Foreword by Shawn Wildermuth

Designing Silverlight® Business Applications

Microsoft® .NET DEVELOPMENT SERIES

Jeremy Likness

Wintellect®
Know how.

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."

—**Beatriz 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

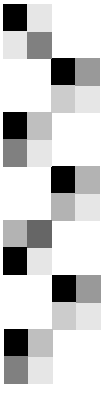
"If you're already familiar with Silverlight, get this book to understand when and how to use proven best practices in your Silverlight line-of-business applications."

—**Jim Wooley**, author, *LINQ in Action*

This page intentionally left blank

Designing Silverlight® Business Applications

This page intentionally left blank



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data is on file.

ISBN-13: 978-0-321-81041-0
ISBN-10: 0-321-81041-4

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing March 2012

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

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.

—Shawn Wildermuth, http://wildermuth.com, AgiliTrain

Preface

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 plugins 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 per-formance 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.

Acknowledgments

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.

7

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 stonecutting. 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).

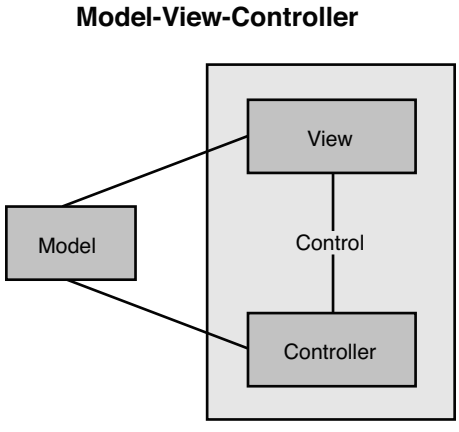


FIGURE 7.1: The Model-View-Controller (MVC) pattern

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.

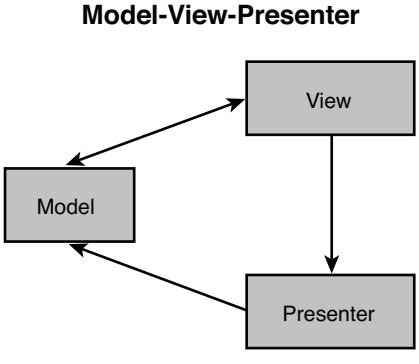


Figure 7.2: The Model-View-Presenter (MVP) pattern

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 patterns, 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 Presentation 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 maintaining 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 updating 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 presentation logic is distributed between the view model and the view. Presentation 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

Misconception	Truth
MVVM is extremely complex.	MVVM can be incredibly simple when implemented correctly.
Code-behind isn't allowed in MVVM.	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.
MVVM is hard to implement.	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 (http://vimeo.com/17926625).
MVVM eliminates the need for value converters.	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.
MVVM reduces the performance of the application.	The improper implementation of any pattern can create performance issues. Proper use of MVVM facilitates unit tests that can help tweak and improve performance.
MVVM is only good for very large projects.	A good MVVM framework coupled with solid understanding is just as suitable for small projects as it is big ones.
MVVM is about commands and messaging systems.	MVVM simply specifies the responsibilities of various modules within the code. Commands, messaging frameworks, and other constructs are just helpers and building blocks.
MVVM is hard to understand.	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.

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:

```
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:

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

```
    var callback = e.UserState as Action<IEnumerable<WidgetItem>>;

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

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.

```
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:

```
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 7.2: The View in MVVM

Component	Description
Xaml	Declarative markup to provide layout, controls, and other components that make up a screen
Value converters	Special classes used to transform data to a user element type and back

Component	Description
Data templates	Templates that map data elements to controls
Visual state groups	Named states that impact the properties of various elements to provide a physical state based on the logical states of controls
Storyboards	Animations and transitions
Behaviors	Reusable algorithms that can be applied to various controls
Triggers	Algorithms that can be applied to controls and invoked based on configured events
Code-behind	Extensions of the Xaml markup to perform additional UI-specific tasks

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 `ViewModelLocators`. Specify a grid with four equally sized quadrants (two rows and two columns). Provide an interface for the view model:

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

Now implement a design-time view model:

```
public class DesignViewModel : IViewModelInterface
{
    public List<string> Shapes
    {
        get
        {
```

```
                return new List<string>
                            {"Circle", "Square", "Rectangle"};
            }
        }

        public string SelectedShape
        {
            get { return "Circle"; }
        }
    }
```

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

```
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);
            }

        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

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:

```
<UserControl.Resources>
    <ViewModelLocators:ViewModelLocator x:Key="Locator"/>
</UserControl.Resources>
<Grid x:Name="LayoutRoot" Background="White"
        DataContext="{Binding Source={StaticResource Locator},
Path=ViewModel}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <ComboBox ItemsSource="{Binding Shapes}"
              SelectedItem="{Binding SelectedShape,
➥Mode=TwoWay}"/>
</Grid>
```

Now define the shapes and place them in the second row of the `LocatorControl`:

```
<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.

```
<VisualStateManager.VisualStateGroups>
   <VisualStateGroup x:Name="ShapeGroups">
      <VisualStateGroup.States>
         <VisualState x:Name="Circle">
            <Storyboard>
               <ObjectAnimationUsingKeyFrames
                   Storyboard.TargetName="CircleShape"
                   Storyboard.TargetProperty="(UIElement
➥.Visibility)">
                  <DiscreteObjectKeyFrame KeyTime="0:0:0">
                     <DiscreteObjectKeyFrame.Value>
                        <Visibility>Visible</Visibility>
                     </DiscreteObjectKeyFrame.Value>
```

```
                </DiscreteObjectKeyFrame>
              </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:

```
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 delegate. 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:

```
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:

```
public partial class ControllerControl
{
    public Controller Controller { get; set; }
```

```
        public ControllerControl()
        {
            InitializeComponent();
            Controller = new Controller(this);
        }
    }
```

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:

```
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

```
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:

```
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:

```
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

```
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();
            }
```

```
            var vm
                = new ViewModel
                        {
                            GoToVisualState =
                                state =>
                                VisualStateManager.GoToState(
                                    targetControl, state, true)
                        };


            return vm;
        }
        return null;
    }
}
```

With the extension created, you can add a namespace reference in the control:

```
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.

```
<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

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y-Z