



Professional Expertise Distilled

# MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF

Eliminate unnecessary code by taking advantage of the MVVM pattern—less code, fewer bugs

*Foreword by Markus Egger*

*Publisher, CODE Magazine*

*President and Chief Software Architect, EPS Software Corp.*

*Microsoft Regional Director and MVP*

**Ryan Vice**  
**Muhammad Shujaat Siddiqi**

**[PACKT]** enterprise   
PUBLISHING professional expertise distilled

# MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF

Eliminate unnecessary code by taking advantage  
of the MVVM pattern—less code, fewer bugs

**Ryan Vice**

**Muhammad Shujaat Siddiqi**



BIRMINGHAM - MUMBAI

# MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2012

Production Reference: 1010812

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-84968-342-5

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Tony Shi ([shihe99@hotmail.com](mailto:shihe99@hotmail.com))

# Credits

**Authors**

Ryan Vice  
Muhammad Shujaat Siddiqi

**Reviewer**

Kanishka (Ken) Abeynayake

**Acquisition Editor**

Dhwani Devater

**Lead Technical Editor**

Dhwani Devater

**Technical Editors**

Felix Vijay  
Manasi Poonthottam  
Lubna Shaikh

**Copy Editors**

Brandt D'Mello  
Laxmi Subramanian  
Alfida Paiva

**Project Coordinator**

Abhishek Kori

**Proofreader**

Lesley Harrison

**Indexer**

Rekha Nair

**Graphics**

Manu Joseph

**Production Coordinator**

Melwyn D'sa

**Cover Work**

Melwyn D'sa



# Foreword

Rich client development remains one of the most popular forms of application development, both from a user and a developer point of view. While nobody denies the importance of thin-client interface technologies such as HTML(5), it is clear that consumers and enterprises alike enjoy using applications that provide a rich, powerful, productive, and sometimes fun experience. Evidence ranges from the current App Craze on mobile devices to the long-running history of rich business applications deployed by many businesses of all sizes. Many of the most successful applications and systems, measured in commercial success and/or popularity, are either entirely based on Rich Client technology or make Rich Clients part of the mix.

If you are a Microsoft developer (and if you are reading this book, the chances are that you are), you find yourself in the lucky position of getting a chance to use one of the best, if not *the* best, sets of Rich Client development technologies and tools. The paradigm first introduced by WPF (then known under its *Avalon* code name) and the XAML declarative approach have turned out to be a super-productive, highly maintainable, and highly reusable approach. The technologies are easy to use once the developer gets acquainted with the ideas behind the setup of XAML-based systems. It is true that there is a learning curve. As an industry, we have used the same UI development paradigm across many languages, systems, and even platforms for a very long period of time, reaching back all the way to MS DOS. The *drop a control on a form, set a few properties, and wire up some event handlers* approach can be found almost universally in pre-XAML scenarios ranging from Visual Basic, to C++, PowerBuilder, Delphi, Visual FoxPro, .NET Windows Forms, ASP.NET WebForms, even standalone HTML scenarios, and many more. XAML breaks that mold. Yes, you can still employ the old paradigm, but you can reap significant benefits by following the new ideas. By reading this book, you are well on your way down that path, and you will find that while there is a hump in the learning curve you need to get over, there also is a significant downward slope on the other side of that hump. While many environments retain a high level of difficulty even once you achieve a high degree of familiarity, WPF is different in that things tend to be pretty straightforward once you know how to do things the right way.

WPF has become the de-facto standard for Windows Desktop Application development. It is now a well-established technology that has superseded the older Windows Forms (WinForms) framework. Microsoft uses WPF in many of its own products and WPF has been continually developed for a number of years and across a number of versions and major releases. While other development environments may be flashier, and technologies like HTML5 get the limelight, I can tell based on personal experience that WPF seems to be a secret hot technology. This may be anecdotal evidence based on my own experiences only, but my experience draws on my interactions not just with our consulting and custom software customers, but also on the interactions with a hundreds of people who attend training classes we teach, thousands of people I interact with at various developer events, and the tens of thousands of people I interact with one way or another as readers of CODE Magazine.

In short, WPF is a very popular development environment that is used for a large number of highly strategic development projects. WPF developers are also highly sought after. While there may not be a need for as many WPF developers as there is for HTML developers, the demand for WPF developers is much higher. In other words, while the world generally needs more HTML developers and designers than WPF equivalents, there is no shortage of those HTML skills. I do not mean to take anything away from the many highly skilled HTML experts (and the same goes for many other platforms and technologies). However, those skills are relatively easily available. WPF skills, on the other hand, are much harder to come by and thus represent a more valuable expertise. Skilled WPF developers routinely command a higher salary or hourly rate. A fact you are probably happy to learn if you are interested in reading this book. ;-)

While this book focuses on WPF, many of the things you learn here will serve you well beyond WPF. The XAML Paradigm is of course used in other environments. Silverlight in its original form as a browser plugin is one such example that has grown out of WPF. While browser plugin technology may have seen its best days as far as strategic importance goes, Silverlight still goes down in history as one of the fastest growing and most rapidly adopted developer technologies ever. Silverlight will also be here to stay for some time to come. While I would not recommend starting new projects in Silverlight unless you have a very good and specific reason to do so, you are probably OK using Silverlight for a bit longer if you have already travelled down that path. For new projects, however, I would recommend WPF.

It is important to remember that the ideas behind Silverlight are not just useful in browser plugins. Silverlight for Windows Phone is turning out to be a beautiful and highly productive development environment embraced by developers. For mobile development, one first chooses the platform of course. If that platform is iOS, Apple's development environments and languages are a given. If the platform is Android,

one probably ends up with Java. It is too bad one cannot choose Microsoft's version of Silverlight for Windows Phone to develop on any of these other mobile platforms, because I would personally choose it any day over any of the other options based on pure productivity and development joy.

And the story continues. XAML is used as one of the cornerstones in Windows 8's new Metro user interface mode. So everything you learn in this book will be of use to you in the bold new world of Windows 8 development as well. Windows 8 Metro also supports a proprietary development model based on HTML5 and JavaScript, which will be on equal footing with XAML. The jury is still out and it is too early to tell (as I am writing these lines, we are still at least a half a year away from the Windows 8 ship date) but based on what we see at events and from readership reactions through CODE Magazine, people seem to be most interested in the XAML development option. A biased result perhaps (after all, current WPF and Silverlight developers are probably most likely to be the first ones in as far as Metro development goes), but it is still interesting to see that XAML development is alive and well, and expected to enjoy a bright future.

Microsoft is planning to ship Windows 8 with two modes; one known as Metro as well as the more conventional Desktop mode, which largely resembles Windows 7's desktop. Which brings us right back to WPF, because all WPF applications will continue to work just fine in Windows 8's Desktop mode. Either way you turn it, the XAML family of technologies is not a bad family to be part of. We are certainly very happy to base a lot of our efforts on these technologies and have a high degree of comfort moving forward with that approach.

But not all WPF development is created equal. There are a lot of different scenarios and approaches. Some good, some bad. One approach may work well in some scenarios while it doesn't work well at all in others. As in all engineering disciplines, knowing the pros and cons of each tool in the toolbox is an important aspect of engineering know-how. With that said however, it is clear that MVVM is a very valuable pattern for a lot of WPF-based applications (and XAML-based applications, in general). If done right, MVVM leads to a range of different advantages ranging from quality to maintainability, reusability, even developer productivity, and more. As with most powerful tools, the power can be wielded both for good and evil. Yes, it is possible to create horrible monstrosities that are hard and slow to develop and result in inflexible and slow applications. If that is the outcome, the developers and architects did a bad job in evaluating the tools at their disposal and made ill-advised choices in how to wield them. Luckily, the book you are currently reading is going to be a valuable first step in learning how to avoid such mistakes and instead unleash the incredible power of MVVM and many of the associated techniques.



Explaining those details is a task I will leave in the capable hands of the authors of this book. It is my hope that reading it is going to be just one of the many steps in your journey of building XAML-based applications for a long time to come. After all, as a User Interface development and design enthusiast, I can't imagine a UI development environment that is more beautiful and elegant than WPF and XAML.

**Markus Egger**

Publisher, CODE Magazine

President and Chief Software Architect, EPS Software Corp.

Microsoft Regional Director and MVP

# About the Authors

**Ryan Vice** is a Microsoft enterprise developer with over 12 years of experience. He lives in Austin, TX with his wife and family, and works as an independent consultant . He has experience creating solutions in numerous industries including network security, geoseismic, banking, real estate, entertainment, finance, trading, construction, online retail, medical, and credit counseling. He has done projects for companies of all sizes including high-volume applications for large fortune 500 companies like Dell and Charles Schwab. He frequently presents sessions at users groups and conferences throughout Texas including Houston Tech Fest and Dallas Day of .NET. He was awarded Microsoft MVP for Connected Systems in 2010, 2011, and 2012. He has also been an MSDN Moderator. His current areas of focus are around MVVM, WPF, XAML, IoC, NHibernate, and Windows 8 Metro.

**Muhammad Shujaat Siddiqi** has been serving the Enterprise Software Industry for more than seven years in Pakistan and USA. He has a bachelor's degree in Computer and Information Systems (BE) from NED University, Karachi. He is a passionate blogger. For his services to WPF development community, Microsoft awarded him MCC in 2011. He is a student of the Shaolin-Do form of martial arts.

# About the Reviewer

**Kanishka (Ken) Abeynayake** has been dabbling in personal computers from their infancy starting out as an Apple and Mac developer. He authored the original Internet suite included with Delphi and CBuilder, and is a Consultant at Sogeti consulting for Fortune 500 companies, such as Dell and Microsoft. When he is not playing around with the latest Microsoft technologies, he and his wife are enjoying their passion for travelling. Kanishka obtained his education from the University of Sri Lanka Moratuwa and the University of Texas. He can be contacted at [ken@lionknight.com](mailto:ken@lionknight.com).

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

## Instant Updates on New Packt Books

Get notified! Find out when new books are published by following [@PacktEnterprise](https://twitter.com/PacktEnterprise) on Twitter, or the *Packt Enterprise* Facebook page.



*To my wife, Heather, my daughter, Grace and my two sons, Dylan and Noah; the time away from you was the hardest part of writing this book. Thanks for all your love and support.*

*-Ryan Vice*

*I dedicate this work to my amazing parents.*

*-Muhammad Shujaat Siddiqi*



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Presentation Patterns</b>	<b>7</b>
<b>The Project Billing sample application</b>	<b>8</b>
Types of state	10
<b>History of presentational patterns</b>	<b>11</b>
Monolithic design	11
The problems with monolithic design	12
Data service stub	14
Monolithic Project Billing sample	17
ProjectsView	19
Running the sample	26
Takeaways	27
Rapid application development	28
RAD Project Billing sample	28
Takeaways	39
<b>MVC</b>	<b>40</b>
View	40
Controller	41
Model	41
<b>Layered design</b>	<b>42</b>
The layers	42
<b>MVC with layered design</b>	<b>43</b>
<b>MVC Project Billing sample</b>	<b>44</b>
Model	46
Controller	48
View	49
How it works	55
Takeaways	57
Memory leaks	57
<b>MVP</b>	<b>60</b>
<b>MVP Project Billing sample</b>	<b>61</b>



Model	62
View	64
Presenter	69
Main window	72
How it works	74
Takeaways	74
<b>Summary</b>	<b>75</b>
<b>Chapter 2: Introduction to MVVM</b>	<b>77</b>
<b>History</b>	<b>77</b>
<b>Structure</b>	<b>80</b>
Pure MVVM	80
View	81
View Model	81
<b>WPF and Silverlight enablers</b>	<b>82</b>
Dependency Properties	82
Dependency property inheritance	83
Rich data binding	86
INotifyCollectionChanged and ObservableCollection<>	87
Automatic dispatching	88
Triggers	88
Styles	89
Control Templates	90
Data templates	90
Commands	91
<b>MVVM project billing sample</b>	<b>93</b>
MVVM design	93
View Models	94
Model	96
Code	96
ProjectsModel	97
ProjectViewModel	100
ProjectsViewModel	102
WPF UI	110
Silverlight UI	115
<b>Benefits of MVVM</b>	<b>125</b>
MVVM and humble views	127
<b>Issues and pain points of MVVM</b>	<b>128</b>
<b>MVVM Light</b>	<b>129</b>
<b>Summary</b>	<b>130</b>
<b>Chapter 3: Northwind—Foundations</b>	<b>131</b>
<b>Northwind requirements</b>	<b>132</b>
<b>Presentation tier foundation</b>	<b>133</b>
Locator pattern	136

---

<b>Data access tier</b>	<b>137</b>
<b>Listing the customers</b>	<b>142</b>
Unit testing getting customers	145
Using an isolation framework	151
<b>Adding tabs</b>	<b>154</b>
<b>Viewing customer details</b>	<b>159</b>
Viewing details for one customer	160
Testing CustomerDetailsViewModel	165
<b>Wiring up the customer list box</b>	<b>167</b>
Testing ShowCustomerDetails()	172
<b>Summary</b>	<b>174</b>
<b>Chapter 4: Northwind—Services and Persistence Ignorance</b>	<b>175</b>
Adding a Service Layer	176
Integrating the Service Layer	181
<b>Persistence ignorance and custom models</b>	<b>186</b>
Trade-offs of generated models	186
Adding persistence ignorance	187
Adding unit tests	192
<b>Summary</b>	<b>201</b>
<b>Chapter 5: Northwind—Commands and User Inputs</b>	<b>203</b>
<b>Pure MVVM</b>	<b>203</b>
<b>Making it easier with frameworks</b>	<b>208</b>
<b>Updating customer details</b>	<b>210</b>
Testing and updating customer details	214
<b>Gestures, events, and commands</b>	<b>216</b>
InputBindings	217
KeyBinding	218
MouseBinding	219
Using code behind	220
Event to command	221
Attached Behavior	222
Using MVVM Light	226
<b>Summary</b>	<b>228</b>
<b>Chapter 6: Northwind—Hierarchical View Model and IoC</b>	<b>229</b>
<b>Adding orders to customer details</b>	<b>229</b>
Service layer	231
Application layer	236
Presentation layer	241
View Models	242
Views	245
Take aways	247

<b>Viewing order details</b>	<b>247</b>
ToolManager	248
Inversion of Control frameworks	255
IoC designs	255
Adding an IoC container to Northwind	258
Order details	271
<b>Summary</b>	<b>280</b>
<b>Chapter 7: Dialogs and MVVM</b>	<b>281</b>
<hr/>	
<b>Should we make a compromise?</b>	<b>282</b>
<b>Dialog service</b>	<b>282</b>
Using DataTemplates with DialogService	286
Convention over configuration	294
<b>Mediators</b>	<b>296</b>
<b>Attached behaviors</b>	<b>306</b>
<b>Summary</b>	<b>310</b>
<b>Chapter 8: Workflow-based MVVM Applications</b>	<b>311</b>
<hr/>	
WF for business rules execution	312
Handling delays in rules execution	322
WF for controlling application flow	327
<b>Summary</b>	<b>332</b>
<b>Chapter 9: Validation</b>	<b>333</b>
<hr/>	
<b>Validations and dependency properties</b>	<b>333</b>
<b>Error templates</b>	<b>334</b>
<b>Validation in MVVM-based applications</b>	<b>342</b>
Validation rules	342
Using validation rules	342
Specializing validation rules—supporting parameters	344
Validation rules and converters	345
Validation mechanism in WPF and Silverlight	349
IDataErrorInfo	350
Validation states	359
Limitations and gotchas	374
INotifyDataErrorInfo	374
Enterprise library validation application block	389
Complex business rules	398
<b>Error notifications</b>	<b>398</b>
Error message box	398
Highlighting fields	400
Error messages in the tooltip	400
Error messages beside the control	400

---

Validation summary pane	401
Flip controls	402
<b>Summary</b>	<b>402</b>
<b>Chapter 10: Using Non-MVVM Third-party Controls</b>	<b>403</b>
Using attached behaviors	405
Using binding reflector	411
readonly CLR properties (with no change notification support)	416
Using .NET 4.0 dynamic	421
Using MVVM adapters	426
Summary	429
<b>Chapter 11: MVVM and Application Performance</b>	<b>431</b>
<b>Asynchronous binding</b>	<b>431</b>
<b>Asynchronous View Model construction</b>	<b>435</b>
<b>Priority binding</b>	<b>437</b>
<b>Virtualization and paging</b>	<b>440</b>
<b>Using BackgroundWorker</b>	<b>441</b>
<b>Targeting system configuration</b>	<b>442</b>
<b>Event Throttling</b>	<b>442</b>
<b>Lazy Initialization</b>	<b>443</b>
<b>Summary</b>	<b>449</b>
<b>Appendix A: MVVM Frameworks</b>	<b>451</b>
<b>Appendix B: Binding at a Glance</b>	<b>453</b>
<b>Basics</b>	<b>453</b>
<b>Validation</b>	<b>453</b>
ValidationRules	453
IDataErrorInfo	454
INotifyDataErrorInfo [.net 4.5]	454
Enterprise Library 5.0 Validation Application Block	454
Windows WF	454
Validation.ErrorTemplate	454
<b>Static properties/fields</b>	<b>454</b>
<b>Executing code in DataContext</b>	<b>454</b>
<b>Binding to DataContext[DC]</b>	<b>455</b>
<b>Resources</b>	<b>455</b>
Types with default constructor	455
XmlDataProvider	455
ObjectDataProvider	455
<b>Binding to resource</b>	<b>456</b>
Static resource	456
Dynamic resource	456

<b>Updating source</b>	<b>456</b>
Binding.UpdateSourceTrigger	456
Binding.Delay: [.net 4.5] [Binding.Mode:TwoWay / OneWayToSource ]	456
<b>Mode [Binding.Mode] [T:Target, S:Source]</b>	<b>457</b>
<b>Binding to other elements in the view</b>	<b>457</b>
ElementName	457
RelativeSource	457
<b>Conversion</b>	<b>457</b>
Binding.StringFormat [SF]	457
Converter [C]	458
<b>Performance</b>	<b>458</b>
Async binding	458
ObjectDataProvider.IsAsynchronous	458
PriorityBinding	458
<b>Index</b>	<b>459</b>

---

# Preface

MVVM (Model View View Model) is a Microsoft best practices pattern for working in WPF and Silverlight that is highly recommended by both Microsoft and industry experts alike. This book will look at the reasons for the pattern still being slow to become an industry standard, addressing the pain points of MVVM. It will help Silverlight and WPF programmers get up and running quickly with this useful pattern.

*MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF* will help you to choose the best MVVM approach for your project while giving you the tools, techniques, and confidence that you will need to succeed. Implementing MVVM can be a challenge, and this book will walk you through the many issues you will come across when using the pattern in real world enterprise applications.

This book will help you to improve your WPF and Silverlight application design, allowing you to tackle the many challenges you will face in creating presentation architectures for enterprise applications. You will be given examples that show the strengths and weaknesses of each of the major presentation patterns. The book then dives into a full 3 tier enterprise implementation of MVVM and takes you through the various options available and the trade-offs for each approach. During your journey you will see how to satisfy many of the challenges of modern WPF and Silverlight enterprise applications including scalability, testability, and extensibility.

Complete your transition from ASP.NET and WinForms to Silverlight and WPF by embracing the new tools in the Silverlight and WPF platforms, and the new design style that they allow for. This book will get you up to speed and ready to take advantage of these powerful new presentation platforms.

## What this book covers

*Chapter 1, Presentation Patterns*, gives the reader an example-driven overview of the history of presentation patterns. We will implement a Project Billing sample application using various approaches including MVC and MVP. Along the way, we will look at the issues with each pattern that motivated the next pattern in the evolutionary chain. This chapter also demonstrates how presentation patterns that require .NET events, such as MVC and MVP, can cause memory leaks if not properly implemented. This chapter will leave the reader with the knowledge needed to discuss the tradeoffs of the various presentation patterns and allow the reader to answer question like why use *MVVM over MVP or MVC*.

*Chapter 2, Introduction to MVVM*, covers the various features of WPF and Silverlight that make MVVM an attractive option on these platforms. We will follow this by re-implementing the Project Billing sample application from the first chapter using MVVM. We will then look at some of the benefits and cost of using MVVM. We will finish off the chapter by taking a quick look at the MVVM Light open source framework that will be used throughout the book.

*Chapter 3, Northwind – Foundations*, will walk through how to lay the foundation of the Northwind application that we will build over the next four chapters. We will wire up the Northwind database using Entity Framework and see how Entity Framework integrates with the binding systems in WPF and Silverlight to provide change notifications. We will also add unit tests that allow us to see how MVVM allows us to test all of our view logic.

*Chapter 4, Northwind – Services and Persistence Ignorance*, will have us attempting to make our application more scalable by adding a WCF service layer between the Presentation Layer and the Application Layer. We will see how WCF integrates with the binding system in both WPF and Silverlight to provide change notifications. We will also look at the benefits and cost of implementing a Persistence Ignorant Presentation Layer.

*Chapter 5, Northwind – Commands and User Inputs*, discusses the benefits of taking advantage of the commanding system in WPF and Silverlight to implement MVVM using the pure approach.

*Chapter 6, Northwind – Hierarchical View Model and IoC*, explains the power and productivity that can be added by using the Hierarchical View Model approach to MVVM. We will also see how to implement an Inversion of Control framework using IoC best practices by updating our application to use the Ninject for IoC framework.

*Chapter 7, Dialogs and MVVM*, discusses the various options for showing modal and modeless dialogs. It also discusses how data can be shared across the dialogs that we will create.

*Chapter 8, Workflow-based MVVM Applications*, explains how we can use Windows WF to control the flow of the user interface. It would also be touching the area of business rules validation using WF including the discussion about slow executing workflows.

*Chapter 9, Validation*, discusses the various techniques for data entry and business rules validation. The chapter will also be shedding some light on how the results of these validations can be displayed to the user.

*Chapter 10, Using Non-MVVM Third-party Controls*, will focus on the discussion regarding the usage of non-MVVM based controls in your MVVM based design to improve the testability of our code base.

*Chapter 11, MVVM and Application Performance*, explains some features of XAML frameworks targeting for better application performance.

*Appendix A, MVVM Frameworks*, outlines the basic features to look for before selecting an MVVM framework or toolkit. It also lists the available MVVM frameworks popular in the industry.

*Appendix B, Binding at a Glance*, summarizes the Binding System infrastructure, which makes MVVM possible in WPF and Silverlight.

## What you need for this book

- Microsoft Visual Studio 2010 Service Pack 1
- Rhino Mocks
- .NET Framework 4 Platform Update 1 for *Chapter 8, Workflow-based MVVM Applications*

## Who this book is for

This book will be a valuable resource for Silverlight and WPF developers who want to fully maximize the tools with recommended best practices for enterprise development. This is an advanced book and you will need to be familiar with C#, the .NET framework, and Silverlight or WPF.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.



Code words in text are shown as follows: "You should now be able to execute `ICustomerService.GetCustomers()` from WCF Test Client."


A block of code is set as follows:

```
public class RepositoryRegistry : Registry
{
    public RepositoryRegistry()
    {
        For<IUIDataProvider>()
            .Singleton();
        For<ICustomerService>()
            .Singleton()
            .Use(() => new CustomerServiceClient());
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class OrderViewModel : ViewModelBase
{
    public const string ModelPropertyName = "Model";
    private Order _model;
    public Customer Customer { get; set; }
    private readonly IToolManager _toolManager; exten =>
        i, 1, Voicemail(s0)
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "This will add a **Show Details** link to our grid".

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on [www.packtpub.com](http://www.packtpub.com) or e-mail [suggest@packtpub.com](mailto:suggest@packtpub.com).

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## **Piracy**

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## **Questions**

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1

## Presentation Patterns

*By Ryan Vice*

**Separation of Concerns** or **SoC** is a core principle of enterprise software development which provides many benefits and has been a key driving force behind many presentation (or UI) design patterns that have emerged over the last 30 years. In the arena of **Silverlight** and **WPF** development, **Model View View Model** or **MVVM** has quickly become the de-facto pattern for achieving SoC in UIs. However, this pattern often leaves developers and architects frustrated and at the time of this writing, can be difficult to implement in an effective way that provides more benefits than some of the older, more familiar **presentation patterns** (**MVC**, **MVP**, and so on).

In this chapter we will cover the evolution of presentational patterns along with the problems that are solved by each pattern along the evolutionary path. We will also dive into the shortcomings of each pattern which led to the next pattern in the evolution and will finish this chapter ready to look at MVVM.

We will begin this chapter by reviewing the functionality of the Project Billing sample application that we will use throughout this book. We will follow this by briefly talking about the various types of state that must be managed in UI applications. Then we dive into the history of presentational patterns and as we go through the history we will implement *Project Billing* using each pattern to show you explicitly the benefits and the shortcomings of each pattern that lead to the next pattern in the evolution. This will help you understand why you'd want to use MVVM through examples and make the benefits of MVVM easier to appreciate when we dive into that topic in the next chapter. This would also help you evangelize the pattern on your projects if needed and be able to explain what benefits MVVM would offer over other presentation patterns.

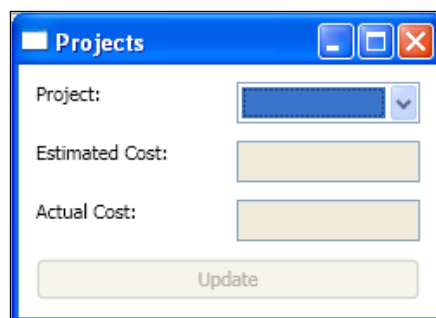
If you are already familiar with (or not interested in) the history of presentation patterns, you should still at a minimum review the following sections:

- **The Project Billing sample application:** This section will review the functionality of the sample application that will be used in the first two chapters
- **Types of state:** This section defines and discusses the various types of state that need to be managed in a UI application
- **Monolithic design:** The introduction of this section discusses the coupling that results from not using some kind of presentational design pattern
  - **The problems with Monolithic design:** This section discusses the many problems that result from not using presentational design patterns
- **Data service stub:** This section covers creating the data service stub that will be used by the Project Billing application throughout this book
- **Memory leaks:** This section covers how .NET events can cause memory leaks

However, I'd recommend that unless you are intimately familiar with patterns such as **Model 2** and **Passive View** that you take the time to go through this chapter as this knowledge will be very useful in driving home some of the fundamentals of presentation patterns which will help you adapt these notoriously flexible patterns to your needs

## The Project Billing sample application

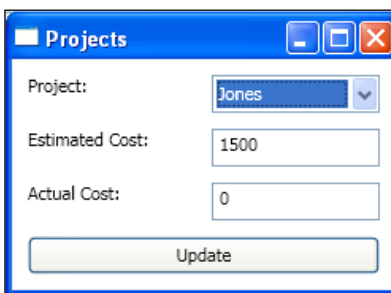
Let's start off by walking through the functionality of the Project Billing application. Project Billing is a contrived application that – as the name suggests – allows for simple project billing. The application's UI is shown in the following screenshot:



The application consists of a simple master/details form for the main window. At the top of the application is a list of projects that when selected make up the master of the master/detail relationship. Following the projects come the details which include the following:

- **Estimated Cost**
- **Actual Cost**

Notice how all the details are disabled along with the **Update** button. Whenever a user selects a project from the list, the UI is updated so that all of the details controls are enabled as shown in the following screenshot:

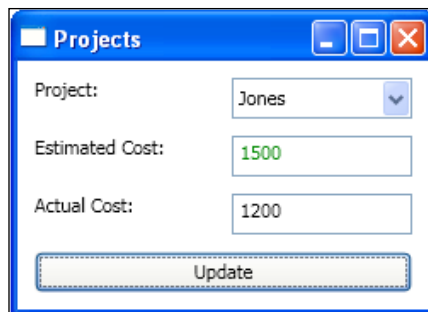


The screenshot shows a window titled "Projects" with a blue header. It contains a dropdown menu for "Project:" with "Jones" selected. Below it are two text input fields: "Estimated Cost:" with the value "1500" and "Actual Cost:" with the value "0". At the bottom is a disabled "Update" button.

Now a user can update any of the details they like. If the user sets a value for **Actual Cost** that is lower than the **Estimated Cost** for the selected project and clicks the **Update** button, the **Estimated Cost** will be displayed in green.



The following screenshot shows Project Billing with an **Actual Cost** that is lower than the **Estimated Cost**; however, this book is not in color and so you will have to run any of the sample implementations of Project Billing in this book to see the color of estimated cost change.



The screenshot shows the same "Projects" window. The "Project:" dropdown still shows "Jones". The "Estimated Cost:" field now contains "1500" and is highlighted in green. The "Actual Cost:" field now contains "1200". The "Update" button is now enabled and has a dashed border.



This is a contrived example and doesn't have validations or robust error handling, so entering invalid values for actual cost can cause problems for the application. However, we will explore validations later in this book.

Putting in a value that is above the estimated value will cause the **Estimated Cost** to be displayed in red. You can also:

- Change the **Estimated Cost**.
- Click on the **Update** button, then change your selection and when you reselect the updated project you will see that your new values have been maintained in the view state.
- After updating a project, you can also open a second **Projects** view and see that the data is synchronized (session state). This is not supported in all versions of Project Billing but only in those versions whose architecture supports easily sharing session state.

It's a very simple example but complex enough to demonstrate the various types of state and logic that need to be managed by a UI application and to show how well the various patterns handle each type of state and logic.

## Types of state

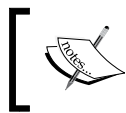
The Project Billing application demonstrates all three types of state that must be managed in all UI applications.

- **View state: UI state** or view state is the state of the UI which includes the data being displayed that was provided by the model but could also include things like what buttons are disabled and the color changes that may have been applied to text. The disabling of the details controls and changing the color of **Estimated Cost** in Project Billing are examples of types of *view state*.



You may be familiar with the concept of view state from working in ASP.NET where the view state is stored in a hidden field in the HTML and accessible server-side via the `ViewState` collection.

- **Session state:** It is the state of the data that has been retrieved from the persistence store and is being held in memory. This data could be accessed by multiple components in the application and remains in memory only until the user terminates their session or until it is persisted. In Project Billing, any changes that are made to project details become session state once you click on the **Update** button.
- **Persisted state:** It is the state of the applications data that has been retrieved from or is persisted to some sort of repository such as a database, service or XML file. In Project Billing, the data that is mocked in the `DataService` is an example of persisted state.



Project Billing uses a data service stub that returns fake data and doesn't demonstrate real persistence. Persistence will be covered in *Chapter 3, Northwind – Foundations*.

## History of presentational patterns

In this section we will cover the history of presentational (or GUI) patterns. Presentational patterns have been around for over 30 years and a full coverage of all the various patterns is outside of the scope of this book. We will instead focus on two of the major trends that have emerged over the last 30 years and look at how those two trends eventually evolved to MVVM for Silverlight and WPF.



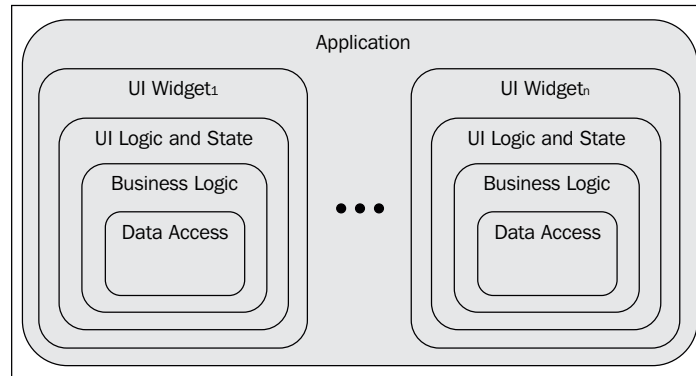
If you are interested in learning more about the history of presentational patterns than what is covered here, then see Martin Fowler's article *GUI Architectures* (<http://martinfowler.com/eaDev/uiArchs.html>).

## Monolithic design

Enterprise applications deal with displaying, manipulating, and saving data. If we build enterprise applications with no design so that each GUI component is coupled all the way down to the data access code, then there are a lot of problems that can emerge.



This style of design is called **monolithic** and the following diagram shows the coupling that exists under monolithic designs:



## The problems with monolithic design

In this section we will review the problems caused by the tight coupling and low cohesion found in monolithic designs.

### Code maintenance

Looking at the previous screenshot if you assume that UI Widget<sub>1</sub> and UI Widget<sub>n</sub> are using the same business logic, then using a monolithic design will cause code duplication. Every time a change needs to be made to the business logic, it would need to be made in both places. This is the type of issue that is solved by SoC and one of the motivators for design paradigms like 3-tier which we will look at in the *Layered design* section later in this chapter.

### Code structure

Not having the code structured into reusable components and well-organized layers makes things like sharing session state difficult under monolithic design. As you will see in the examples that follow, once we move to MVC and MVP, there are many benefits including:

- The session state becomes much easier to manage and share
- Code is easier to reuse
- Code is well-organized and easier to understand and maintain

- Code scales easier as you can build components into separate DLLs for distributed deployment
- Code is more extensible as you can replace components to provide different behaviors

## Code testability

Creating code that can be effectively tested with unit tests requires designing for testability. The monolithic approach poses several problems for code testability including:

- **Poor isolation of tests:** One of the core principles of unit testing is *isolation of the tests*. You want your unit tests to test one scenario of one method of one class and not to test the dependencies. Following this principle makes your tests more valuable because when a test fails it's more likely that developers who didn't write the test but introduced the change that broke the test will fix the issue. This is because it will be very easy for the developer to determine what the problem was that broke the test because it's so isolated and clear in its purpose. A big part of getting return on investment from unit tests comes from making them easy for developers to use and avoid making your unit tests high maintenance. With high-maintenance unit tests the developers might just delete, disable, or comment out the test instead of fixing the problem, which makes the expense that was put into creating the test a waste.
- **Testing the UI is difficult:** Using automated testing to test the UI is notoriously difficult. Monolithic design makes this problem worse as there is no separation between the UI and the rest of the layers of logic. One of the major contributors to the need of separated UI patterns is the desire to move as much logic as possible out of the UI and into separate testable components.
- **Poor code coverage:** Code coverage refers to how much of your code is covered by unit tests. Generally speaking, the more code you have covered by tests, the more stability you will create in your development process, and the more benefits you will reap from your tests. High code coverage provides fewer bugs and quicker refactoring times. When you create a monolithic application, it affects your ability to achieve high code coverage levels, because you can't test the UI logic and the coupling between the various layers as it makes mocking dependencies difficult, prohibiting creation of unit tests.



100 percent test coverage is not always the best level of coverage as too much coverage can make the code brittle to change and make the code high maintenance. My general rule of thumb is that I want to test the functionality that is defined by the public interface of the class under test. Testing internal details that could change can provide more inconvenience than benefit. However, this rule of thumb assumes that you have a good separation of concerns and have applied the **Single Responsibility Principle** to the design of your application. Single Responsibility Principle is part of the **SOLID** design principles and more details about SOLID are easily found online if needed.

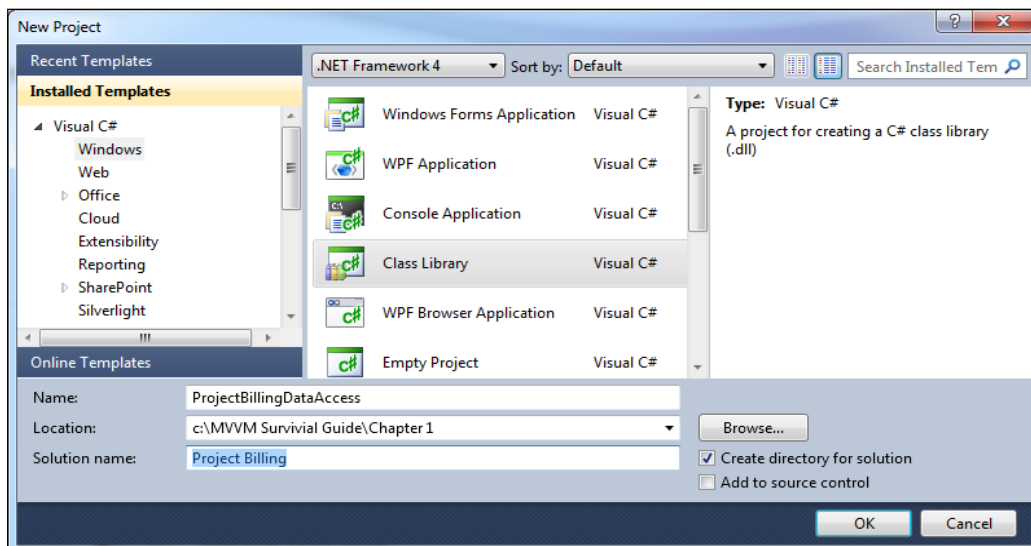
## Data service stub

We will be using a data service stub as part of our **data layer** to take the place of a real data service in our sample applications so that we can focus on presentation patterns and not on data access patterns and techniques.



Data layer will be explained in the *Layered design* section later in this chapter.

Let's start by creating a new **Class Library** project called **ProjectBilling.DataAccess** in a solution called **MVVM Survival Guide** as shown in following screenshot:



Now delete the `Class1.cs` file that is created by default by the project template and add a new class called `Project` and add the following code to `Project.cs`:

```
namespace ProjectBilling.DataAccess
{
    public interface IProject
    {
        int ID { get; set; }
        string Name { get; set; }
        double Estimate { get; set; }
        double Actual { get; set; }
        void Update(IProject project);
    }

    public class Project : IProject
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public double Estimate { get; set; }
        public double Actual { get; set; }

        public void Update(IProject project)
        {
            Name = project.Name;
            Estimate = project.Estimate;
            Actual = project.Actual;
        }
    }
}
```



There are certainly better options than using an interface with an update method to allow for updating data objects but this approach will allow us to keep the code in this chapter and the next concise and allow keep our focus on the topic at hand.

`Project` is a simple **domain object** (or **business object**) that stores the project name, estimated cost, and actual cost. It's implemented off an interface to provide more flexibility and better testability and it provides an update method to make it easy to update an instance's values.

Now we will create the data service stub that will return fake data for our various clients to consume so that we don't have to be concerned with data access patterns and techniques and can instead focus on presentation patterns. Add a class to the project called `DataService` and add the code that follows to `DataService.cs`.

This class exposes one method called `GetProjects()`, which creates three projects and then returns them as a `ICollection<Project>`. We have implemented our data service stub based on an interface to support **dependency injection**.



Dependency injection is a pattern where a dependency is allowed to be specified by an external component instead of being created internally. This pattern will be covered in more detail in *Chapter 6, Northwind – Hierarchical View Model and IoC*.

```
using System.Collections.Generic;
namespace ProjectBilling.DataAccess
{
    public interface IDataService
    {
        IList<Project> GetProjects();
    }
    public class DataServiceStub : IDataService
    {
        public IList<Project> GetProjects()
        {
            List<Project> projects = new List<Project>()
            {
                new Project()
                {
                    ID = 0,
                    Name = "Halloway",
                    Estimate = 500
                },
                new Project()
                {
                    ID = 1,
                    Name = "Jones",
                    Estimate = 1500
                },
                new Project()
                {
                    ID = 2,
                    Name = "Smith",
                    Estimate = 2000
                }
            };
            return projects;
        }
    }
}
```



#### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

This will allow us the flexibility to provide different implementations depending on the context. In a unit test we can provide a testing fake (stub or mock), in blend we can return a stub that returns design-time data and at runtime we can provide a real data service that returns real data. We will look into all of these techniques and also the use of inversion of control frameworks that make this process easier later in this book.

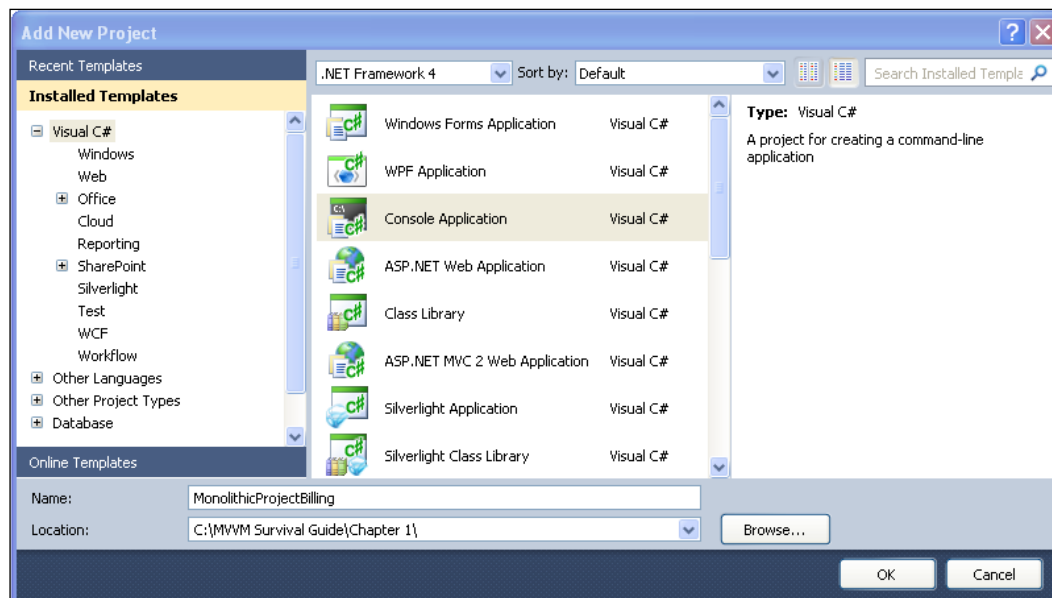
## Monolithic Project Billing sample


Let's go ahead and walk through a simple implementation in WPF of the Project Billing application that was introduced at the beginning of this chapter. We will create the UI using a monolithic style.



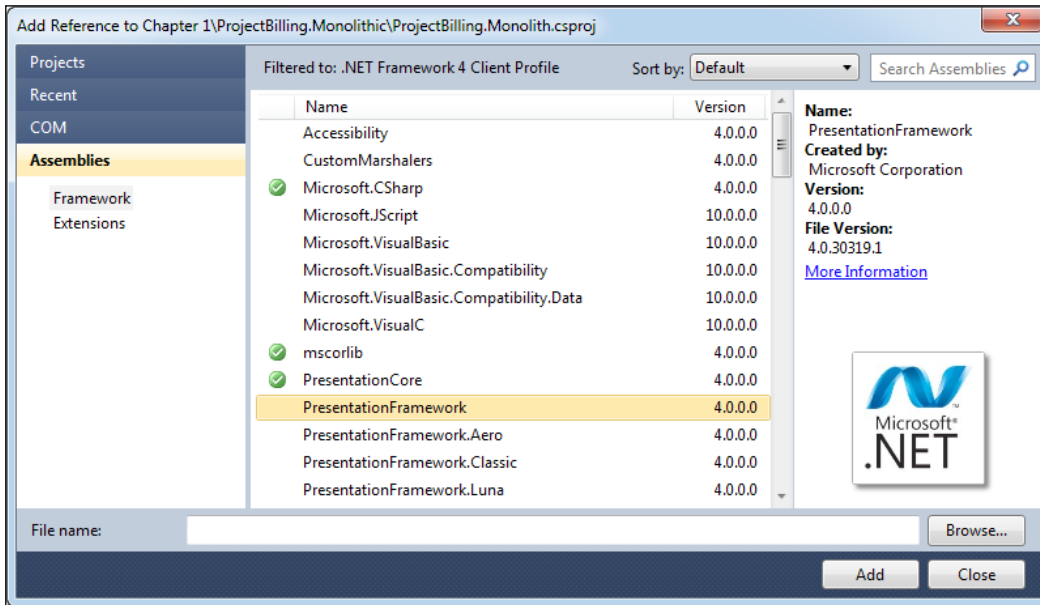
This will be a WPF application but we are not using RAD (Rapid Application Development) support available in Visual Studio, XAML or WPF project templates as it better demonstrates the monolithic style. If you are not familiar with writing code only WPF applications in this style and want to learn more then see *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation*, by Charles Petzold.


Start by creating a solution and then adding a new **Console Application** project named **ProjectBilling.Monolithic** to your solution, as shown in the following screenshot:



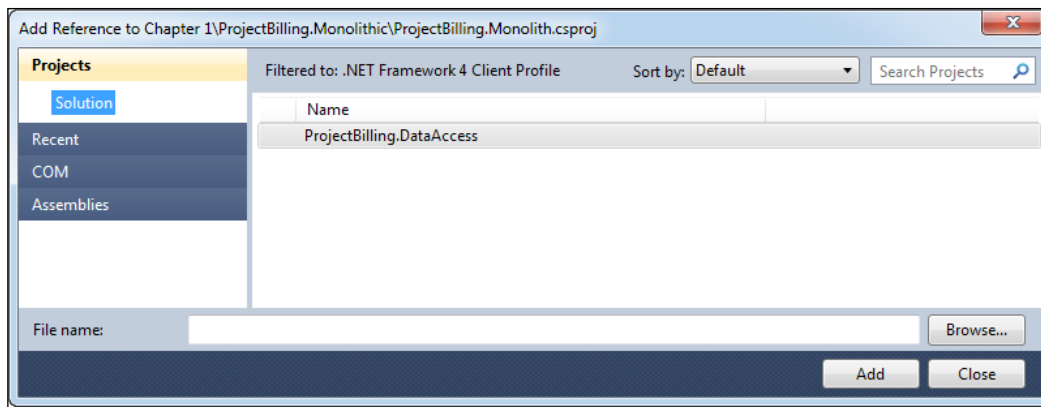
 We will convert this console application to a Windows application later in this section but it's not necessary to do so as you can run a WPF application from a console application. Full details are coming later in this section.

Now add a reference to the **PresentationFramework**, **PresentationCore**, **System.Xaml**, and **WindowsBase** assemblies, as shown in the following screenshot:



 The previous screenshot only shows adding a reference to **PresentationFramework**. Repeat this process for **PresentationCore**, **System.Xaml**, and **WindowsBase** as well.

Now add a project reference to **ProjectBilling.DataAccess**, as shown in the following screenshot:



Next, delete `Program.cs` and add a new class named `ProjectsView` and add the following code to that file.



Using data service means that technically we are not implementing a monolith as we are introducing a data access layer. This is done to keep the code as short as possible. Keep in mind that a purely monolithic application would not have a separate data access layer. The variation of monolithic design that we are implementing here is commonly referred to as **autonomous view**.

## ProjectsView

The heart of this application is the `ProjectsView` class. Let's start by making this class a window and bringing in the namespaces we need.

```
using System;
using System.Windows;
using System.Windows.Controls;
using ProjectBilling.DataAccess;
using System.Windows.Media;

namespace ProjectBilling.UI.Monolithic
{
    sealed class ProjectsView : Window
    {
    }
}
```



This class now derives from `System.Windows.Window`, which is what allows it to be displayed as a WPF application. Add a main function to `ProjectsView` as follows:

```
[STAThread]
static void Main(string[] args)
{
    ProjectsView mainWindow
        = new ProjectsView();
    new Application().Run(mainWindow);
}
```

The main function is given the `STAThread` attribute – which makes it run in a single threaded apartment – which is a requirement of WPF and for interoperability with COM (Component Object Model). The main function simply creates a `ProjectsView` and then passes it to `System.Windows.Application.Run()`, which initializes WPF, starts a message loop, and then displays `ProjectsView` as the application's main window.


## Initialization

Most of the work of the application will be done by the `ProjectsView` constructor and field initializers. Add the following fields to the class:

```
private static readonly Thickness _margin
    = new Thickness(5);
private readonly ComboBox _projectsComboBox
    = new ComboBox() { Margin = _margin };
private readonly TextBox _estimateTextBox
    = new TextBox()
        { IsEnabled = false, Margin = _margin };
private readonly TextBox _actualTextBox
    = new TextBox()
        { IsEnabled = false, Margin = _margin };
private readonly Button _updateButton = new Button()
{
    IsEnabled = false,
    Content = "Update",
    Margin = _margin
};
```

Here we've created the **Project** combobox, **Estimated Cost** and **Actual Cost** textboxes in addition to the **Update** button.

Next let's add a constructor with the following code. We'll start by setting the `Title` and size of the `MonolithicProjectBillingWindow` instance. We will then call two helper methods that will be covered shortly and also add an event handler for the `updateButton.Click` event.


 This event handler will allow the code to be notified of user input via .NET's built-in support for the **Observer** pattern that is implemented by .NET events.

```
public ProjectsView()
{
    Title = "Project";
    Width = 250;
    MinWidth = 250;
    Height = 180;
    MinHeight = 180;

    LoadProjects();

    AddControlsToWindow();

    _updateButton.Click += updateButton_Click;
}
```

 See the *Helpers* section for methods that are called but not yet defined such as `LoadProjects()` and `AddControlsToWindow()`.

## Event handlers

Most of the rest of the functionality of the application is contained within the event handlers:

- The following code will create `projectsComboBox_SelectionChanged()`, which is an event handler for the `projectsComboBox.SelectionChanged` event that we will wire up in the `LoadProjects()` method that was called from the constructor. This code first determines if an item is selected by casting the sender to a `comboBox`, making sure it isn't null and also that an item is selected.

```
private void projectsListBox_SelectionChanged(
    object sender, SelectionChangedEventArgs e)
{
    ComboBox comboBox = sender as ComboBox;
```

```
// If there is a selected item
if (comboBox != null && comboBox.SelectedIndex > -1)
{
    UpdateDetails();
}
else
{
    DisableDetails();
}
}
```

- If there is an item selected in `projectsComboBox` then the `UpdateDetails()` helper method is called; if no item is selected then the `DisableDetails()` helper method is called.
- `updateButton.Click()` is shown in the following code:

```
private void updateButton_Click(object sender,
    RoutedEventArgs e)
{
    Project selectedProject
        = _projectsComboBox.SelectedItem
        as Project;
    if (selectedProject != null)
    {
        selectedProject.Estimate =
            double.Parse(_estimateTextBox.Text);
        if (!string.IsNullOrEmpty(
            _actualTextBox.Text))
        {
            selectedProject.Actual
                = double.Parse(
                    _actualTextBox.Text);
        }
        SetEstimateColor(selectedProject);
    }
}
```

`updateButton.Click()` will fire when the user clicks on the **Update** button and determine if an item is selected. If an item is selected, it will update the details controls with the details of the selected item. The values to populate the details controls will be fetched from the properties of the details controls which we are currently using for view state. Next `updateButton.Click()` will call the `SetEstimateColor()` helper function to update the color of the `estimateTextBox` (*view state*) based on whether the estimated cost is higher or lower than the actual cost (*view logic*).



`_actualTextBox` is checked for null or empty as it starts out in an empty state and could be empty that state if the user updates only the **Estimated Cost** but not actual. This validation was provided to keep the application running down the happy path while all other validation have been left out to keep the code short.

## Helpers

These private helper methods will add the remaining functionality:

- Add the `LoadProjects()` method, as shown in the following code:

```
private void LoadProjects()
{
    foreach (Project project
        in new DataServiceStub().GetProjects())
    {
        _projectsComboBox.Items.Add(project);
    }
    _projectsComboBox.DisplayMemberPath = "Name";
    _projectsComboBox.SelectionChanged
        += new SelectionChangedEventHandler(
            projectsListBox_SelectionChanged);
}
```

- The `LoadProjects()` method will do the following:
  - Fetch the projects to populate the `projectsComboBox` with data retrieved from persisted state by instantiating a new `DataService` and then calling `GetProjects()`
  - The results of `GetProjects()` are iterated over and added to `_projectsComboBox` for display
  - Set the `DisplayMemberPath` to "Name" to use the `Project.Name` property for the displayed text for each project in the `_projectsComboBox.Items` collection
  - Wire up an event handler for the `projectsComboBox.SelectionChanged` event allowing us to update the details view when the user changes the selected project

- Add the `AddControlsToWindow()` method with the following code:

```
private void AddControlsToWindow()
{
    UniformGrid grid = new UniformGrid()
        { Columns = 2 };
}
```

```
grid.Children.Add(new Label()
    { Content = "Project:" });
grid.Children.Add(_projectsComboBox);
Label label = new Label()
    { Content = "Estimated Cost:" };
grid.Children.Add(label);
grid.Children.Add(_estimateTextBox);
label = new Label()
    { Content = "Actual Cost:" };
grid.Children.Add(label);
grid.Children.Add(_actualTextBox);
grid.Children.Add(_updateButton);
Content = grid;
}
```

- The previous code will do the following:
  - Create a new UniformGrid
  - Configure the controls we will be using and then add the controls to the grid
  - Set the grid as the content of the window for display
- Add the `GetGrid()` method to `ProjectsView` as follows:

```
private Grid GetGrid()
{
    Grid grid = new Grid();
    grid.ColumnDefinitions
        .Add(new ColumnDefinition());
    grid.ColumnDefinitions
        .Add(new ColumnDefinition());
    grid.RowDefinitions
        .Add(new RowDefinition());
    grid.RowDefinitions
        .Add(new RowDefinition());
    grid.RowDefinitions
        .Add(new RowDefinition());
    grid.RowDefinitions
        .Add(new RowDefinition());
    return grid;
}
```

- This code creates a 2x3 Grid that is used to create a basic form layout.



We are not trying to make this form pretty but are instead trying to focus on the presentation patterns. One of the big benefits of MVVM is that it will allow us to give our view XAML to a designer and have them make it look nice without having the need to involve the developer. We will look at this approach in detail later in this book in *Chapter 7, Dialogs and MVVM*.

- Add the `UpdateDetails()` method as follows:

```
private void UpdateDetails()
{
    Project selectedProject
        = _projectsComboBox.SelectedItem
        as Project;

    _estimateTextBox.IsEnabled = true;
    _estimateTextBox.Text
        = selectedProject.Estimate.ToString();
    _actualTextBox.IsEnabled = true;
    _actualTextBox.Text
        = (selectedProject.Actual == 0)
        ? ""
        : selectedProject.Actual.ToString();
    SetEstimateColor(selectedProject);
    _updateButton.IsEnabled = true;
}
```

- The `UpdateDetails()` method simply transfers data from the `projectsComboBox.SelectedItem` (or master) to the details controls and then updates the `estimateTextBox` by calling `SetEstimateColor()`.
- Add a `DisableDetails()` method as follows:

```
private void DisableDetails()
{
    _estimateTextBox.IsEnabled = false;
    _actualTextBox.IsEnabled = false;
    _updateButton.IsEnabled = false;
}
```

- The `DisableDetails()` method sets the details controls `IsEnabled` to `false` along with the update button.

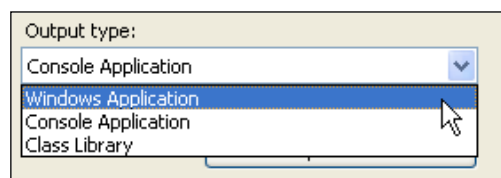
- Add `SetEstimateColor()` as follows:

```
private void SetEstimateColor(Project selectedProject)
{
    if (selectedProject.Actual == 0)
    {
        this.estimateTextBox.Foreground
            = _actualTextBox.Foreground;
    }
    else if (selectedProject.Actual
        <= selectedProject.Estimate)
    {
        this.estimateTextBox.Foreground
            = Brushes.Green;
    }
    else
    {
        this.estimateTextBox.Foreground
            = Brushes.Red;
    }
}
```

- The `SetEstimateColor()` method will be called by both event handlers to update the color of **Estimated Cost** (view state) by examining the Actual Cost and Estimated Cost.

## Running the sample

Right-click on the **ProjectBilling.Monolithic** project and select **Properties**. Next, set the **Output type** to **Windows Application** as shown in the following screenshot:





If you leave the **Project** type as **Console Application** then a **Console Window** will be displayed while your WPF application runs. This can be useful for debugging as you can write debug messages to the console and easily kill the application using *Ctrl + C* when debugging.

Finally set **ProjectBilling.Monolithic** as the startup project by right-clicking on it and selecting **Set as StartUp project**. Now run the application by hitting *F5*.

You should now see an application as shown in *The Project Billing sample application* section at the beginning of this chapter.

## Takeaways

This code gets the job done, so what's the problem and why is there the need to restructure it?

### Poor testability

This code has poor testability as the entire code is tightly coupled to the view and requires the view to fire the events that drive the logic of application. You could change the access modifiers of the methods of `ProjectsView` to public the help alleviate the situation but then you weaken the design from the **encapsulation** and **design by contract** perspectives.



Encapsulation and design by contract are basic principles of Object-oriented design that are covered extensively on the Web. Please look up for them if you are already not familiar with them.

### Poor extensibility and code reuse

If the users wanted a command line or web-interface, all of the code would need to be rewritten. Also, supporting multiple synchronized **ProjectView** is not possible under this design and would require at a minimum refactoring out a model.



We will demonstrate how adding SoC allows for creating multiple synchronized views of the model when we get to the *MVC* section.



## Rapid application development

Microsoft puts a lot of development effort into creating **Rapid Application Development** (or **RAD**) tools that allow developers to simply drag-and-drop controls onto the IDE's design surface and then allow for configuring the controls' data needs mostly through the IDE's designer. The designer then creates monolithic code to get the job done. These tools make the problems of monolithic design worse by encouraging that style of design and by making it easier to do.

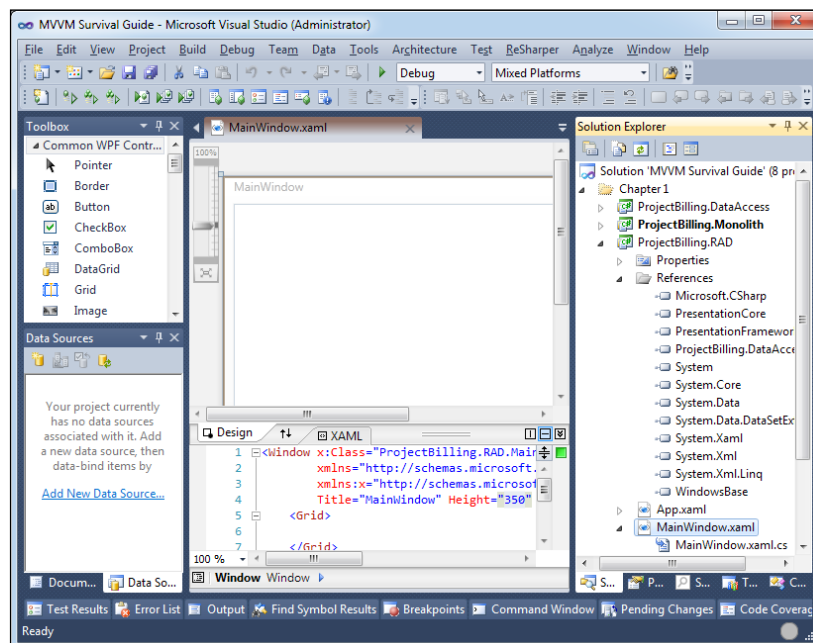
## RAD Project Billing sample

This section will walk through rewriting the Project Billing application using RAD tools in Visual Studio.

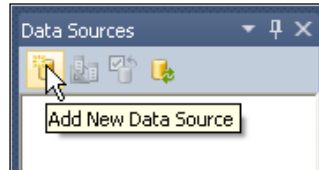
Start by adding a new **WPF Application** project to your solution called **ProjectBilling.RAD**. This project template creates two files for you, `App.xaml` and `MainWindow.xaml`.

Next add a project reference to **ProjectBilling.DataAccess**.

Open `MainWindow.xaml` in **Cider** (the WPF designer) by double-clicking on `MainWindow.xaml` in the **Solution Explorer**. If they're not already expanded, expand the **Toolbox** window and the **Data Sources** window. You should have Visual Studio set up as shown in the following screenshot:



The first step is to add an **Object Data Source** to connect to `DataService.GetProjects()`. To do this start by clicking on **Add New Data Source** in the **Data Sources** window, as shown in the following screenshot:



You will now be presented with a dialog that will allow you to specify an **Object Data Source**, as shown in the following screenshot:

