cache-expiration policies can make it slower than fetching the data directly from the database! There is no rule of thumb, and you must be prepared to analyze performance and obtain metrics in order to determine if there truly is a benefit.
Following this principle simplifies projects and makes them faster and easier to deliver. The fear many people have is that it will be too complex to rip out the code and replace it when a need is identified down the road, but you can cover those bases as well. As long as you build your code on a “solid” foundation, you’ll find it’s easy to introduce the features you do need when you are certain you really need them. That foundation is the next principle.

**The S.O.L.I.D. Principle**

I saved this principle for last because it’s the more complex one to learn and really builds on the first two. I also cheated because S.O.L.I.D. is not a single principle, but really a set of five principles that I believe are keys to building quality code. I include them here because I know firsthand the impact of applying them can have on the quality of your code. For MVVM to work well, the model must be done right, and S.O.L.I.D. is the foundation to make that happen.

An entire book could be written (and probably has been) about these principles, so I’ll only briefly introduce them here and hope you will explore them further online. S.O.L.I.D. is an acronym for the following principles:

- **Single Responsibility**—A class should be responsible for exactly one thing. A great example is a class that reads and parses a comma-separated text file. This class has more than one responsibility because it both reads the file and parses it. A better strategy would be to provide a class that reads files and a class that parses them. Now you can easily add a class that retrieves the file from a web service and passes it along to the parser, or a class that uses the reader to retrieve configuration information.

- **Open/Closed Principle**—A class should be open for extension but closed for modification. A great example is a class that has the responsibility for logging. It is common to provide a simple API that takes a message to log. This is fine until you need to specify different places for messages to go. Now the class has to be opened up to modify it
and the API changed to provide routing information. A more robust implementation would be a logger that provides a delegate for the call and allows registration of logging implementations. The class can now be extended by providing different logging mechanisms, and does not have to be opened to modify the behavior.

- **Liskov Substitution Principle**—This principle states that you should be able to substitute any class for its base class and have it behave the same way. The classic example of this is a square and a rectangle. If you derive the square from the rectangle and override the properties so that setting the width always sets the height, you can no longer substitute the class for its base class. Casting the square to a rectangle will provide unexpected behavior because with a rectangle, you expect to be able to set different widths and heights. A better design is to create an abstract base rectangle that provides getters and private setters. The rectangle implementation derives from this and makes the setters public, whereas the square derives from the same base class and provides a “length of side” property that sets the width and height at the same time.

- **Interface Segregation Principle**—This principle goes hand in hand with single responsibility and states that interfaces should be fine-grained and focused on a specific set of related tasks. An example of this is in the .NET Framework. Most classes have a concept of equality (what makes two classes the same) and comparability (what order the classes are in relation to each other in a list). These concepts are used in different ways at different times. Instead of a single interface for both concepts, the .NET Framework provides the IComparable interface and the IEquatable interface. If you are only concerned with equality, you can cast to IEquatable and deal directly with that simple interface. There is no need to involve any other interfaces that aren’t being used.

- **Dependency Injection/Inversion of Control**—This principle goes hand in hand with the single responsibility. When the class is required to create an instance of the logger, it is taking control of that dependency and going beyond its single responsibility. This can lead
to issues if you decide to swap to a different logging implementation and have to track down all of the places the old logger was created. Instead, you can simply provide a logging interface that is passed into the class. Now the class no longer has the responsibility of finding the logger; it simply consumes the interface. The dependency for the logger was injected and the control inverted to something else. We’ll discuss this concept in detail in the next chapter, which covers the Managed Extensibility Framework (MEF).

These principles work together to provide a set of guidelines for writing flexible, extensible, testable, and maintainable code. When the model of your application follows these principles, the MVVM pattern can easily connect to the interfaces and classes that are needed without creating dependencies on parts of the system that have nothing to do with presentation logic. The model is the “application model” of the real world, but at some point that model must be presented to the end user. This is done through output, which in the case of Silverlight is a very rich and powerful user interface (UI). The screen that the user is presented with is referred to as the view.

**The View**
The view in Silverlight is the easiest part to describe—it is what interacts with the user. The view itself is the user interface. The user interface is almost always represented using the declarative Xaml markup. The Xaml participates in the dependency property system, and the view is able to present information to the user as well as respond to user inputs. Table 7.2 shows common parts of the views and their function.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaml</td>
<td>Declarative markup to provide layout, controls, and other components that make up a screen</td>
</tr>
<tr>
<td>Value converters</td>
<td>Special classes used to transform data to a user element type and back</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Data templates</td>
<td>Templates that map data elements to controls</td>
</tr>
<tr>
<td>Visual state groups</td>
<td>Named states that impact the properties of various elements to provide a physical state based on the logical states of controls</td>
</tr>
<tr>
<td>Storyboards</td>
<td>Animations and transitions</td>
</tr>
<tr>
<td>Behaviors</td>
<td>Reusable algorithms that can be applied to various controls</td>
</tr>
<tr>
<td>Triggers</td>
<td>Algorithms that can be applied to controls and invoked based on configured events</td>
</tr>
<tr>
<td>Code-behind</td>
<td>Extensions of the Xaml markup to perform additional UI-specific tasks</td>
</tr>
</tbody>
</table>

It should be obvious from Table 7.2 that the view is not completely ignorant of presentation logic. Commands map controls to actions on the controller, and data-binding declarations require knowledge of the structure of the underlying data to bind to. Animations, visual states, templates, behaviors, and triggers all represent various components of business logic that relate to the view.

What may not be as obvious is that all of these components are stateless with regard to the model of the application. Storyboards maintain a state (started, stopped, playing, and so on) and visual state groups maintain a state, but all of these states are related to the UI. Behaviors and triggers also operate based on events or act on generic controls and should not be designed with dependencies on the underlying data and business logic. Even code-behind is typically written to facilitate certain aspects of the UI. More complex code should go somewhere else—not because there is a rule that code-behind is not allowed, but because more complicated algorithms need to be tested, and having a separate and decoupled class makes it easier to test without having to wire up a full UI.

So where does the bulk of presentation logic go, and what is responsible for maintaining the business state of the application? This state includes the data that is being presented as well as the status of various commands and processes that both drive the UI and respond to user inputs. The
answer is the essence of the MVVM pattern and the one element that makes it unique: the view model.

**The View Model**
The view model is what makes MVVM unique. It is simply a class that holds the responsibility of coordinating the interaction between the view and the model. The view model is where the bulk of the presentation logic should reside. In my opinion, a well-written view model can be tested without creating any views and has three main methods for communication with the view:

- Data-binding
- Visual states
- Commands and/or method calls

With this definition in mind, you’ve already created a view model. In the previous example using countries and states, the class that held the lists of states and countries was the view model. View models typically implement the property-change notification interface and one of the validation interfaces, and they also have some type of connection to the Visual State Manager. (I’ve seen some fairly elaborate hacks try to launch storyboards from view models when a simple visual state transition was all that was needed.)

In the previous examples, you used one of two methods to instantiate the class that functions as the view model. The first was to declare an instance in the resources and then use a static reference to bind to the class. The second was to instantiate the class directly in the data context definition for the root panel of the control, most often the grid called LayoutRoot. This approach creates a direct dependency between the view and the view model. Oftentimes you’ll want to share the same view model between different views (for example, when each view is a different representation of the same data) or you’ll want to dynamically change the view or view model based on a condition. It is also popular to use a design-time view model to represent design data, so the view model may be different during design and runtime. What is the best way to resolve this binding?
Binding the View Model to the View

Binding the view model to the view is one of the key problems many MVVM frameworks try to solve. It can be done various ways, and there is no preferred method; otherwise, all experienced developers would be doing it the same way. The most popular method seems to be using a view model locator, whereas some people prefer to spin up controllers that perform the binding. My preferred method is to use the built-in design-time view model functionality provided by Silverlight using the design-time extensions. With Silverlight 5, it is also possible to use custom markup extensions.

In this section, you explore various methods to bind the view model to the view, and you can decide which method makes the most sense for your applications. The view model in each case processes a selection from a drop-down and render a shape.

View Model Locators

View model locators are simply classes that expose the view models as properties—typically as interfaces. The locator can determine the state of the application and return the appropriate view model based on whether it is called during design time or runtime. To see how a view model locator works, create a new Silverlight Application project called ViewModel-Locators. Specify a grid with four equally sized quadrants (two rows and two columns). Provide an interface for the view model:

```csharp
public interface IViewModelInterface
{
    List<string> Shapes { get; }
    string SelectedShape { get; }
}
```

Now implement a design-time view model:

```csharp
public class DesignViewModel : IViewModelInterface
{
    public List<string> Shapes
    {
        get
        {
```
Create the runtime view model. It will implement the property-change notification interface and provide a delegate for transitioning the visual state. The code for the view model is shown in Listing 7.1.

Listing 7.1: The Main View Model

```csharp
public class ViewModel : IViewModelInterface, INotifyPropertyChanged
{
    public ViewModel()
    {
        GoToVisualState = state => { }; 
    }

    private readonly List<string> _shapes
        = new List<string>
        {
            "Circle",
            "Square",
            "Rectangle"
        };

    public List<string> Shapes
    {
        get { return _shapes; }
    }

    public Action<string> GoToVisualState { get; set; }

    private string _selectedShape;
    public string SelectedShape
    {
        get { return _selectedShape; }
        set
        {
            _selectedShape = value;
            var handler = PropertyChanged;
        }
    }
}
```
if (handler != null)
{
    handler(this, new PropertyChangedEventArgs("SelectedShape"));
}

if (!string.IsNullOrEmpty(value))
{
    GoToVisualState(value);
}

Now you can create the ViewModelLocator class. The locator will simply determine whether it is being called during design time or runtime and return the appropriate view model.

public class ViewModelLocator
{
    private IViewModelInterface _viewModel;

    public IViewModelInterface ViewModel
    {
        get
        {
            return _viewModel ?? (_viewModel = DesignerProperties.IsInDesignTool
                ? (IViewModelInterface)
                 new DesignViewModel()
                : new ViewModel());
        }
    }
}

Now you can create a control to implement the behavior. The control will include the view model locator in its resources (the locator is typically declared at the App.xaml level to make it available to the entire application). Add a new control called LocatorControl.xaml and fill it with the following Xaml:
Now define the shapes and place them in the second row of the LocatorControl:

```xml
<Ellipse x:Name="CircleShape"
    Width="50" Height="50" Fill="Green" Grid.Row="1"/>
<Rectangle x:Name="SquareShape"
    Width="50" Height="50" Fill="Blue" Grid.Row="1"/>
<Rectangle x:Name="RectangleShape"
    Width="100" Height="50" Fill="Red" Grid.Row="1"/>
```

Add the visual state groups. Part of the Xaml for the first state is shown in the following code. Repeat the ObjectAnimationUsingKeyFrames for the square and the rectangle, but set their visibility to the collapsed state. Next, copy and paste the visual state twice more for the square state and the rectangle state and then update which shapes are visible and which shapes are not.

```xml
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ShapeGroups">
    <VisualStateGroup.States>
      <VisualState x:Name="Circle">
        <Storyboard>
          <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="CircleShape"
            Storyboard.TargetProperty="(UIElement.Visibility)"
            Storyboard.KeyTime="0:0:0">
            <DiscreteObjectKeyFrame Value="Visible"/>
          </ObjectAnimationUsingKeyFrames>
        </Storyboard>
      </VisualState>
      <VisualState x:Name="Square">
        <Storyboard>
          <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="SquareShape"
            Storyboard.TargetProperty="(UIElement.Visibility)"
            Storyboard.KeyTime="0:0:0">
            <DiscreteObjectKeyFrame Value="Visible"/>
          </ObjectAnimationUsingKeyFrames>
        </Storyboard>
      </VisualState>
      <VisualState x:Name="Rectangle">
        <Storyboard>
          <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="RectangleShape"
            Storyboard.TargetProperty="(UIElement.Visibility)"
            Storyboard.KeyTime="0:0:0">
            <DiscreteObjectKeyFrame Value="Visible"/>
          </ObjectAnimationUsingKeyFrames>
        </Storyboard>
      </VisualState>
    </VisualStateGroup.States>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```
Set the locator control to navigate to a default visual state and set the
delegate for transition states. This is done in the control code-behind:

```csharp
public LocatorControl()
{
    InitializeComponent();

    if (DesignerProperties.IsInDesignTool) return;

    VisualStateManager.GoToState(this, "Circle", false);

    var locator = (ViewModelLocator) Resources["Locator"];
    var vm = locator.ViewModel as ViewModel;
    vm.GoToVisualState = state => VisualStateManager.GoToState(this, state, true);
}
```

The binding to the view model is the side effect of using the locator. You
want to change the visual states from the view model, but the UserControl
implements the functionality. Because of the context it is used in, it is not
possible to get access to the host control and automatically bind the dele-
gate. This must be done in the control itself by passing a delegate for the
GoToState method to the view model.

Now you can declare an instance of the control in the main page in the
upper-left quadrant and run the application. The shapes will swap out each
time you select a new control. You will also see values in the combo box
during design time because the view model locator returns the design-time
view model for you.

**Controllers**

One of the less-common methods for binding the view model to the view
is by using a controller. Contrary to popular belief, there is no law that
MVVM cannot be mixed with other patterns. The controller pattern
involves creating a class specifically tied to the view it is managing and then using the controller to spin up the view model and wire the logic.

First, create a new control called **ControllerControl.xaml** and copy the contents of the locator control. Remove the resources collection that references the view model locator, and remove the data context attribute on the grid that uses the locator resource. Next, create a new class called **Controller** and provide it with a constructor, like this:

```csharp
public class Controller
{
    private readonly IViewModelInterface _vm;

    public Controller(Control control)
    {
        if (DesignerProperties.IsInDesignTool)
        {
            _vm = new DesignViewModel();
        }
        else
        {
            _vm = new ViewModel
            {
                GoToVisualState =
                    state => VisualStateManager.GoToState(
                        control, state, true)
            };
            VisualStateManager.GoToState(
                control, "Circle", false);
        }
        control.DataContext = _vm;
    }
}
```

Note that for testing, the element passed in would normally be an interface that includes the data context element. This would allow the controller to be tested without passing an actual control, as is done here to keep the example simple. Now that the controller is defined, it simply needs to be created in the code-behind of the control:

```csharp
public partial class ControllerControl
{
    public Controller Controller { get; set; }
}
```
Create an instance of the control in the upper-right quadrant of the main page and compile the application. You’ll see design-time data in the main page, and when you run the application you’ll be able to swap the shapes in the control in the upper right.

**Design-Time View Models**

My favorite method to use because it works directly with the attributes supplied by the runtime is the use of design-time view models. In this approach, the design-time data is supplied using the design-time extensions. The runtime binding is done using an external mechanism. You’ll learn a more elegant way to perform the binding in Chapter 8, “The Managed Extensibility Framework (MEF).” For now, a simple class will perform the necessary wiring.

Create a new control called `DesignControl` and copy the contents of the controller control (basically everything from the main grid down). Add the following attribute to the `LayoutRoot` grid to specify the design-time view model:

```xml
<d:DataContext="{d:DesignInstance Type="ViewModelLocators:DesignViewModel, IsDesignTimeCreatable=True"}"
```

Listing 7.2 shows what the binder might look like—although there is only one entry in the dictionary, you can see how it would be easy to add additional entries mapping the control type to the view type. The dictionary could also reference instances of the view models to share the same one between views, and the wiring of the delegate can be made generic by implementing a common view model interface that specifies the delegate.
LISTING 7.2: A View Model Binder

```csharp
public class Binder
{
    private readonly Dictionary<Type, Type> _bindings =
        new Dictionary<Type, Type>
        {
            { typeof(DesignControl), typeof(ViewModel) }
        };

    public void Bind(Control control)
    {
        if (!_bindings.ContainsKey(control.GetType()))
        {
            return;
        }

        var vm = Activator
            .CreateInstance(
                _bindings[control.GetType()]);

        ((ViewModel) vm).GoToVisualState =
        state => VisualStateManager
            .GoToState(control, state, true);

        control.DataContext = vm;
    }
}
```

The control can then call the binder and set the default state, like this:

```csharp
public DesignControl()
{
    InitializeComponent();
    new Binder().Bind(this);
    VisualStateManager.GoToState(this, "Circle", false);
}
```

Declare an instance of this control in the lower-left quadrant and run the application.

**Custom Markup Extensions**

Custom markup extensions can extend the Xaml model to include your own tags and properties. View model binding is a perfect example of how custom extensions can be used. First, create a control called `MarkupControl`
and copy the content of the controller control (should be just the grid and so on, with no resources and no data context). In the code-behind, just set the default visual state:

```csharp
public MarkupControl()
{
    InitializeComponent();
    VisualStateManager.GoToState(this, "Circle", false);
}
```

Now create a custom markup extension. You can revisit Chapter 3, “Extensible Application Markup Language (Xaml),” to learn more about how to create the extensions. This extension grabs the object root to get a reference to the control that is hosting it. See Listing 7.3 for the full extension. It spins up the view model based on the type of the control, binds the visual state transitions, and returns the view model (in design time, it just returns a new instance of the design view model).

**Listing 7.3: A Custom Markup Extension for View Model Binding**

```csharp
public class ViewModelBinderExtension : MarkupExtension
{
    public override object ProvideValue(
        IServiceProvider serviceProvider)
    {
        var targetProvider = serviceProvider
            .GetService(typeof(IRootObjectProvider))
            as IRootObjectProvider;

        if (targetProvider == null) return null;

        var targetControl = targetProvider.RootObject
            as UserControl;

        if (targetControl == null) return null;

        if (targetControl is MarkupControl)
        {
            if (DesignerProperties.IsInDesignTool)
            {
                return new DesignViewModel();
            }
        }
    }
}
```
With the extension created, you can add a namespace reference in the control:

```xml
xmlns:local="clr-namespace:ViewModelLocators"
```

Then set the data context of the grid using the extension. It will return the design-time view model in the designer and resolve the runtime view model when you run the application.

```xml
<Grid x:Name="LayoutRoot" Background="White"
     DataContext="{local:ViewModelBinder}"
>
```

Add the control to the lower-right quadrant of the main page and run the application. You will now have four separate controls that perform the same thing using the MVVM pattern. The view model is the same in each case; the only difference is how it is bound to the control.

**View-Model-First Approach Versus View-First Approach**

All of the examples in this chapter have focused on what is referred to as a view-first approach. The view is instantiated and then the view model is found and bound to the view. I believe this makes sense because most users operate by moving through screens and most developers comprehend their applications in terms of screens. It’s important to note that this is not the only way to manage the application.
Some frameworks such as Caliburn.Micro (http://caliburnmicro.codeplex.com/) follow what is known as a view-model-first approach. In this approach, the application logic spins up view models and the view models determine what views are provided. In a sense the binding is more straightforward because you can use a convention-based approach to map the view to the view model. (Convention-based simply means the bindings can be made based on how you name your classes, so a view is always named SomethingView and a view model is always named SomethingViewModel. Therefore, the Something in common creates the binding.)

The disadvantage to this approach is that it is often not design-time friendly. It is difficult to design the application in Visual Studio or Blend because the designer doesn’t know the right way to spin up views based on the view models. The designers are inherently view-first because you navigate to a Xaml view in order to render the view in the designer.

Although I strongly prefer the view-first approach, the view-model-first frameworks are very popular and have been used in many applications. Proponents of this approach highly prefer it over the view-first approach. It’s not a question of one approach being correct but more a style preference. It makes sense to follow the model that not only is easiest for you to understand, but will also be easy for your team to follow and maintain.

Lists and Data Elements

One situation you might find confusing when using the MVVM model is how to handle lists. Although it’s clear that you may have a view model for a list of contacts that provides a list of contact items, it’s not as clear how to manage displaying those contacts in an editable grid. Do you have to create a view model for each contact? Is it a violation of MVVM if the contacts implement property-change notification directly?

These questions are really a matter of splitting hairs rather than solving the problem. The answer is to define the solution that makes the most sense. In most cases, implementing property-change notification on the individual items is fine. The overhead of wrapping each item in a view model doesn’t make sense when so much logic is duplicated. By default,
Silverlight client proxies for web services will implement property-change notification for complex data types. You can also specify a shared assembly to define the type directly, and inherit from a simple base type to implement the property-change notification.

Each item can then be tracked for edits. When common commands can be performed, such as a delete command, the item can be passed to a command on the parent view model so the logic is maintained in one central place. This keeps the overall approach simple and straightforward. If you end up implementing a detail screen that has a dedicated view model, you can refactor the grid to use the same view model if it makes sense.

The Silverlight framework provides the ObservableCollection class for managing lists that may change. This special collection will raise a collection-change notification. This can be used by UI elements to refresh their contents and by your code to respond to changes in the list. If each individual item also has significant presentation logic, you may end up projecting the list to a collection of view model items.

There are no hard-and-fast rules. MVVM is about making it easier to develop applications by taking advantage of code reuse and the built-in data-binding system that Silverlight uses. If you find that using MVVM is making your project more complex and difficult to maintain, you’re likely doing it wrong and should take a step back to reevaluate your approach. The next few chapters will cover some ways to make sure you’re reaping the benefits of MVVM.

**Summary**

In this chapter, you learned about the Model-View-ViewModel pattern and how it may benefit you when building Silverlight LOB applications. You learned that the model represents the application in general and should follow best practices around decoupled code. You learned about the D.R.Y., Y.A.G.N.I., and S.O.L.I.D. principles as cornerstones for quality software development. You learned about the role of the view and how the view model is designed to coordinate and synchronize state between the view and the underlying application model. The view model essentially encapsulates what is referred to as presentation logic.
In the next chapter, you will learn about a framework that works hand in hand with MVVM to help you create modular and extendable applications. The Managed Extensibility Framework (MEF) provides services that help you manage decoupled code and create extension points in your application so it is easy and straightforward to extend at both compile-time and runtime. You will learn about lifetime management and metadata as tools to create a framework for building large LOB applications.
This page intentionally left blank
Index

A
access, isolated storage, 478-482
accessing files and folders
isolated storage, 482-484
actions, asynchronous techniques, 456-458
ActivateVm, 522
adapters, 246
region management, 395-396
AndThat method, 242
animation behaviors, Expression Blend SDK, 23
animations, 94
Apache, XAP files, 58
APM (Asynchronous Programming Model), 452
versus events, 452-455
application lifecycle, 33-36
application services, 32-33
application lifecycle, 33-36
sample XAP download service, 37-41
application signing, elevated trust, 528-530
applications
maintaining, reasons for testing, 325-326
profiling with Visual Studio, 622-625
architecture, 2
BCL (Base Class Library), 2
CLR (Common Language Runtime), 1
Communications, 4
data, 4
improving, reasons for testing, 326
presentation core, 3
presentation framework, 3
arrange, layout (Xaml), 96
ASP.NET callbacks, 402, 407-412
assemblies, XAP files (standard solution), 63-64
assumptions, eliminating, 323
Asynchronous Programming Model (APM), 452
asynchronous techniques, 452
actions and callbacks, 456-458
events versus APM, 452-455
lambda expressions and method chaining, 455-456
Rx (Reactive Extensions), 459-463
tasks and await, 463-465
asynchronous validation, 230-232
atomicity, 590
attached properties, 71, 75
attributes, validation, 223
AutoCompleteBox, 110
automated testing, 343-345
automation peers, testing, 367-371
await, asynchronous techniques, 463-465

B
backing properties, 72
Base Class Library (BCL), 2
basic controls, Xaml, 108-109
Baskin, Dave, 615
BCL (Base Class Library), 2
behaviors, 141-145, 400-401
binding
  markup extensions, 77
  view model to the view, MVVM, 265-275
binding objects, parameters, 196
bitmap print, forcing, 570
bootstrappers, 281
Border, 108
bounce ease, 95
browsers, elevated trust, 552-553
bugs, 597
  killing at the source, reasons for testing, 324
Button, 108
button groups, states and, 163

C
Calendar, 110
callbacks
  ASP.NET, 407-412
  asynchronous techniques, 456-458
CanExecute method, 218
canvas, layout (Xaml), 96
case studies
  Fluent and Fast, 243
  regions through the looking glass, 395
catalogs, discovery (MEF), 287-288
chain extension methods, 241
challenges, testing, 366
character spacing, text (Xaml), 120-121
CheckBox, 108
checking for updates, OOB apps, 523-526
child windows, elevated trust, 535-540
ChildWindow, 110
choosing
client technologies, 14
  analyze third-party dependencies, 17-18
  consider the development team, 16-17
  listen to the customer, 15-16
  use what you know, 15
page-based navigation, 377-378
Cider, 132
circle ease, 95
class instantiation, Xaml, 68-70
class libraries, 29-32
classes, linked classes, 338-343
client logging, 617-619
client technologies, choosing, 14
  analyzing third-party dependencies, 17-18
  consider the development team, 16-17
  listen to the customer, 15-16
  use what you know, 15
client tracing, 617-619
clientaccesspolicy.xml, 57
clients, sharing domain objects between clients and servers, 401-402
CLR (Common Language Runtime), 1
  key differences from the full CLR, 5
code, documenting (reasons for testing), 324-325
code smell, 257
coded UI tests, 364-366
ColorAnimation, 94
COM interop, elevated trust, 540-546
ComboBox, 108
commands, data-binding, 218-220
commercial profilers, 616
Common Language Runtime (CLR),
  key differences from the full CLR, 5
communication, 4, 412
  local messaging, 435-436
POX (plain old XML), 423-424
REST (Representational State Transfer), 412-413
  consuming REST services, 421-423
  repository pattern, 416-418
  securing the website, 418-421
Sterling, 414
  Sterling database definition, 414-416
sockets, 436-443
WCF (Windows Communication Foundation), 424
  SOAP, 425-431
WCF RIA Services, 431-435
composition initializer, discovery (MEF), 289-291
composition step, 577
concurrency, 590-593
conditional behaviors, Expression Blend SDK, 23
consistency, 590
consuming REST services, 421-423
contact class, 239
containers
discovery, MEF, 288-289
navigation, 379
  content control, 379-382
  custom panels, 382-383
  items control, 382
Xaml, 104
  ContentControl, 104-105
  ItemsControl, 105
  ScrollViewer, 105
  ViewBox, 106-107
content control, containers, 379-382
ContentControl, containers (Xaml), 104-105
ContentProperty, 68
contracts, 577
control contracts, 123
controllers, MVVM, 269-271
controls
  basic controls, 108-109
toolkit controls, 110-111
Converter, 196
ConverterCulture, 196
ConverterParameter, 196
critical code, 5
cross-domain considerations, web hosts
(second solution), 56-58
cross-domain policies, 57
CRUD (Create, Read, Update, and Delete), 19
custom markup extensions, MVVM, 272-274
custom navigation, 378-379
custom panels, containers, 382-383
custom type providers, mapping and transformation, 451-452
custom VSM (Visual State Manager), 176-177
CustomValidationAttribute, 223
Cwalina, Krysztof, 279
D
D.R.Y. (Don’t Repeat Yourself), 256-258
data, 4
Data Annotations, 221
data behaviors, Expression Blend SDK, 23
data elements, MVVM, 275-276
data error info, validation, 226-230
data paging with MVVM, 586-590
data sets, extremely large
scalability, 578-580
data paging with MVVM, 586-590
OData, 580-583
WCF RIA, 583-585
data templates, Xaml, 128-131
data-binding, 195
commands, 218-220
debugging, 200-203
overview, 196-200
within styles, 203-207
synchronizing lists, 208-218
validation, 220-224
asynchronous validation, 230-232
data error info, 226-230
with exceptions, 224-226
fluent validation, 232-242
database team, 560
DataGrid, 110
DataPager, 110
DataPicker, 110
DataTypeAttribute, 223
debug mode, 314
debug symbols, 600-601
debug tests, 322
debuggers
MEF debugger, 316-318
stopping, 609
debugging
data-binding, 200-203
Fiddler, 625
Silverlight apps, 598-600
debug symbols, 600-601
debugging applications already launched, 608-611
debugging tips, 602-607
WinDbg, 611-616
Silverlight Spy, 626-627
debugging tips, 602-607
decryption method, 496
dependency injection, 261
dependency properties, 70
Xaml, 70-74
attached properties, 75
value precedence, 76
deployment catalog, extensibility (MEF), 301-302
DescriptionViewer, 110
design patterns, UI design patterns model, 255-262
Model-View-ViewModel pattern, 251-254
MVVM, 246-250
view, 262-264
view model, 264
design teams, 560
design-time extensions, Xaml, 132-141
design-time view models, MVVM, 271-272
designer/developer workflow, 559-561
development tests, 322
discovery, MEF, 281
catalogs, 287-288
composition initializer and host, 289-291
containers, 288-289
exports, 284-285
imports, 281-284
parts, 286-287
discovery step, 577
distributing fonts, 155-156
distribution, 553
execution, 554
installation, 553-554
Document Object Model (DOM), 366
documenting code, reasons for testing, 324-325
DOM (Document Object Model), 366
DOM interop, 402
DOM interoperability, 404-407
updated to-do list, 402-404
DOM interoperability, 404-407
domain data, 400-401
domain model, 255
domain objects, sharing between clients and servers, 401-402
Don’t Repeat Yourself (D.R.Y.), 256-258
DoubleAnimation, 94
downloading XAP, 38
durability, 590
dynamic types, mapping and transformation, 448-450

e
edit-and-continue debugging, 598
ElementName, 197
elevated trust, 526-527
application signing, 528-530
child windows, 535-540
COM interop and script host, 540-546
file system access, 530-531
in browsers, 552-553
native Silverlight extensions, 546-547
p/Invoke, 547-551
toast notifications, 531-535
eliminating assumptions, reasons for testing, 323
embedding fonts, 155-156
encrypting files in Silverlight, 494-498
enterprise concerns, 558
Enterprise Library, 258
entity view model, 353
EnumDataTypeAttribute, 224
environments, setting up, 21
Expression Blend SDK, 22-24
open source projects, 25-26
Silverlight 5 SDK and Visual Studio tools, 22
Silverlight Toolkit, 25
event aggregators, navigation events, 384-389
event-handler methods, Xaml editor, 89
events
versus APM, 452-455
VSM, 172-175
exceptions, validation, 224-226
Execute method, 218
execution, 554
exports, discovery (MEF), 284-285
exposing the model versus mapping, 443-448
Expression Blend
VSM (Visual State Manager), 177-185
XAP extensions, 157
Expression Blend SDK, 7
setting up environments, 22-23
animation behaviors, 23
conditional behaviors, 23
data behaviors, 23
motion behaviors, 24
triggers, 24
visual state actions, 24
extensibility, 576-577
MEF, 295-297
deployment catalog, 301-302
offline catalog, 305
recomposition, 297-301
XAP files, 303-305
extension methods, 241
extension XAP, creating, 41-47
extensions, native Silverlight extensions (elevated trust), 546-547
external parts, 63
failed updates, 525
FallbackValue, 197
Fiddler, 625
file system access, elevated trust, 530-531
file systems, iterating, 485-490
files
encrypting in Silverlight, 494-498
isolated storage, 476-478
accessing, 482-484
signing files, 491-494
filters, to-do list application, 471-472
finding isolated storage, 484-485
fluent validation, 232-242
fluid layout, 95
folders, isolated storage, 476-478
accessing, 482-484
fonts, embedding and distributing, 155-156
frames, 374
Garland, John, 31
Gossman, John, 251
GPU (graphics processing unit), 3
grid, layout (Xaml), 97-101
grid listings, 147
grid sizes, 101
GridLength property, 100
GridSplitter, 110
groups, VSM (Visual State Management), 162-165

H
happy path, 323
“Hello, Silverlight” application, 26
application services, 32-33
application lifecycle, 33-36
sample XAP download service, 37-41
class libraries, 29-32
creating, 27-28
creating extension XAP, 41-47
sharing between Silverlight and the core framework, 48-53
templates, 53-55
Hoare, Sir Charles Antony, 557
hosts, discovery (MEF), 289-291
HTML 4.01, 9
HTML5, 8-14
benefits of, 14
limitations of, 11
LOB apps, 10-11
HyperlinkButton, 108

I
IApplicationService, 32
IIS 4.0, XAP files, 58
IIS 5.0, XAP files, 58
IIS 6.0, XAP files, 59
IIS 7.0, XAP files, 59
IL (Intermediate Language), 48
IL disassembler, 48
ildasm.exe, 48
Image, 108
import parameters, MEF, 315-316
imports, discovery (MEF), 281-284
improving architecture, reasons for testing, 326
InitializeVm method, 520
InkPresenter, 108
INotifyPropertyChanged interface, 45
/install, 553
installation, 553-554
integer range validator, 236
integration tests, 322
interactions, 250
interactivity
behaviors, 141-145
triggers, 141, 147-149
Interface Segregation principle, 261
Intermediate Language (IL), 48
inversion control, 261
IServiceProvider, 79
isolated storage, 467
files and folders, accessing, 482-484
finding, 484-485
folders and files, 476-478
iterating the file system, 485-490
managing access and quotas, 478-482
isolating themes, 151-155
isolation, 590
IsSelected property, 216
items control, containers, 382
ItemsControl, containers (Xaml), 105
iterating file systems, isolated storage, 485-490
iterators, 246
INDEX

J-K
JavaScript debugging, 599
JavaScript Object Notation (JSON), 4, 56, 423-424
Jounce, 318-319, 388, 461
journal ownership, 375
journals, 375
JSON (JavaScript Object Notation), 4, 56, 423-424
just-in-time debugging, 598
JustMock, 346

L
Label, 110
lambda expressions, 455-456
Language Integrated Query (LINQ), 4
layout, Xaml, 95-96
arrange, 96
canvas, 96
grid, 97-101
measure, 96
StackPanel, 101-103
VirtualizingPanel, 103
libraries, 5
class libraries, 29-32
lifetime management, MEF, 291-295
line height, text (Xaml), 121-123
linked classes, 338-343
LINQ (Language Integrated Query), 4
LINQ query, 214
Liskov Substitution principle, 261
ListBox, 108
listings
Code to Parse Text and Create a Highlight Rectangle, 118
Code to Parse the Handler and Create Deployment Catalogs, 304
Creating a Style Dynamically, 90
Custom Visual State Manager, 176
Debugging the Rectangle Sizes, 100
Detail View Code-Behind, 293
Downloading the XAP File Using WebClient, 39
Generated Code for the App Class, 60
Grid Set Up to Demonstrate the ViewBox Control, 107
Grid with Associated States and Transitions, 188
IL Code for the Constructor of the WelcomeContainer Class, 49
Main Page with Code to Flip the Switch, 168
MainPage.xaml Rewritten with Code Only, 69
MainPage.xaml.cs for the Parts and Templates Example, 131
Markup Extension that Uses the IServiceProvider Interface, 80
Modified WelcomeContainer Class, 46
Output from a Sample Run with the VSM Debugger Behavior, 175
Parsing the MEF Container, 316
Parsing the XAP File Manifest and Assemblies, 40
Sample clientaccesspolicy.xml File, 57
Sample Debug Output, 312
Shell for the XapDownloadService Class, 37
Style with a Little Bit of Everything, 89
Style, Grid, and Rectangle Definitions, 98
Testing the Addition of a New To-Do Task, 358
Type Converter for Floats (Singles), 83
Using RichTextBoxOverflow to Flow around Images and Create Columns, 119
Value Converter, 84
Visual State Aggregator, 191
Visual State Subscription, 189
Visual State Subscription Behavior, 192
Xaml for MainPage.xaml, 68
Xaml to Create a Simple Switch, 165
Xaml to Define and Execute an Animation, 74

lists
MVVM, 275-276
synchronizing, 208-218
LOB (line of business), 558
LOB (line of business) apps, 18-19
HTML5, 10-11
local messaging, 435-436
local tests, 321
localization, 571-575
LocalizeExtension, 573
logging, 616-617
client logging, 617-619
Looking Glass, regions case study, 395

M
maintaining applications, reasons for testing, 325-326
MainViewModel class, 520
Managed Extensibility Framework (MEF)
discovery, 281
catalogs, 287-288
composition initializer and host, 289-291
containers, 288-289
exports, 284-285
imports, 281-284
parts, 286-287
extensibility, 295-297
deployment catalog, 301-302
offline catalog, 305
recomposition, 297-301
XAP files, 303-305
Jounce, 318-319
lifetime management, 291-295
metadata, 306
strongly typed metadata, 309-313
weakly typed metadata, 306-309
mocking and, 345-352
troubleshooting, 313-314
import parameters, 315-316
MEF debugger, 316-318
stable composition, 314-315
managing isolated storage access and quotas, 478-482
manifests, XAP files (standard solution), 62-63
manual navigation, 379
containers, 379
content control, 379-382
custom panels, 382-383
items control, 382
navigation events, 384
  event aggregators, 384-389
navigation journals, 389-391
region management, 391
  adapters, 395-396
  nested regions views, 397
  regions, 391-395
  views, 397
mapping
transformation and, 443
  custom type providers, 451-452
dynamic types, 448-450
  versus exposing the model, 443-448
markup, Xaml, 68-70
mark up extensions, Xaml, 76-81
measure, layout (Xaml), 96
MediaElement, 108
MEF (Managed Extensibility Framework), 279-281
discovery, 281
  catalogs, 287-288
  composition initializer and host, 289-291
  containers, 288-289
  exports, 284-285
  imports, 281-284
  parts, 286-287
extensibility, 295-297
  deployment catalog, 301-302
  offline catalog, 305
  recomposition, 297-301
  XAP files, 303-305
Jounce, 318-319
lifetime management, 291-295
metadata, 306
  strongly typed metadata, 309-313
  weakly typed metadata, 306-309
mocking and, 345-352
troubleshooting, 313-314
import parameters, 315-316
  MEF debugger, 316-318
  stable composition, 314-315
MEF debugger, 316-318
memory profiling, 622
metadata, MEF, 306
  strongly typed metadata, 309-313
  weakly typed metadata, 306-309
method chaining, 455-456
Microsoft UI Automation, 367
mixed-mode debugging, 598
mocking, MEF and, 345-352
  MEF and, 345-352
  MEF debugger, 316-318
        Mocks, 351
        Mode, 197
model, MVVM, 255
  D.R.Y. (Don’t Repeat Yourself), 256-258
  S.O.L.I.D. principle, 260-262
  Y.A.G.N.I. (You Aren’t Going to Need It), 258-260
Model-View Presenter (MVP)
  pattern, 250
Model-View-Controller (MVC)
  pattern, 249
Model-View-ViewModel pattern,
  251-254
Model-View-ViewModel. See MVVM,
  245, 279
modularity, 576-577
Moq, 345-346
motion behaviors, Expression Blend SDK, 24
MultiScaleImage, 108
multiselection class, 214
MVVM (Model-View-ViewModel), 245, 279
binding view model to the view, 265 controllers, 269-271
custom markup extensions, 272-274
design-time view models, 271-272
view model locators, 265-269
view-model-first approach versus view-first approach, 274-275
data paging, 586-590
Jounce, 318-319
lists and data elements, 275-276
misconceptions, 252
UI design pattern model, 255-262
Model-View-ViewModel pattern, 251-254
view, 262-264
view model, 264
UI design patterns, 246-250
view model locators, 266

native Silverlight extensions
  elevated trust, 546-547
  p/Invoke, 547-551
natural user interface (NUI), 113, 149-151
navigation, manual navigation, 379 containers, 379-383
  navigation events, 384-391
  region management, 391-397
  navigation events, 384
  event aggregators, 384-389
  navigation journals, 389-391
  navigation framework, 374-375
    basics of, 375-377
    choosing page-based navigation, 377-378
    custom navigation, 378-379
    navigation journals, navigation events, 389-391
  navigation parameters, to-do list application, 472-473
  navigation template, 54
  NavigationContext, 376
  nested regions, views and, 397
  .NET Framework, libraries, 5
  NotifyOnValidationError, 197
  NUI (natural user interface), 113, 149-151

O
  ObjectAnimationUsingKeyFrames, 94
  OData, handling data sets, 580-583
  OData (Open Data) protocol, 413
  offline catalog, extensibility (MEF), 305
  old XML (POX), 412
  OOB (Out of Browser) apps, 305, 405, 505
    checking for updates, 523-526
    distribution and installation, 553-554
    elevated trust in browsers, 552
    getting started, 516-523
    uninstalling, 555
  open source projects, setting up environments, 25-26
  Open/Closed Principle, 260
OpenFileDialog, 109
optimistic concurrency, 591
/out, 553
Out of Browser (OOB) app, 305, 405
overflow, rich text (Xaml), 119-120
overwrite, 554

P
p/Invoke, elevated trust, 547-551
Page, 110
page-based navigation, choosing,
377-378
parameters, passed to the binding
object, 196
parsing rich text, Xaml, 116-119
parts
discovery, MEF, 286-287
Xaml, 123-126
parts and states model, 123
PasswordBox, 109
path, 197
PDB files, 599
performance profiling, 622
persistence, 467
persisting preferences with settings,
473-476
pessimistic concurrency, 591
Plain Old CLR Objects (POCOs), 196
Platform Invoke (p/Invoke), 516
POC (proof of concept), 594-595
POCOs (Plain Old CLR Objects), 196
PointAnimation, 94
Popup, 109
portable assemblies, 48
POX (plain old XML), 4, 56, 412, 423-424
presentation core, 3
presentation framework, 3
presentation model, 249
print event, 568
printing, 558, 561-571
bitmap print, forcing, 570
private builds, 599
ProcessPage method, 589
ProcessReport method, 568
profiling, 622
applications with Visual Studio,
622-625
program database file, 599
ProgressBar, 109
proof of concept (POC), 594-595
properties
attached properties, 71, 75
dependency properties, 70
of Timeline class, 91-92
prototype pattern, 246
public builds, 599

Q
QA testing, 322
quotas, isolated storage, 478-482

R
RadioButton, 109
RangeAttribute, 224
Reactive Extensions (Rx), asynchronous
techniques, 459-463
recomposition, extensibility (MEF),
297-301
refactoring isolation, 245
reflection cache, building, 445
region management, 391
manual navigation, 391
   adapters, 395-396
   nested regions views, 397
   regions, 391-395
   views, 397
regions
   region management, 391-395
   super regions, 396
RegularExpressionAttribute, 224
RelativeSource, 197
   markup extensions, 78
remote procedural calls (RPC), 412
RepeatButton, 109
repository pattern, 416-418
Representational State Transfer (REST), 412-413
   consuming REST services, 421-423
   repository pattern, 416-418
   securing the website, 418-421
Sterling, 414
   database definition, 414-416
REST services, consuming, 421-423
Rhino Mocks, 346
rich text, Xaml, 114-115
   overflow, 119-120
   parsing, 116-119
RichTextBox, 109, 114
RichTextBoxOverflow, 109
Robbins, John, 615
rows, width, 100
RPC (remote procedural calls), 412
Rx (Reactive Extensions), asynchronous techniques, 459-463
S
safe critical code, 5
sample XAP download service, 37-41
SaveFileDialog, 109
scalability, 558, 578
   concurrency, 590-593
   extremely large data sets, 578-580
   data paging with MVVM, 586-590
   OData, 580-583
   WCF RIA, 583-585
   synchronization, 593-594
script host, elevated trust, 540-546
ScrollViewer, containers (Xaml), 105
   securing websites, REST, 418-421
   separation of concerns, 248
servers, sharing domain objects
   between clients and servers, 401-402
service, 399
service layer, 399
services, web hosts (standard solution), 55-56
services team, 560
shared testing, 338-343
sharing
between Silverlight and the core framework, 48-53
domain objects, between client and server, 401-402

/shared shortcut, 554
signatures, verifying, 492
signing files, 491-494
Silverlight, 6
encrypting files, 494-498
libraries, 5
Sterling, 498-509, 511-514
Unit Testing Framework, 332-338
WPF and, 7-8
Silverlight apps, debugging, 598-600
apps already launched, 608-611
debug symbols, 600-601
debugging tips, 602-607
WinDbg, 611-616
Silverlight development teams, 560
Silverlight project, standard solution, 59-61
Silverlight Spy, 626-627
Silverlight Toolkit, setting up environments, 25
SilverUnit, 346
single responsibility, 260
SketchFlow, 328
Slider, 109
slow concurrency, 593

SOAP (Simple Object Access Protocol), 425-431
sockets, 436-443
S.O.L.I.D. principle, 260-262
Son of Strike, 611
sorts, to-do list application, 471-472
Source, 197
stable composition, 314
MEF, 314-315
StackPanel, layout (Xaml), 101-103
standard solution, 55
Silverlight project, 59-61
web hosts, 55
cross-domain consideration, 56-58
services, 55-56
XAP files, 58-59
XAP files, 61-62
Assemblies, 63-64
manifests, 62-63
state dictionary, 210
states
button groups and, 163
VSM (Visual State Management), 165-170
Xaml, 123, 127-128
StaticResource, markup extensions, 77
Sterling, 414
database definition, 414-416
Silverlight, 498-514
stopping debuggers, 609
storage, isolated storage, 467
storyboards, Xaml, 91-95
StringFormat, 197
StringLengthAttribute, 224
style-binding, 91
styles
  data-binding, 203-207
  Xaml, 86-91
super regions, 396
surface areas, 6
synchronization, 593-594
synchronizing lists, 208-218
syndication items, 620
Syslog protocol, 618

TabControl, 110
target property, 92
TargetNullValue, 197
Task Parallel Library (TPL), 463-465
tasks, asynchronous techniques, 463-465
TdD (Test during Development), 331
Team Foundation System (TFS), 343
template binding, markup
  extensions, 77
templates
  data templates, 128-131
  “Hello, Silverlight” application, 53-55
  Xaml, 123
Test during Development (TdD), 331
test harness, Silverlight Unit Testing
  Framework, 332
testing
  automated testing, 343-345
  automation peers, 367-371
  challenges, 366
  coded UI tests, 364-366
  debug tests, 322
  development tests, 322
integration tests, 322
linked classes, 338-343
local tests, 321
mocking and MEF, 345-352
QA testing, 322
reasons for, 322-323, 327
  documenting code, 324-325
  eliminating assumptions, 323
  extending and maintaining
    applications, 325-326
  improving architecture and
design, 326
  killing bugs at the source, 324
  making better designers, reasons for
testing, 326
shared testing, 338-343
TdD (Test during Development), 331
  Silverlight, 332-338
  unit tests, 321, 327-331
  view model tests, 352-359
  view-logic testing, 247
  Xaml, 359-363
text, Xaml, 114
  character spacing, 120-121
  line height, 121-123
  rich text, 114-120
textBlock, 109
textBox, 109
TFS (Team Foundation System), 343
theme packs, 152
themes
  isolating, 151-155
  resource dictionaries, 203
Timeline class, properties of, 91-92
to-do list application, 468
filters, 471-472
navigation parameters, 472-473
sorts, 471-472
WCF RIA services, 468-470
toast notifications, elevated trust, 531-535
ToDoListItem, 400
toolkit controls, 110-111
ToolTip, 109
TPL (Task Parallel Library), 463-465
trace log levels, 618
tracing, 616-617
client tracing, 617-619
WCF tracing, 619-622
transformation, mapping and, 443
custom type providers, 451-452
dynamic types, 448-450
transitions, VSM (Visual State Management), 170-171
transparent code, 5
TreeView, 111
triggers, 141, 147-149
Expression Blend SDK, 24
troubleshooting
MEF, 313-314
import parameters, 315-316
MEF debugger, 316-318
stable composition, 314-315
VSM, 172-175
trusted, elevated trust, 526-527
application signing, 528-530
in browsers, 552-553
child windows, 535-540
COM interop and script host, 540-546
file system access, 530-531
native Silverlight extensions, 546-547
p/Invoke, 547-551
toast notifications, 531-535
type converters, Xaml, 81-83

U
UI design patterns, MVVM, 246-250
model, 255-262
Model-View-ViewModel pattern, 251-254
view, 262-264
view model, 264
uninstalling OOB apps, 555
unit of work, 592
Unit Testing Framework, Silverlight, 332-338
unit tests, 321, 327-331
updated to-do lists, 402-404
updates
failed updates, 525
OOB apps, checking for, 523-526
UserControl Tag, 200

V
ValidatesOnDataErrors, 197
ValidatesOnExceptions, 197
ValidatesOnNotifyDataErrors, 197
ValidateThat method, 242
validation, data-binding, 220-224
asynchronous validation, 230-232
data error info, 226-230
fluent validation, 232-242
with exceptions, 224-226
validation attributes, 223
ValidationExtensions, 241
ValidationSummary, 111
value converters, Xaml, 84-86
value precedence, dependency
   properties (Xaml), 76
verifying signatures, 492
view, MVVM, 262-264
view model, MVVM, 264
   binding to the view, 265-275
view model locators, MVVM, 265-269
view model tests, 352-359
view-first approach versus view-
   model-first approach, 274-275
view-logic testing, 247
view-model-first approach versus
   view-first approach, 274-275
ViewBox, containers (Xaml), 106-107
ViewModels, 353
views, region management, 397
VirtualizingPanel, layout (Xaml), 103
visual state actions, Expression Blend
   SDK, 24
Visual State Aggregator, 185-194
Visual State Manager. See VSM, 7, 127
Visual Studio
   installing, 22
   profiling applications, 622-625
Visual Studio Test System (VSTS),
   16, 322
Visual Studio tools, setting up
   environments, 22
visual tree, 381
VSM (Visual State Manager), 7, 127,
   161, 251
custom, 176-177
events, 172-175
Expression Blend, 177-185
   groups, 162-165
   overview, 162
   states, 165-170
   transitions, 170-171
   troubleshooting, 172-175
   workflow, 171
VSTS (Visual Studio Test System),
   16, 322

W
WCF (Windows Communication
   Foundation), 4, 424
   services, 55
SOAP, 425-431
WCF RIA, handling data sets, 583-585
WCF RIA Services, 431-435, 468-470
WCF tracing, 619-622
WeakReference class, 385
web hosts, standard solution, 55
   cross-domain considerations, 56-58
   services, 55-56
XAP files, 58-59
Web Platform Installer. See WPI, 22
web service, 399
websites, securing (REST), 418-421
WelcomeContainer class, 49
   Xaml, 69
white box tests, 327
width of rows, 100
Wilcox, Jeff, 332
WinDbg, 611-616
Windows Communication Foundation
   (WCF), 4, 424
   services, 55
SOAP, 425-431
wireframe, 330
workflow
  designer/developer workflow, 559-561
  VSM, 171
WPF, Silverlight and, 7-8
WPI (Web Platform Installer), 22

X
Xaml, 3, 67
  basic controls, 108-109
  containers, 104
    ContentControl, 104-105
    ItemsControl, 105
    ScrollViewer, 105
    ViewBox, 106-107
  dependency properties, 70-74
    attached properties, 75
    value precedence, 76
  design-time extensions, 132-141
  fonts, embedding and distributing, 155-156
interactivity
  behaviors, 141-145
  triggers, 141, 147-149
isolating themes, 151-155
layout, 95-96
  arrange, 96
  canvas, 96
  grid, 97-101
  measure, 96
  StackPanel, 101-103
  VirtualizingPanel, 103
markup and class instantiation, 68-70
markup extensions, 76-81
NUI (natural user interface), 149-151
Parts, 123-126
resource dictionaries, 151-155
states, 123, 127-128
storyboards, 91-95
styles, 86-91
templates, 123
  data templates, 128-131
testing, 359-363
text, 114
  character spacing, 120-121
  line height, 121-123
  rich text, 114-120
type converters, 81-83
value converters, 84-86
Xaml development, 560
Xaml editor, event-handler methods, 89
XAP
downloading, 38
extension XAP, creating, 41-47
XAP download service, sample, 37-41
XAP extensions, tips for, 157-158
XAP files
  extensibility, MEF, 303-305
  standard solution, 61-62
    assemblies, 63-64
    manifests, 62-63
  web hosts, standard solution, 58-59

Y-Z
Y.A.G.N.I (You Aren’t Going to Need It), 258-260