

Prism 4

Succinctly

by Eric Stitt

Prism 4 Succinctly

By

Eric Stitt

Foreword by Daniel Jebaraj



Copyright © 2014 by Syncfusion Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Zoran Maksimovic, [@zoranmax](https://twitter.com/zoranmax), www.agile-code.com

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	5
About the Author.....	7
Introduction	8
Chapter 1 What is Microsoft Prism 4?	9
Chapter 2 Getting Started.....	11
Chapter 3 The Prism 4 Startup Process	18
Chapter 4 The Virtual Calculator	25
Chapter 5 Dependency Injection and the Prism 4 Bootstrapper.....	28
Chapter 6 The Prism 4 Shell Form	41
Chapter 7 Prism 4 and MVVM	48
Chapter 8 Prism 4 Regions	61
Chapter 9 Prism 4 Modules	67
Chapter 10 Prism 4 Commands	74
Chapter 11 Prism 4 Event Aggregation	81
Chapter 12 Prism 4 Navigation	90
Chapter 13 The Virtual Calculator Solution	95
Chapter 14 Conclusion	110
References	111

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Eric Stitt is a developer who has worked with Microsoft .NET technologies for more than 20 years.

He has worked as a .NET developer in Government (Law Enforcement), Finance, Medical, and the Insurance industries.

His current focus is WPF, Microsoft Silverlight, Windows Phone, Microsoft Prism, and other associated technologies. He Works with both Visual Basic .NET and C#.

Currently he is CEO of Practical Technologies, LLC in Phoenix, Arizona.

Introduction

My first experience with Microsoft Prism 4 was when I started working with WPF solutions about three years ago. At first glance, I found Prism 4 to be quite intimidating. Looking at Prism 4 from a high level, all that I could see were complexities. It seemed that I had more questions than answers!

What is the best method for starting a Prism 4 solution? What is Dependency Injection (DI)? Why are modules, regions, bootstrappers and navigation necessary? How do I format folder structures in Prism 4 projects? Why use different module projects in the first place?

Before I could successfully build Prism 4 WPF solutions, I had to answer all of these questions and more. I addressed these issues in the same manner that I would solve electronics issues when building a circuit: break down the full set of issues into discrete parts and solve them one at a time. When all of the parts are solved, there are no more issues!

That's the objective of this book. We'll look at the parts of Prism 4 and in a succinct manner, show how some of the more important Prism 4 parts work and integrate. When we finish, we'll have a set of working parts that can be used to build Microsoft Prism 4 solutions.

This book includes three demonstration projects: *Hello World*, *Virtual Calculator* and the *Event Aggregation – View-Based Navigation* solution.

Chapter 1 What is Microsoft Prism 4?

Application Architecture

When most developers see the words, "Application Architecture," they either cringe or their eyes light up with anticipation. This love/hate relationship is due to the complexities involved when developing well-designed solutions.

As developers we tend to see a broad range of applications ranging from well-designed architectures to solutions that seem to be patched together with no regard to good programming practices whatsoever. In a lot of cases we're stuck with the current architecture because that's what was designed before we even came into the picture. The arguments for refactoring these projects are strong, but not always practical.

We are going to talk about cases where we can refactor or build solutions from scratch; that's where application architecture comes into play. Most developers find the time to read at least one book on this subject and if you are interested in Prism 4, my assumption is that you are aware of the principles and design patterns that are used to define a well-designed solution. So, where does Prism 4 fit into this equation?

Microsoft Prism 4

Microsoft Prism 4 is a framework designed by the Patterns and Practices team at Microsoft. It is designed to serve as a template to help developers build solutions. It's designed to be used with Extensible Application Markup Language (XAML) solutions that depend on the .NET framework. These solution types include Microsoft WPF, Silverlight, and Windows Phone 7 or 8 at this time. The Prism 4 framework has a library for each of these three solution types.



Note: In this book we are going to use WPF. WPF is the most feature-rich of the three libraries with Silverlight and Phone 7 or 8, exposing less features respectively.

Why Use Prism 4?

Microsoft designed Prism 4 to be used with enterprise solutions that use WPF, Silverlight, or Windows Phone.

After using Prism 4 in numerous solutions, I tend to take a broader view. Why use Prism 4 in just enterprise solutions? If the goal is to promote good programming practices, why relegate it to only large, enterprise-wide solutions? In your experience, how often has a seemingly simple project grown into a massive project? How often have you had to deal with shortcomings in these projects when it was necessary to modify or add features?

If Prism 4 promotes good programming practices with enterprise solutions, why should we relegate smaller projects to anything but the same good programming practices?

I wish that I could tell you that Prism 4 is the cure-all for developing well architected XAML-based solutions, but alas, that's not the case. Prism 4 will not automatically enforce good programming practices; it's necessary for us as developers to understand DI, OOP, SOLID, DRY, and other good practices to effectively use Prism 4. What you will find is that as you use Prism 4, you'll start to see how it can be used to reduce the amount of work needed to produce well-designed solutions.

In this book I'll explain some of the best practices for creating well architected solutions and we'll then correlate how Prism 4 allows us to implement each practice.

Microsoft Prism 4 Features

Prism 4 is actually composed of a number of features that in most cases can be used independently or with other features of the framework. You can, for instance, use Dependency Injection (DI), and exclude all of the other Prism 4 features. But Prism 4 is most effective when all of its features are used in conjunction with one another.

Here is a list of Prism 4 features:

- Bootstrapper class
- Dependency Injection Container (Unity, Managed Extensibility Framework (MEF), etc.).
- Shell form
- Regions
- Modules
- Navigation
- Commanding
- Event aggregation

Other patterns and applications that will be used:

- The Model–View–ViewModel (MVVM) design pattern.
- The NUnit (Unit Test Application).

The companion demo solutions for this book will use most of the features in this list to create three simple Prism 4 solutions. The *Hello World* solution will walk you through building a barebones WPF/Prism 4 solution. The *Virtual Calculator* will work with a more complex solution that is used to add and subtract two numbers. The *Event Aggregation - View Based Navigation* solution shows an example of using Prism 4 to communicate between modules and to show views in a loosely coupled manner. We'll use these three solutions to show Prism 4 integration with WPF solutions.

Any one of Prism 4's features is a strong motivation to use the framework. As we progress in this book, I show you why and how to combine these features to create reliable XAML-based solutions.

Chapter 2 Getting Started

What is Needed to use the Prism 4 Framework?

The Prism 4 framework can be used to with any version of Microsoft Visual Studio 2008 and above. As you already know, it can be used with WPF, Silverlight, and Phone 7 or 8 solutions. Here is a list of applications that were used to build the three solutions that accompany this book. If you decide to build these solutions, you will need to download and install the following applications on your computer.

Applications that were used with the Microsoft Prism 4 demo solutions:

- Microsoft Visual Studio 2012 Express or higher
- Syncfusion Essential Studio Enterprise Edition
- Microsoft Expression Studio 4 Ultimate
 - Microsoft Expression Blend 4
 - Microsoft Expression Design 4
- NUnit (Unit Test Application).

Syncfusion Versions

The version of Essential Studio Enterprise Edition that is used with the demo solutions is version 11.2045.0.25; this is the version that is needed for the demo solutions. If you are going to build the solutions from scratch, just download the latest version of the Syncfusion suite and use that version.

What is the Cost of the Prism 4 Framework?

First things first. What is the cost of the Prism 4 framework? One of the things that you'll like most about Prism 4 is the cost—it's free. All you have to do is go the website and download the framework. Let's do that now.

Downloading Microsoft Prism 4.1


You can download Prism 4.1 from the Microsoft Download Center at:
<http://www.microsoft.com/en-us/download/details.aspx?id=28950>.



Prism 4.1 - February 2012

Share  Language: **English****Download**

Prism provides guidance designed to help you more easily design and build rich, flexible, and easy to maintain Windows Presentation Foundation (WPF) desktop applications and Silverlight Rich Internet Applications (RIAs) and Windows Phone 7.1 ("Mango") applications.

 [Details](#) [System Requirements](#) [Install Instructions](#) [Additional Information](#) [Related Resources](#)

Free PC updates


- Security patches
- Software updates
- Service packs
- Hardware drivers

 [Run Microsoft Update](#)

Microsoft suggests

**Visual Studio 2013 Release Candidate**

Get the tools you need to build great software.

 [Download now](#)

Popular downloads

Figure 1: Microsoft Prism 4 Download Site

The download includes:

- Prism 4.1 DLLs
- Numerous quick-start solutions for WPF and Silverlight
- The Prism 4 reference manual for C# or Visual Basic .NET
- Two reference implementations (Stock Trader and MVVM)
- Other resources

We'll be working with Prism version 4.1. To start the download, click the **Download** button. After the download has finished, run the **Prism4.1_Source** exe file in the **Downloads** folder of your computer. You will be asked to accept the license agreement and to determine where the files are to be loaded. Click **OK** to extract the files to the selected location.

After the files are extracted, you will be able to either automatically associate Prism 4 DLL files with Visual Studio projects or do so manually.



Note: I move the Prism 4 DLL files to a centralized folder that also contains other DLLs that are used on a regular basis in my Prism 4 solutions.

Getting the Other Applications

Before we start the project, there are a few things that we should address. First, I used Visual Studio Professional or higher for the demo solutions. If you don't have Visual Studio 2012 Professional or higher, Visual Studio Express 2013 for Windows Desktop is a newer and free version of the application that can be used.

If you don't have a copy of Visual Studio 2013, you can get the express version at <http://www.microsoft.com/visualstudio/eng/downloads#d-express-windows-desktop/>.

Visual Studio

Products ALM Download Buy

Visual Studio Express 2013 for Windows Desktop

Microsoft Visual Studio Express 2013 for Windows Desktop

Visual Studio Express 2013 for Windows Desktop enables the creation of desktop apps in C#, Visual Basic, and C++, and supports Windows Presentation Foundation (WPF), Windows Forms, and Win32.

Sign in to Visual Studio within 30 days with your Microsoft account to synchronize your settings across multiple machines and register your product.

System requirements

Download language

English

Installation options

Microsoft Visual Studio Express 2013 for Windows Desktop - English
Install now

Microsoft Visual Studio Express 2013 for Windows Desktop - English
Download now

Microsoft Visual Studio Express 2013 for Windows Desktop Language Pack

The Visual Studio Express 2013 for Windows Desktop Language Pack is a free add-on that you can use to switch the language that's displayed in the Visual Studio user interface.

Download language

Český

Installation options

Microsoft Visual Studio Express 2013 for Windows Desktop Language Pack - Český
Download now

Visual Studio Team Foundation Server Express 2013

Visual Studio Express 2012 for Windows Phone

No cost to students: Visual Studio Professional 2013!
Download it today at DreamSpark.com

Microsoft DreamSpark

Figure 2: Visual Studio Express 2013 for Windows Desktop Download Site

Microsoft Expression Blend 4 and Microsoft Expression Design 4 were used to create the animations and graphics for the *Virtual Calculator*. If you don't have a copy, you can download a free copy of these two applications at <http://www.microsoft.com/expression/eng>.

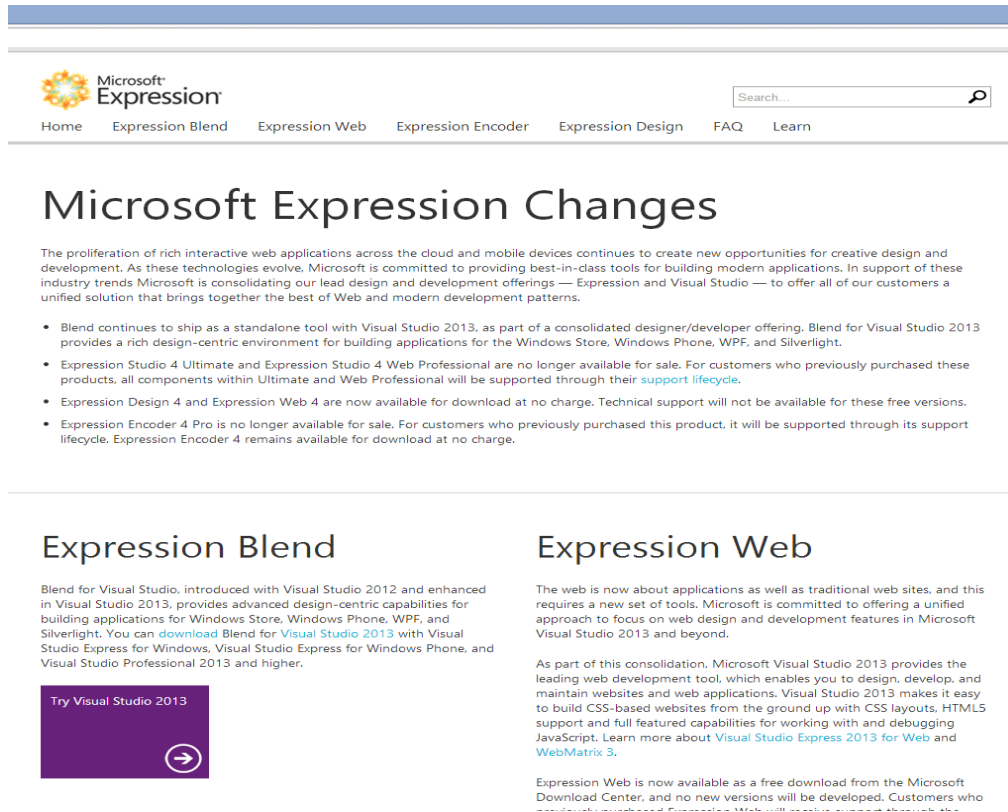


Figure 3: Microsoft Expression Web Site

We'll also use Syncfusion's Essential Studio Enterprise Edition control suite in this book. A 30-day evaluation of the suite can be downloaded at: <http://www.syncfusion.com/downloads/evaluation>.

[Products](#)
[Solutions](#)
[Support](#)
[Purchase](#)
[Downloads](#)
[Company](#)

[Log In](#)
[Download](#)
[Live Demo](#)

[Home](#)
[Downloads](#)
[Evaluation](#)

You need to register for a free account to download Enterprise Edition

Signing up is easy and takes less than 30 seconds. Your data is safe with us. [Privacy Policy](#)

Free registration benefits

10+
E-books
on the latest technologies

2,500+
Metro-style icons
with a developer-friendly editor

- Fully functional evaluation.
- Access to award-winning support during evaluation.
- Access to forums.
- White papers on the latest technologies.

Name (required)

Company (required)

Email (required)

Phone (required)

Download Now

Existing Users — [Log in](#)

Call Toll Free 1-888-9DOTNET

Products	Solutions	Sales	Downloads	Company	Call us
Essential Studio	Visualization	Order Online	Evaluation	About Us	Toll Free: 1-888-9DOTNET
Orubase Studio	Business Intelligence	Price List	Free E-Books	News & Events	Fax: +1 919.573.0306
Metro Studio	Enterprise Reporting	Studio Comparisons	White Papers	Media Kit	US: +1 919.481.1974
What's New	Enterprise Mobility	Resellers	Brochures & Datasheets	Blog	UK: +44 20 7084 6215
Road Map	Support	Contact Sales	Case Studies	Newsletter	Sales

Figure 4: The Syncfusion Download Site

If you want to run the unit tests in the solution, I used NUnit as the unit test application. You can download a free copy of NUnit at: <http://www.nunit.org/index.php?p=download>.

[HOME](#)[DOWNLOAD](#)[DOCUMENTATION](#)[WIKI](#)[BLOG](#)[CONTACT US](#)

Downloads

Current Stable Release: NUnit 2.6.3

NUnit 2.6.3 - October 10, 2013	
<i>win</i>	NUnit-2.6.3.msi
<i>bin</i>	NUnit-2.6.3.zip
<i>win .net 1.1</i>	NUnit-2.6.3-net-1.1.msi
<i>bin .net 1.1</i>	NUnit-2.6.3-net-1.1.zip
<i>src</i>	NUnit-2.6.3-src.zip
<i>doc</i>	NUnit-2.6.3-docs.zip

Under Development: NUnit 2.9.6

NUnit 2.9.6 - October 4, 2013	
<i>bin</i>	NUnit-2.9.6.zip
<i>src</i>	NUnit-2.9.6-src.zip

Download Types

The following types of downloads are provided:

- Windows installer packages (marked *win*) are for use on Windows under Microsoft .NET or Mono.
- Zipped source packages (*src*) are provided for those who plan to contribute to the NUnit project or to modify NUnit for their own use.
- Zipped binary releases (*bin*) are suitable for use on either Windows or Linux and are also useful for teams that prefer to place NUnit into their source code management system.
- Separate documentation packages (*doc*) are provided starting with NUnit 2.4.2. Since all the other packages include documentation, this is only useful for those wanting to review the docs first.
- Starting with NUnit 2.6.2, samples are packaged separately (*samples*).

Instructions for installing each of these are found on the Installation page linked to from the [Documentation](#) for the particular release. The current version of NUnit can also be found on [Launchpad](#).

Figure 5: NUnit Download Site

What's Next?

Now that we have Prism 4 and the other applications that are needed, we'll talk about how to build a *Hello World* solution with WPF and Prism 4.

Chapter 3 The Prism 4 Startup Process

Building a Hello World Solution

We're going to start our exploration of Microsoft WPF and Prism 4 by building the simplest solution possible. Our objective will be to create a solution that shows the following shell form.



Figure 6: The Hello World Shell Form

Create the Hello World Solution

The first step is to create a solution that will contain our projects. Run Visual Studio and select **FILE - New - Project** from the Visual Studio main menu and create a Blank Solution. The Blank Solution is located in the **Other Project Types** folder of the New project dialog under Visual Studio Solutions. Name the Solution **HELLO_WORLD**. Place it in the folder of your choice.

Create the Main Project

Next we'll create a WPF project for the solution. Select **FILE - Add - New Project** from the Visual Studio main menu and create a C# WPF Application. Name it **HELLO_WORLD.MAIN**.

We now have a WPF solution that is ready to run. If you would like, click **Start** in Visual Studio and you will see a blank form displayed.

If we take a look at the files that were generated by Visual Studio, we'll see that there is a `MainWindow.xaml` file and an `App.xaml` file. Expand the `App.xaml` file and you'll see an `App.xaml.cs` file. There is also a references folder. These are the files and folders that we will be concerned with in this solution.

Referencing the Prism 4 Library

It's necessary to reference the Microsoft Prism 4 library before we can take advantage of the framework. Right-click on the **References** folder in the *Main* project and select **Add Reference**. Browse to the folder where the Prism 4 DLLs are located on your computer. Add the following highlighted DLLs:

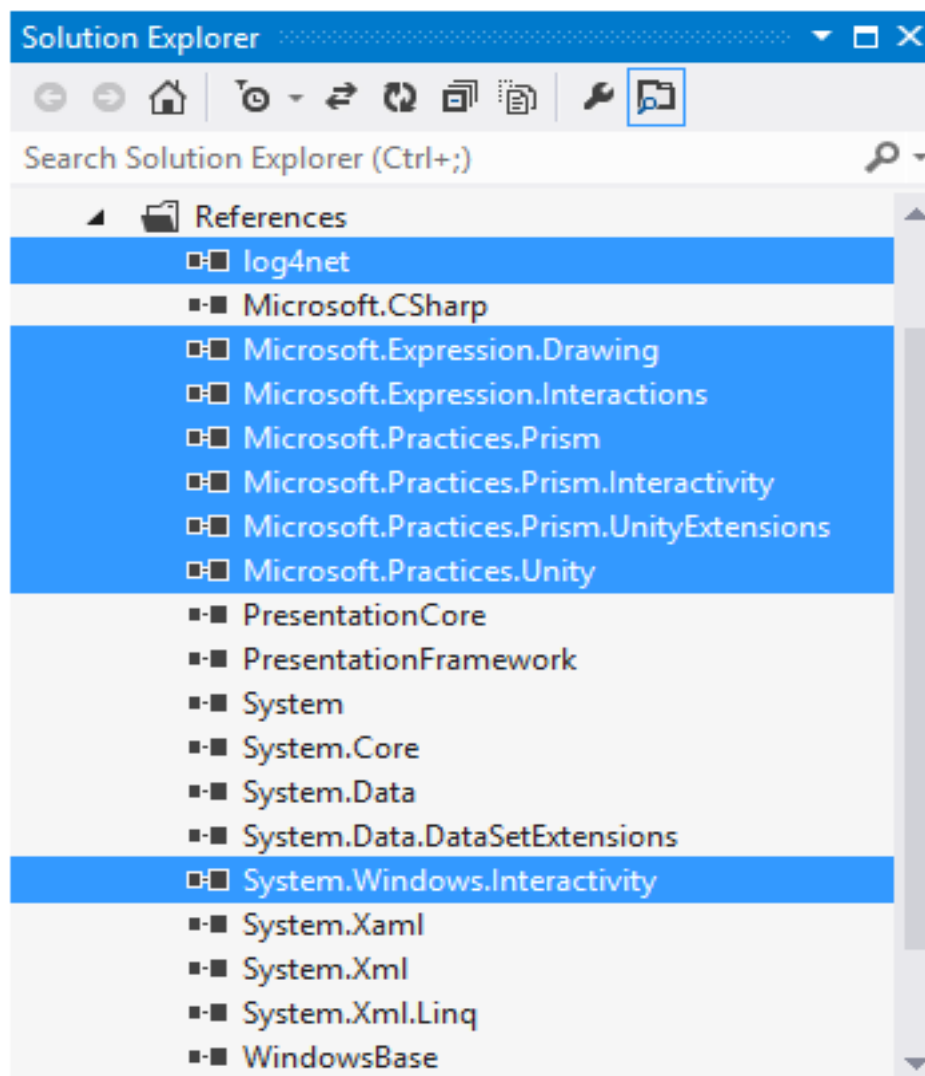


Figure 7: The *HELLO_WORLD.MAIN* project references



Tip: You can find the DLL files for Microsoft Prism 4 in the `Bin\Desktop` folder of the Prism 4 framework library that you downloaded. The `log4net` DLL can be downloaded via NuGet.

We now have the necessary libraries to work with Prism 4.

The Microsoft Prism 4 Startup Process

As we saw earlier, starting a WPF solution is easy. Simply add the project and run it. With Prism 4, the startup process is different. Let's first take a look at the markup that is used to start the WPF solution.

Listing 1: The App.xaml File Markup

```
<Application
  x:Class="HELLO_WORLD.MAIN.App"

  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

  StartupUri="MainWindow.xaml">

  <Application.Resources>

  </Application.Resources>

</Application>
```

In Listing 1, the markup uses the **StartupUri** property to set the startup form to the **MainWindow.xaml** file. While this is a fast and easy manner in which to start the solution, it severely limits the amount of control that we have over the startup process. To address this issue, we'll first remove the **StartupUri** property from the **App.xaml** file. This is the markup that should be removed: **StartupUri="MainWindow.xaml"**. Remove the markup now.



Tip: There is a closing bracket (>) at the end of the **StartupUri** property. Make sure to only remove the property and not the closing bracket. Failure to do so will result in an exception.

The Prism 4 Bootstrapper Class

Our next step in creating our Hello World project is to create a **Bootstrapper** class. This class is used to start Prism 4 solutions. As you will see later, this class gives us almost unlimited control over the startup process. We'll talk in detail about the **Bootstrapper** class in Chapter 5; but for now, let's add and create the class for this project.

Add The Prism 4 Bootstrapper Class to the Main Project

Right-click on the *Main* project and select **Add - Class** from the context menu. In the **Add New Item** dialog, enter **Bootstrapper** into the **Name** textbox. Click **Add** to add the class.

Add The Prism 4 Bootstrapper Code

Add the following code to the **Bootstrapper** class:

Listing 2: The Bootstrapper Class

```
using System;
using System.Windows;

using Microsoft.Practices.Prism;
using Microsoft.Practices.Unity;
using Microsoft.Practices.Prism.UnityExtensions;

namespace HELLO_WORLD.MAIN
{
    public class Bootstrapper : UnityBootstrapper
    {
        protected override DependencyObject CreateShell()
        {
            //Create and show the Prism 4 shell form:
            MainWindow Shell = Container.Resolve<MainWindow>();

            Shell.Show();
            return Shell;
        }
    }
}
```

Let's go over the code in Listing 2. First notice that the class inherits the **UnityBootstrapper** class. This is our Dependency Injection Container (DIC). If you're not familiar with DICs, don't worry—I'll go over what they are and how they work in Chapter 5.

Next is the **CreateShell** method. This method is used to replace the **StartupUri** markup that we removed in the App.xaml file earlier. This method returns a **DependencyObject**; returning this type of object allows us to return any object that derives from the **DependencyObject** type. In this case, we return the **MainWindow** as our shell form. Notice that we use the DIC to create our shell form instead of using the **new** keyword to instantiate the object. We'll go into why we do things this way in later chapters. But for now, just remember that with Unity, resolving entities is equivalent to instantiating objects with the **new** keyword.

Use the Prism 4 Bootstrapper Code

You may have noticed that we didn't actually create and use the bootstrapper yet. This is our next step. Now that we have a **Bootstrapper** class, let's instantiate and call the **CreateShell** method.

Listing 3: The App.xaml.cs Code

```
using System;
using System.Windows;

namespace HELLO_WORLD.MAIN
```

```

{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            Bootstrapper Bootstrapper = new Bootstrapper();
            Bootstrapper.Run();
        }
    }
}

```

The code in Listing 3 is used to instantiate the **Bootstrapper** class. Next, we call the **Bootstrapper.Run** method. This method calls the other methods in the class, including the **CreateShell** method, which is used to show the shell form.

Running the solution will display our shell form. The shell form at this time is just a blank form. Let's add a message to the form.

Add a Label to the Shell Form

Now that we have a working solution, let's finish up by adding a message to our shell form. Add the following markup to the shell form:

Listing 4: The Shell Form Markup

```

<Window
    x:Class="HELLO_WORLD.MAIN.MainWindow"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    Title="Hello World"
    Height="350"
    Width="525"
    WindowStartupLocation="CenterScreen">

    <Window.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF2D6FA6" Offset="0"/>
            <GradientStop Color="#FF084D87" Offset="1"/>
            <GradientStop Color="#FFF0F045" Offset="0.549"/>
        </LinearGradientBrush>
    </Window.Background>

    <Grid>

        <Label
            Name="MainLabel"
            Content="HELLO PRISM 4!"
            HorizontalAlignment="Center"

```

```
        VerticalAlignment="Center"
        FontSize="36"
        FontWeight="Bold"
        Foreground="Red">
    </Label>

</Grid>

</Window>
```

The Listing 4 XAML markup is a good example of what is necessary to create a basic layout with WPF. As we progress, you'll find that XAML is one of the most powerful and extensive markup languages that you are likely to encounter. Let's take a closer look at the markup.

The base control is a **System.Windows.Window** control. This is the container control for all of the other controls that will be used in the window.

The **x:Class** tag combines the window's namespace and class name; it describes the associated class for the window control.

The two XML namespace tags associate the window with DLL files that the window will use when working with objects.

The next lines of markup expose properties of the window. For instance, the window title is set to "Hello World" and the **WindowStartupLocation** is set to "CenterScreen".

Next, the window **Background** is set to a **LinearGradientBrush**. These are the three colors that you see on the background of the form.

The **Grid** control is a powerful layout control that can be used to position controls on the shell form and in other container controls.

Contained inside the **Grid** control is a label with its content set to **HELLO PRISM 4**.

Figure 6 shows an example of the finished solution.

One interesting thing to note is that we have only used one Prism 4 construct in this example. The only Prism 4 library that we have used so far is the Unity Dependency Injection Container (DIC) in our **Bootstrapper** class! The Unity DIC is included with Prism 4, but it is really a standalone DIC—it could just as easily be used independently.

Don't despair! In the following chapters we'll integrate Prism 4 into this chapter's basic framework as we look at more detailed Prism 4 solutions.



Note: Another interesting point is that the Unity DIC is also included in the Microsoft Enterprise Library as a one of its application blocks.

Summary

In this chapter we took a look at what it takes to build a Microsoft Prism 4 barebones solution. It takes a bit more work to start a Prism 4 solution, but in the following chapters we'll see how this extra work pays off when building well architected WPF solutions.

Chapter 4 The Virtual Calculator

Now that we have a basic starting point for building Prism 4 solutions, we'll start to progressively look at Prism 4 in greater detail. To do this, we'll use a demonstration solution, The *Virtual Calculator*. Figure 8 shows the solution's logo top shell form.

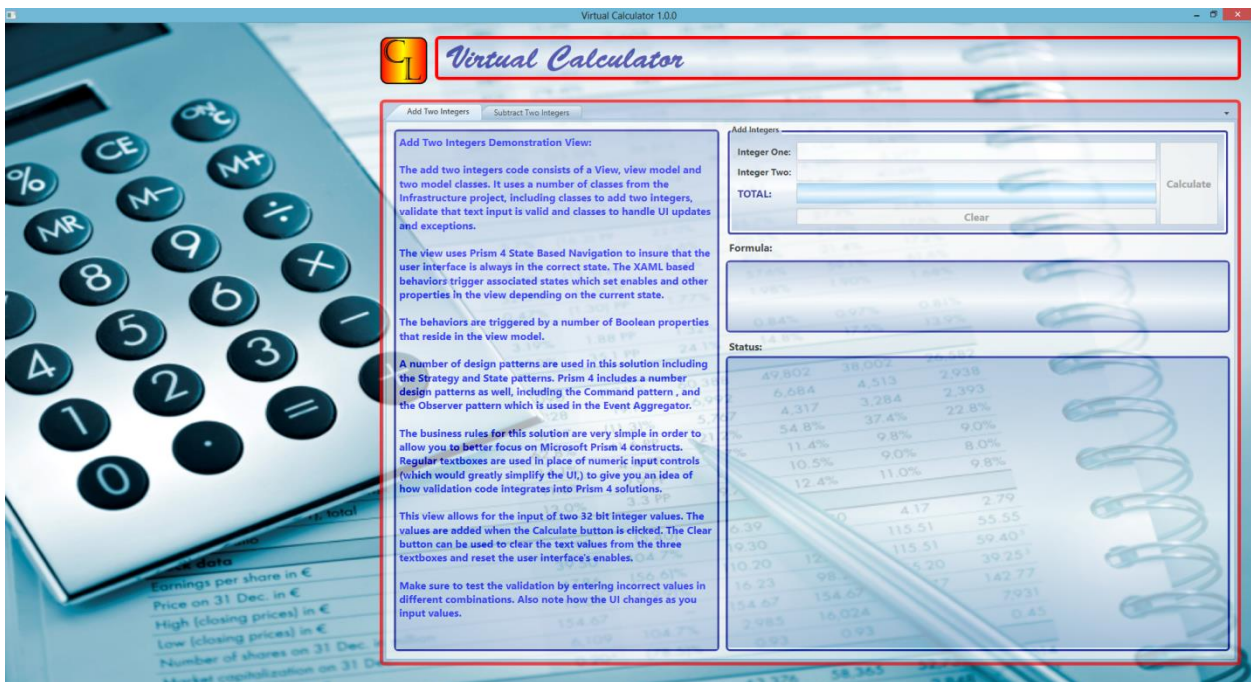


Figure 8: The Virtual Calculator Logo Top Shell Form



Note: In Chapter 13, we'll take a deeper look at the *Virtual Calculator* and how it relates to Prism 4.

Virtual Calculator Specifications

The *Virtual Calculator* is a WPF solution that uses the Microsoft Prism 4 framework. The solution is designed to allow users to either add or subtract two numbers.

I have kept the business rules simple so that we can focus on the WPF and Prism 4 constructs in the solution.

The *Virtual Calculator* solution consists of two shell forms that can be interchanged by setting an integer value (**ShellSelector**) in the **Bootstrapper** class. The difference between the two forms is the logo placement. Figure 9 shows an example of the logo bottom shell form.

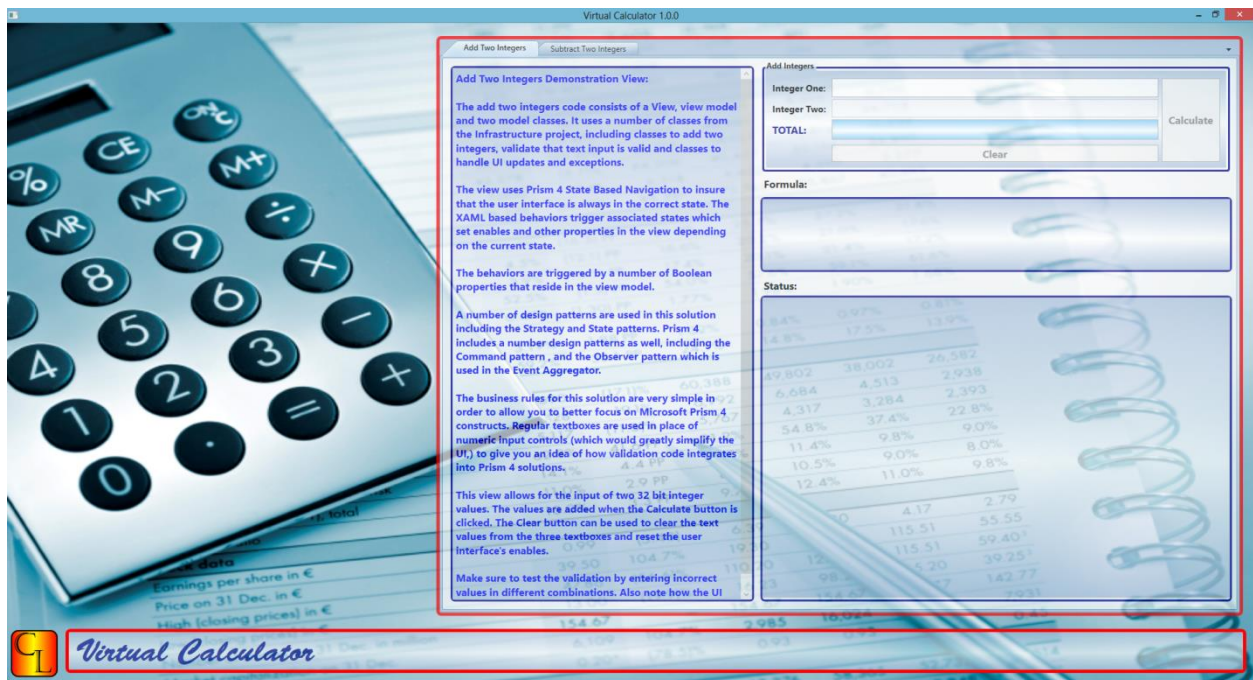


Figure 9: The Virtual Calculator Logo Bottom Shell Form

The solution uses three Microsoft Prism 4 modules:

- The Application Logo Module
- The Company Logo Module
- The Math Module

There are also three other projects in the solution:

- The Main Project
- The Infrastructure Project
- The Unit Test Project

The Application Logo Module

The Application Logo Module is used to show the Virtual Calculator's application logo graphic. It is the graphic with the red border and **Virtual Calculator** text.

The Company Logo Module

The Company Logo Module is used to show the Virtual Calculator's company logo-animated graphic. It is the graphic with the red and yellow background and the **CL** text, which stands for Company Logo in this case.

The Math Module

The Math Module is used to show the Virtual Calculator's two tabs: the Add Two Integers and Subtract Two Integers tabs. This module also enforces the solution's business rules. This includes data input validation, updates to the User Interface (UI), and the code needed to add and subtract.

The Main Project

The Main Project is the startup project for the solution. This is where most of the initialization is carried out, and where the shell forms are located. The bootstrapper and main configuration file also reside in this project.

The Infrastructure Project

The Infrastructure Project is unique in the solution as it is the only project that is referenced in all of the other projects. This is our global project that provides services to each of the other five projects in the solution. This project is used to provide both Business and User Interface services to the Virtual Calculator.

In Prism 4 solutions that use relational or other kinds of persistent data, the infrastructure project can be used to access and modify the data.

It is also where Prism 4 global commands and the event aggregator code reside. These Prism 4 design patterns can be used to communicate between modules in a loosely coupled manner.

We'll go into more detail about all of these projects in Chapter 13.

The Unit Test Project

The Unit Test Project is where Unit and Integration tests reside for all of the other projects in the solution. The NUnit framework was used to create the unit tests for this solution.

I won't be going into detail about unit testing in this book, but some unit test examples are included in the solution. See Chapter 2 if you need to download NUnit.

Summary

These six projects provide the functionality needed to generate the user interfaces and enforce the business rules in the Virtual Calculator solution. The next chapters will explain the code that was used to create this solution and give you some insights into Microsoft WPF-based Prism 4 solutions.

Chapter 5 Dependency Injection and the Prism 4 Bootstrapper

Dependency Injection (DI) tends to be one of the most confusing concepts in programming.

In this chapter, we'll address some common questions and show how DI and DICs integrate into Microsoft Prism 4. We'll talk about some strategies that will help when working with DI in Prism 4, and we'll look at how to use DI with some of the Gang of Four (GoF) design patterns.

Let's take a look at some commonly asked questions about Dependency Injection.

Are Dependency Injection and Inversion of Control (IoC) the same?

No. IoC refers to a broad range of programming styles where a framework controls program flow. DI is included in this definition as an implementation, but not as IoC. So it is more accurate to think of DI as one aspect of IoC.

Is DI simply an Enhanced Factory Design Pattern?

No. See *Is DI a Service Locator?*

Factories simply create single objects; they could be enhanced to do more, but why do so when we have DI? DI is designed to create object graphs. What this means is that when an object is created, any needed dependences (entities that the object needs to do its work) are created without the parent object specifically requesting them. In most cases the dependent entities are injected into the parent entity when it is created.

Is DI a Service Locator?

No. Service locators are general-purpose factories that supply services to a requesting entity. DI, on the other hand, provides a means to transparently ensure that an entity is supplied with its necessary services. The entity should not find it necessary to ask for a collaborating entity—it should simply be supplied.

Are DI and Dependency Injection Containers (DIC) the same?

No. DI is the technique that is needed to create loosely coupled code. This can be done without a DIC. A DIC simply encapsulates the DI functionality into a library such as Unity. While DI can be implemented as needed, a DIC makes things a lot easier.

When should I use DI?

DI should be used when volatile dependencies are introduced to your solutions. Volatile dependencies are entities in your code that are not stable. A good example is when a relational database is used with your solution; you don't really have control over the database that your clients use. In fact, there is nothing stopping clients from starting with one database, Microsoft Access or Oracle for example, and changing to another, say SQL Server. Another scenario is when your clients want to have both relational and cloud-based data repositories that are accessed from the same solution.

In the previous cases, it's important to build loosely coupled entities. It's also important to consider that these entities may need to use late binding. These are just two examples where DI would be useful.

When should I not use DI?

We don't need DI (in most cases) when stable dependencies are introduced into our solutions. When we talk about stable dependencies, we mean that we are using libraries that are not going to change. These are mostly libraries that reside in stable frameworks such as the .NET framework. Most classes in this framework have deterministic code that will not change over time.

It is important to ensure that the code in these frameworks is stable. Using libraries such as ADO .NET and Entity Framework are exceptions where it's better to use DI, even though they are a part of a stable framework.

What is Dependency Injection?

Now that we have a pretty good idea of what DI isn't, let's take a look at what it is. The main reason for DI is to help enforce loose coupling in your code. There are other aspects that are important such as testability, extensibility, parallel development, late binding, and maintainability. But loose coupling really encompasses all of the above items, with the exception of testability.

What do we mean by loosely coupled code? Loosely coupled code tends to adhere to the SOLID principals. The SOLID principals define five methods that help in designing well architected solutions.

S: The Single Responsibility Principle.

A class should be atomic. Classes should have a single reason to change. A database connection class should only be concerned with database connections and not CRUD, for instance.

O: The Open/Closed Principle.

Software entities should be open to extensibility and closed to modification.

L: The Liskov Substitution Principle.

Classes should be designed with contracts (abstractions) so that concrete implementations that use the abstractions (interfaces or abstract classes) can be easily changed without adverse reactions.

I: Interface Segregation Principle.

Program to fine-grained interfaces and not to general (course-grained) interfaces.

D: Dependency Inversion Principle.

Program to abstractions and not to concrete implementations. Dependency Injection uses this principle.

Why Use Dependency Injection?

Object Composition

DI allows us to defer object creation to the DIC of choice; this is called Object Composition. This gives us the ability to create objects in an almost transparent manner. In the Virtual Calculator solution, a number of services are registered with the Unity DIC. Constructor injection is then used to instantiate the objects when the parent object is created. This allows for the creation of object graphs that take responsibility for the creation of all dependent objects associated with a parent object. Unlike factories, which are designed to create a single object, DI not only creates the parent entity but also resolves any dependent entities.

Refactoring and extending code becomes easier because changes only need to be made in a single location. As long as the new code inherits from a base class or implements the same interface, concrete classes can be changed as needed.

DI is also important because it makes substituting classes much easier. This is particularly useful when working with Test Driven Development (TDD), because it is easy to substitute mock classes with the actual classes that are used in the solution when testing.

Object Lifetime

DICs also have the ability to manage object lifetime. Instance or singleton objects can be created by simply adding code when registering the class.

Listing 5: Using the Unity Container Controlled Lifetime Manager to Create a Singleton Class

```
Container.RegisterType<ICalculatorExceptionHandler,  
    CalculatorExceptionHandler>  
    (new ContainerControlledLifetimeManager());
```

Listing 5 shows an example of creating a singleton **CalculatorExceptionHandler** with the Unity DIC. A new **ContainerControlledLifetimeManager** is passed to the **RegisterType** method of the **Container** class. The unity DIC creates instance objects as a default. These objects are created and destroyed by the garbage collector when the enclosing entity goes out of scope. Singletons, on the other hand, once created, persist as single entities for the lifetime of the application.

Dependency Injection Containers

There are a number of DICs that either work out of the box or can be configured to work with Prism 4. Here is a list of some of these DICs:

- Unity *
- Managed Extensibility Framework (MEF) *
- Autofac
- Spring .NET
- Castle Windsor
- Structure Map

* Works with Prism 4 without the need to create a custom **Bootstrapper** class.

The Bootstrapper Class

The **Bootstrapper** class is the startup class that is responsible for the initialization of XAML-based solutions. The **Bootstrapper** class exposes a number of methods that can be used to configure both WPF and Prism 4 constructs. The rest of this chapter will be devoted to explaining how this class works and how the Unity DIC integrates with the class.

Listing 6: The Main Bootstrapper Class Definition

```
public class MainBootstrapper : UnityBootstrapper
{
}

```

Listing 6 shows an example of the **MainBootstrapper** class definition. Notice that the class inherits from the **UnityBootstrapper** class. The Unity DIC is used in all example code in this book.

To use the **MainBootstrapper** class with other DICs, simply inherit from a class that is associated with the DIC. Microsoft provides two DIC bootstrapper base classes that are ready to use. Classes are provided for Unity and MEF.

If you are familiar with another DIC other than these two (Unity and MEF), you can write a base class for your DIC of choice.

The Bootstrapper CreateShell Method

As we saw earlier in the Hello World solution, the **CreateShell** method is used to create and show the shell form. The shell form was designed to be as simple as possible. It only used a single label control. You'll find that the shell forms in the Virtual Calculator solution include quite a bit more functionally. We'll go into more detail in Chapter 6.

For now, let's look at the code that creates the shell forms.

Listing 7: The Main Bootstrapper CreateShell Method


```

private Window ShellForm;
private byte ShellSelector = 0;

protected override System.Windows.DependencyObject CreateShell()
{
    //Register services and objects with the Dependency Injection Container.
    CreateServices();

    //Create and show the Prism 4 shell form:
    ShellForm = CreateShellForm(ShellSelector);

    return ShellForm;
}

```

Unlike the Hello World solution, which has a single shell form, the Virtual Calculator solution has two shell forms. One form has logos at the top and the other has logos located on the bottom of the form.

A class-level member of the type byte, named **ShellSelector**, is used to determine which form is shown. A value of zero displays the logo-top form and a value of one displays the logo-bottom form. See Figures 8 and 9 to see examples of the forms. A byte type was used to reduce the complexity of adding more than two shell forms if necessary.

We'll talk about the call to the create services method later in this chapter.

The CreateShellForm Method

The **CreateShellForm** method is used to instantiate one of the two shell forms when the solution starts. The **ShellSelector** value is passed as an argument to the method and a **Window** control is returned. Here is an example of the **CreateShellForm** code:

Listing 8: The Main Bootstrapper CreateShell Method

```

private Window CreateShellForm(byte ShellSelector)
{
    if (ShellSelector == 0)
    {
        var ShellConverter = Container.Resolve<ICreateShellForm>();
        return ShellForm = ShellConverter.CreateShellForm();
    }
    else
    {
        var ShellConverter = Container.Resolve<ICreateShellForm>("LogoBottom");
        return ShellForm = ShellConverter.CreateShellForm();
    }
}

```

The **CreateShellForm** method could be refactored to eliminate the conditional code (if-then-else). This would reduce the amount of code that is needed to extend the functionality.

This code is interesting because the strategy design pattern is used to create the shell forms. Rather than create and call methods in the **Bootstrapper** class or hardcode the shell creation code into the **CreateShellForm** method, an interface was created—**ICreateShellForm**. Classes were then created for each of the two forms. These classes implement the **ICreateShellForm** interface. Here is an example of the **ICreateShellForm** code:

Listing 9: The **ICreateShellForm** Interface

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace PRISM4.MAIN.CLASSES
{
    public interface ICreateShellForm
    {
        Window CreateShellForm();
    }
}
```

As you can see, the **ICreateShellForm** interface has one method, **CreateShellForm**. This method returns a **Window** control.

The two classes **CreateShellLogoTop** and **CreateShellLogoBottom** actually create the forms. Here is an example of the **CreateShellLogoTop** code:

Listing 10: The **CreateShellLogoTop** Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

using Microsoft.Practices.Unity;
using Microsoft.Practices.Prism.UnityExtensions;

using PRISM4.MAIN.FORMS;

namespace PRISM4.MAIN.CLASSES
{
    public class CreateShellLogoTop : ICreateShellForm
    {
        private Window Shell;

        public CreateShellLogoTop(ShellLogoTop Shell)
        {
            if (Shell != null)
            {

```

```

        this.Shell = Shell;
        CreateShellForm();
    }
}

public Window CreateShellForm()
{
    Shell.Show();
    return Shell;
}
}
}

```

The class implements **ICreateShellForm**. The interesting code is the constructor of the class. Notice that a type of **ShellLogoTop** is passed to the constructor. Constructor injection is being used to inject the correct window type into the class. Now what this means to us is that the only difference between the **ShellLogoTop** and **ShellLogoBottom** class is the value that is passed to the constructor.

If we go back and look at the **CreateShellForm** method in the bootstrapper, you may notice that neither of the two constructors pass a type. In fact, one of the constructors passes no value at all!

Listing 11: The Main Bootstrapper CreateShell Method

```

private Window CreateShellForm(byte ShellSelector)
{
    if (ShellSelector == 0)
    {
        var ShellConverter = Container.Resolve<ICreateShellForm>();
        return ShellForm = ShellConverter.CreateShellForm();
    }
    else
    {
        var ShellConverter = Container.Resolve<ICreateShellForm>("LogoBottom");
        return ShellForm = ShellConverter.CreateShellForm();
    }
}
}

```

So, what's going on? To understand how this code works we need to take a look at the **CreateServices** Method.

The CreateServices Method

The **CreateServices** method is used to register services with the Unity DIC. Once these services are registered the container can then resolve the entities. When we talk about resolving entities in Unity, we mean Instantiate.

Listing 12 shows a partial view of the **CreateServices** method.

Listing 12: The Main Bootstrapper CreateServices ICreateShellForm Code

```
private void CreateServices()
{
    ...

    //Setup the Strategy Design Pattern for the shell form:
    Container.RegisterType<ICreateShellForm,
    CreateShellLogoTop>();

    Container.RegisterType<ICreateShellForm,
    CreateShellLogoBottom>("LogoBottom");

    ...
}
```

Each line of code uses the DIC to register types of **ICreateShellForm**. The first line registers a concrete entity of type **CreateShellLogoTop**. The second line registers a concrete entity of type **CreateShellLogoBottom**. Each of the concrete types corresponds to the two classes that were created in conjunction with the **ICreateShellForm** interface.

If you look closely at the code, you'll see that the two lines of code are not identical. The first line passes no value to the container's **RegisterType** method. In the case of the second line of code, the "LogoBottom" string is passed. Let's talk about why we do things in this manner.

If we look at the strategy design pattern we see that the intent of the pattern is to allow for code extensibility. In other words, we want to make it as easy as possible to add new shell forms to our solution. We also want to adhere to the SOLID Open/Closed principle. We want our code to be open for extensibility and closed to modification.

So, why the different code? When we register a single concrete class that implements an interface with Unity, that object is associated with the interface. So when the interface is resolved, Microsoft Unity knows to use the associated concrete class.

But what about when multiple concrete objects are implemented with a single interface? If we were to create three objects using the first line of code in Listing 12, only the last object registered would be resolved by Microsoft Unity. The first two registered objects would be discarded.

So, how do we resolve multiple entities that use the same interface? We have two options. The first option is to make one entity the default entity by leaving the **RegisterType** name argument blank. Each additional entity that is associated with the interface or abstract class would pass a unique name to the **RegisterType** method. Listing 12 shows an example of a default and named shell form that uses the same interface. The **ShellFormTop** window is the default entity, a name is not passed to the method. The **ShellFormBottom** window, on the other hand, does pass a name to the method (**LogoBottom**). This unique name differentiates the default logo top entity from the logo bottom entity,

The second method of registering multiple concrete entities with a single abstraction is to name each entity that is registered.



Note: Keep in mind that if you name all entities that use the same interface, there is no default entity, and if you try to resolve a no-argument entity, Unity will raise an exception.

Now it becomes clear why in the first case in the **CreateShellForm** method we pass no name, while the second line of code passes the "LogoBottom" instance name.

Later we'll see how to resolve named instances in code when they do not reside in the bootstrapper.

You may find it interesting that the **CreateServices** method is the only code in the solution that registers entities with the Unity DIC. In fact, the only place where we explicitly resolve entities is in the **CreateShellForm** method. We'll go into details about why we do things this way later.

The last thing that we do in the **CreateShell** method is to assign the returned window to the class-level member.

The Bootstrapper CreateModuleCatalog Method

The **CreateModuleCatalog** method is used to load modules at runtime. There are a number of methods that can be used to load modules with Prism 4. In this solution we use the configuration file (App.config) to load modules. Here is the code for the bootstrapper **CreateModuleCatalog** method in the Virtual Calculator solution:

Listing 13: The Main Bootstrapper CreateModuleCatalog Method

```
protected override IModuleCatalog CreateModuleCatalog()
{
    ConfigurationModuleCatalog configurationCatalog = new ConfigurationModuleCatalog();
    return configurationCatalog;
}
```

We will defer going into more detail about modules for now; Chapter 9 is dedicated to this subject.

The Bootstrapper ConfigureRegionAdapterMappings Method

The **ConfigureRegionAdapterMappings** method is used to add Region Adapters to the Prism 4 **RegionAdapterMappings** class. Here is an example of the method from the Virtual Calculator solution:

Listing 14: The Main Bootstrapper ConfigureRegionAdapterMappings Method

```
protected override RegionAdapterMappings ConfigureRegionAdapterMappings()
{
    RegionAdapterMappings mappings = base.ConfigureRegionAdapterMappings();

    if (mappings != null)
    {
        mappings.RegisterMapping(typeof(TabControlExt),
```

```
        this.Container.Resolve<SyncfusionTabCtrlRegionAdapter>());  
    }  
  
    return mappings;  
}
```



Note: We will defer going into more detail about regions and region adapters in this chapter. The reason that we need the region adapter code is to allow non-Microsoft user controls to work with Prism 4 regions. In Chapter 8, we will go into detail about how Prism 4 region adapters are configured and used.

The Composition Root

The composition root of a solution is the point where program execution starts. The composition root varies depending on the type of solution that is being built. In WPF solutions the default composition root is located in the App.xaml file. You'll recall that we changed where the solution started in the *Hello World* solution from the App.xaml to the App.xaml.cs file. In Microsoft Prism 4 solutions this is where the composition root resides. For our purposes this is not exactly true.

You may be wondering why I'm suddenly talking about the composition root and how it relates to Prism 4. It's actually quite important when working with Prism 4 and Dependency Injection.

One issue that comes up when working with DICs is that once you choose a DIC, your solution is tightly coupled to that container. You may have noticed that any tight coupling when it comes to programming is considered a bad practice; if we add DIC code throughout our solutions, it becomes difficult to change DICs. In other words, when we commit to a DIC, the syntax that we use is coupled to that container and if it is used throughout the solution, it's very difficult to change DICs. How can we reduce the impact of using a DIC in our solutions?

That's where the Composition root comes into play. If you look at all of the code in the *Virtual Calculator* solution, you'll find that there is not one place outside of the bootstrapper where Microsoft Unity code is used.

Why is this important? When the DIC code is restricted to the composition root, your solution comes as close as possible to being DIC agnostic. Earlier when I said that the App.xaml.cs file is not exactly the composition root for our purposes, this is why. The **Bootstrapper** class in Prism 4 solutions is considered to be the composition root because it is where the DIC is created.

Now what are the advantages to using the composition root for all of our DIC code?

- All DIC Code is in one place. If changes are necessary for the DIC, you know where to go.
- If for some reason the DIC needs to be changed, all of the changes can be done in the composition root. The solution only knows that constructor, property, or method injection is being used to resolve entities.
- Specific DIC code syntax is limited to the composition root.

In the **Bootstrapper** class of the *Virtual Calculator*, the **CreateServices** method is used to register services. Here is the full code for that method:

Listing 15: The Main Bootstrapper CreateServices Method

```
private void CreateServices()
{
    Container.RegisterType<IValidateInteger,
        Validate32BitSignedInteger>();

    Container.RegisterType<IValidateInteger,
        Validate64BitSignedInteger>("IntegerLong");

    Container.RegisterType<IntegerValidationResult>();

    //Setup the Strategy Design Pattern for the shell form:
    Container.RegisterType<ICreateShellForm,
        CreateShellLogoTop>();

    Container.RegisterType<ICreateShellForm,
        CreateShellLogoBottom>("LogoBottom");

    Container.RegisterType<ICalculatorExceptionHandler,
        CalculatorExceptionHandler>
        (new ContainerControlledLifetimeManager());

    Container.RegisterType<ExceptionResult>();

    Container.RegisterType<IStatus,
        OkStatus>();

    Container.RegisterType<IStatus,
        CalculatingStatus>("CalculateStatus");

    Container.RegisterType<IFormula,
        AddFormula>();

    Container.RegisterType<IFormula,
        SubtractFormula>("SubtractFormula");

    Container.RegisterType<IConvertInteger,
        ConvertInteger>();

    Container.RegisterType<ICalculate,
        AddIntegers>();

    Container.RegisterType<ICalculate,
        SubtractIntegers>("Subtract");
}
```

```

        Container.RegisterType<IDescription,
            Description>();

        Container.RegisterType<IUpdateUIState,
            TextboxOneAndTwoBlankOrClearState>();

        Container.RegisterType<IUpdateUIState,
            TextboxOneExceptionState>
            ("TextBoxOneException");

        Container.RegisterType<IUpdateUIState,
            TextboxTwoExceptionState>
            ("TextBoxTwoException");

        Container.RegisterType<IUpdateUIState,
            TextboxOneAndTwoExceptionState>
            ("TextBoxOneAndTwoException");

        Container.RegisterType<IUpdateUIState,
            TextboxOneHasValueState>
            ("TextBoxOneHasValue");

        Container.RegisterType<IUpdateUIState,
            TextboxOneAndTwoHasValueState>
            ("TextBoxOneAndTwoHaveValues");

        Container.RegisterType<UpdateUserInterface>();
        Container.RegisterType<UpdateSubtractionUserInterface>();

        Container.RegisterType<Signed32BitIntegerConversionResult>();
        Container.RegisterType<Signed64BitIntegerConversionResult>();

        Container.RegisterType<IDescription,
            Description>();

        Container.RegisterType<DescriptionResult>();
    }

```

Without the Prism 4 **Bootstrapper** class, it would be necessary to resort to calling forms from the App.xaml file or writing initialization code from scratch.

Summary

In this chapter we spoke about DI and the Prism 4 bootstrapper. We talked about the different DICs and looked in detail at the Unity DIC. We talked about why DI and DICs are important and looked at how DICs are integrated into Prism 4 bootstrapper classes. We saw how to configure a DIC when using the strategy design pattern and learned that we have options when doing so.

We also spoke about the composition root and why it is important when using DICs. We covered different kinds of injection types that are used to resolve entities and why this method of entity resolution is desired.

In the next chapter, we'll take a look at the Prism 4 shell form and get our first detailed look at XAML markup.

Chapter 6 The Prism 4 Shell Form

The Prism 4 shell form is the template that defines what the user of your solution sees. It provides layout and dynamic view population with Prism 4 regions and affects the workflow and feel of your solution. The shell form is the container for the solution's visual and business content. It is the first thing that a user sees when using your solution. If first impressions are important, the shell form of your solution is the equivalent of a firm handshake and good eye contact.

What is XAML?

Although the shell form is used with Prism 4 solutions, it's actually a WPF container control. Being a WPF control, it uses some completely different methodologies (in comparison to Windows controls, for instance) when it comes to rendering User Interfaces (UI). Direct X is used to render graphics rather than the older GDI and GDI+ libraries that were used with previous technologies. This means that WPF solutions are able to take advantage of the advanced graphic rendering abilities of today's graphic cards.

It also means that we get a new markup language that's much more powerful than previous languages. Extensible Application Markup Language (XAML) is derived from the XML specification. It is an advanced markup language that addresses many of the issues that are associated with older markup languages.

XAML's power comes from the fact that it can perform tasks that were relegated to scripting languages in the older markup languages. XAML releases developers from the necessity of learning both a markup language and scripting language. Features such as data binding, data templates, control templates, data context, and advanced layout make XAML an easy-to-learn and powerful markup language.

Listing 16: The ShellLogoTop XAML Markup

```
<Window
    x:Class="PRISM4.MAIN.FORMS.ShellLogoTop"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:syncfusion="http://schemas.syncfusion.com/wpf"
    xmlns:prism="http://www.codeplex.com/prism"

    Title="Virtual Calculator 1.0.0"
    Height="auto"
    Width="auto"
    WindowState="Maximized">

    <Window.Background>
        <ImageBrush
            ImageSource=
```

```

        "/PRISM4.MAIN;component/IMAGES/Calculator.jpg"/>
</Window.Background>

<Grid
    ShowGridLines="false">

    <Grid.RowDefinitions>

        <RowDefinition
            Height="auto">
        </RowDefinition>

        <RowDefinition
            Height="*">
        </RowDefinition>

    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>

        <ColumnDefinition
            Width="30*">
        </ColumnDefinition>

        <ColumnDefinition
            Width="8*">
        </ColumnDefinition>

        <ColumnDefinition
            Width="62*">
        </ColumnDefinition>

    </Grid.ColumnDefinitions>

    <Grid
        Name="HeaderGrid"
        Grid.Row="0"
        Grid.Column="1"
        Grid.ColumnSpan="2"
        Margin="0,10,10,10">

        <Grid.ColumnDefinitions>

            <ColumnDefinition
                Width="auto">
            </ColumnDefinition>

            <ColumnDefinition
                Width="*">
            </ColumnDefinition>

        </Grid.ColumnDefinitions>

    </Grid>

    <ContentControl
        Name="CompanyLogo"
        Grid.Row="0"

```

```

        Grid.Column="0"
        Margin="0"
        HorizontalAlignment="Left"
        prism:RegionManager.RegionName="CompanyRegion">
    </ContentControl>

    <ContentControl
        Name="ApplicationLogo"
        Grid.Row="0"
        Grid.Column="1"
        Margin="0"
        prism:RegionManager.RegionName="ApplicationRegion">
    </ContentControl>

</Grid>

<Border BorderBrush="Red"
        BorderThickness="5"
        CornerRadius="5"
        Grid.Row="1"
        Grid.Column="1"
        Grid.ColumnSpan="2"
        Margin="0,10,20,30"
        Opacity="0.7"
        HorizontalAlignment="Stretch">

    <Border.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="LightSteelBlue" Offset="0"/>
            <GradientStop Color="LightSteelBlue" Offset="1"/>
            <GradientStop Color="White" Offset="0.535"/>
        </LinearGradientBrush>
    </Border.Background>

    <syncfusion:TabControlExt
        Name="MathControl"
        CloseButtonType="Hide"
        prism:RegionManager.RegionName="MathRegion"
        IsSynchronizedWithCurrentItem="True"
        SelectOnCreatingNewItem="False">
    </syncfusion:TabControlExt>

</Border>

</Grid>

</Window>

```

Let's take a look at the markup in Listing 16.

Namespaces

One of XAML's virtues is the ability to easily associate class libraries to the XAML markup. This is done with namespaces. Namespaces are added by first setting a project reference to the required DLL. Next, a namespace is created in the XAML markup that references the correct library.

Listing 17: Example of the Prism 4 XAML Namespace

```
xmlns:prism="http://www.codeplex.com/prism"
```

Listing 17 shows an example of the Prism 4 namespace in the **ShellLogoTop** form of the *Virtual Calculator*.

The Syncfusion Namespace

The *Virtual Calculator* solution uses a Syncfusion Tab control in the *Math* project. The Syncfusion namespace provides the library that is needed to use this control in XAML.

Listing 18: Example of the Syncfusion XAML namespace

```
xmlns:syncfusion="http://schemas.syncfusion.com/wpf"
```

Listing 18 shows an example of the markup needed to create the Syncfusion namespace.

The Prism Namespace

The *Virtual Calculator* solution also needs to create a namespace for the Prism 4 library. Listing 17 shows an example of this namespace.

The Content Control Tag

Content controls are controls that show single views. When a new view is added to a content control, the previous view is replaced with the new one. Two content controls are used in the *Virtual Calculator* shell forms. The first control provides a Prism 4 region that is populated by the company logo view. The second control provides a Prism 4 region that is populated by the application logo view.

Listing 19: The CompanyLogo Content Control

```
<ContentControl
    Name="CompanyLogo"
    Grid.Row="0"
    Grid.Column="0"
    Margin="0"
    HorizontalAlignment="Left"
    prism:RegionManager.RegionName="CompanyRegion">
</ContentControl>
```

How is a regular control converted to a Prism 4 region? The **RegionManager.RegionName** attached property is used to identify a user control as a Prism 4 region. Listing 19 shows an example of the markup that identifies the **CompanyLogo** content control with the region name of **CompanyRegion**. When we look at Prism 4 modules in Chapter 9, we'll see how Prism 4 uses the region name to dynamically populate regions with views at runtime.



Note: The reason why we can use the Prism 4 `RegionManager.RegionName` attached property, is because we added the prism namespace to the Window. Without this namespace there would be no access to the property.

The Syncfusion:TabControlExt Tag

The Syncfusion tab control is used as the view for the *Math* module in the *Virtual Calculator* solution. This type of control is a selector control. Unlike the content control, selector controls allow for multiple views to be shown in a single control. The Syncfusion tab control in the *Virtual Calculator* solution has two tabs: *add two integers* and *subtract two integers*. Each of these tabs are separate views that are injected into the **MathRegion** region at runtime.

Listing 20: The MathControl Tab Control

```
<syncfusion:TabControlExt
    Name="MathControl"
    CloseButtonType="Hide"
    prism:RegionManager.RegionName="MathRegion"
    IsSynchronizedWithCurrentItem="True"
    SelectOnCreatingNewItem="False">
</syncfusion:TabControlExt>
```

This control also uses the `RegionManager.RegionName` attached property to convert the control to a Prism 4 region. Listing 20 shows an example of the markup that identifies the **MathControl** tab control with the region name of **MathRegion**.

In the case of the Syncfusion tab control, we have to do a bit of work to have the control integrate correctly with Prism 4. In Chapter 8, "Prism 4 Regions", we'll look at region adapters. These classes are used with third-party controls to adapt them for integration with Prism 4.



Note: Adding the `RegionManager.RegionName` property to the tab control without a region adapter class for the Syncfusion tab control will result in an exception. We'll talk about what region adapters do and how to use them in Chapter 8.



Tip: We don't need a region adapter for the content control because Microsoft controls already have them. Only third-party controls need customized region adapter classes.

XAML Element And Data Binding

Unlike other markup languages, XAML provides built-in, data-binding services. This allows the view to bind to public view model properties that are exposed through the view's data context. It is also possible to bind between the user control properties of different controls in the view; this is called *element binding*.

Listing 21 shows an example of data binding between the **AddTwoView** view and the **AddTwoViewModel.TabHeaderText** view model property.

Listing 21: Binding to a Tab's Header Property with XAML Data Binding

```
...
Header="{Binding Path=TabHeaderText}"
...
```

The ability to bind between controls and to bind between the view and view model are powerful features of the XAML markup language. We'll look closer at data binding and other XAML constructs when we explore MVVM in Chapter 7.

The View's Data Context Property

At this point you may be wondering how the view accesses the view model properties. The view's **DataContext** property provides this functionality. When the data context of a view is set to a view model class, all of the public properties in that class are exposed to the view.

The **INotifyPropertyChanged** interface is also exposed through the view model. This interface is used to ensure that the view is updated when changes are made to the view model properties. We'll talk in greater detail about **INotifyPropertyChanged** when we look at the view model in Chapter 7.

Views can be associated with view models in two ways: with XAML markup or with C# code in the view's code behind file. Listing 22 shows data context code in the code behind file.

Listing 22: Setting A view's Data Context with Code Behind

```
public partial class AddTwoView : TabItemExt
{
    private AddTwoViewModel ViewModel;

    public AddTwoView(AddTwoViewModel ViewModel)
    {
        InitializeComponent();

        if (ViewModel != null)
        {
            this.ViewModel = ViewModel;
        }

        this.DataContext = this.ViewModel;
    }
}
```

```
}
```

Notice that we use a constructor injection to instantiate the view model. We then use a guard clause to make sure that a non null view model object is available for use. The view model argument is assigned to a local member if it is not null.

Next, we set the **DataContext** property of the view to the local view model member.

The other option is to use XAML markup to set the view model. Listing 23 shows an example of using this option.

Listing 23: Setting A view's Data Context with XAML

```
<syncfusion:TabItemExt.DataContext>  
    <vm:AddTwoViewModel>  
    </vm:AddTwoViewModel>  
</syncfusion:TabItemExt.DataContext>
```

The markup in Listing 23 can be added to the **syncfusion:TabItemExt** control's XAML markup directly.



Note: The *syncfusion:* and *vm:* tags are namespaces that were added to the to the view from referenced libraries in the project. The *vm:* tag references the view model class. The *syncfusion:* tag references the main syncfusion DLL.

Summary

In this chapter we learned that the shell form is the first thing that users see. Without the shell form, it would be difficult or impossible to provide user interaction with Prism 4 solutions. The shell form provides a method of dynamic user interface construction as the solution starts and during its lifetime. This allows for loose coupling between the shell form and the views that populate it.

We also learned that XAML is a powerful and versatile markup language. It is used to render user interfaces and to provide a means of connecting the UI elements to the business rules through the view model. As we progress further into WPF and the Prism 4 framework, we'll learn about some of the other features of this markup language.

But at this time let's look at another aspect of well designed solution architectures: the Model–View–ViewModel design pattern.

Chapter 7 Prism 4 and MVVM

The Model–View–ViewModel (MVVM) design pattern is not included in the Prism 4 library, but is a particularly well-suited companion to the framework. MVVM is all about separation of concerns. Its main objective is to isolate the solution's views from its business rules. The MVVM pattern consists of three parts: the view, the view model, and the model.

The MVVM View

Views contain the User Interfaces (UI) in the solution. Views are how users interact with the solution.

In Prism 4 solution views can be user controls, data templates, or almost any container control. Prism 4 views can also be injected into regions on the shell form or into other views. This allows for dynamic construction of user interfaces at runtime. We'll show how to inject views into regions in Chapter 9.

Listing 24: A Listing Of The AddTwoView View

```
<syncfusion:TabItemExt
    x:Class="PRISM4.MATH_MODULE.VIEWS.AddTwoView"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
    xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"

    xmlns:syncfusion="http://schemas.syncfusion.com/wpf"

    x:Name="AddTwo"
    mc:Ignorable="d"
    d:DesignWidth="300"
    Header="{Binding Path=TabHeaderText}"
    Height="auto">

    <i:Interaction.Triggers>

        <i:EventTrigger
            EventName="Loaded">

            <i:InvokeCommandAction
                Command="{Binding TabLoadedCommand}">
            </i:InvokeCommandAction>

        </i:EventTrigger>

    </i:Interaction.Triggers>

</Grid
```



```

Name="MainGrid">

<Grid.RowDefinitions>

...
...

<!--*****
ADD Interaction Behaviors here.
*****-->
<i:Interaction.Behaviors>

    <ei:DataStateBehavior
        x:Name="TextBox1ExceptionBehavior"
        Binding="{Binding TextBox1Exception}"
        Value="true"
        TrueState="TextBox1ExceptionStatusTrueState"
        FalseState="TextBox1ExceptionStatusFalseState">
    </ei:DataStateBehavior>

...

</i:Interaction.Behaviors>

<!--*****
Add Visual State Manager (VSM) groups here.
*****-->
<VisualStateManager.VisualStateGroups>

    <VisualStateGroup
        x:Name="Exception1TextBoxStates">

        <VisualState
            x:Name="TextBox1ExceptionStatusTrueState">

            <Storyboard>

                <BooleanAnimationUsingKeyFrames
                    Storyboard.TargetProperty="IsEnabled"
                    Storyboard.TargetName="CalculateButton">

                    <DiscreteBooleanKeyFrame
                        KeyTime="0"
                        Value="false">
                    </DiscreteBooleanKeyFrame>

                </BooleanAnimationUsingKeyFrames>

            ...

        </VisualState>
    </VisualStateGroup>

    <Border
        Grid.Row="0"
        Grid.Column="0"
        Grid.RowSpan="5"

```

```

        BorderBrush="DarkBlue"
        BorderThickness="3"
        CornerRadius="5"
        Margin="10">

        <ScrollView
            VerticalScrollBarVisibility="Auto">

            <TextBlock
                Name="HelpText"
                Text="{Binding Path=DescriptionTextboxText}"
                Height="auto"
                Width="Auto"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                TextWrapping="WrapWithOverflow"
                FontWeight="Bold"
                FontSize="16"
                Foreground="Blue"
                Padding="5">

                <TextBlock.Background>
                    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                        <GradientStop Color="LightSteelBlue" Offset="0"/>
                        <GradientStop Color="LightSteelBlue" Offset="1"/>
                        <GradientStop Color="White" Offset="0.556"/>
                    </LinearGradientBrush>
                </TextBlock.Background>

            </TextBlock>

        </ScrollView>

    </Border>

    <ScrollView
        Grid.Row="0"
        Grid.Column="1"
        VerticalScrollBarVisibility="Auto"
        Margin="0,0,5,0">

        <GroupBox
            Name="ContentGroupBox"
            BorderBrush="DarkBlue"
            BorderThickness="3"
            Header="{Binding Path=GroupBoxHeaderText}"
            FontWeight="Bold"
            Padding="2">

            <GroupBox.Background>
                <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                    <GradientStop Color="LightSteelBlue" Offset="0"/>
                    <GradientStop Color="LightSteelBlue" Offset="1"/>
                    <GradientStop Color="White" Offset="0.541"/>
                </LinearGradientBrush>
            </GroupBox.Background>

```

```

<Grid
  Name="CalculationGrid"
  HorizontalAlignment="Stretch"
  VerticalAlignment="Stretch"
  Margin="5,10">
...

  <Label
    Name="Integer1Label"
    Grid.Row="0"
    Grid.Column="0"
    FontWeight="Bold"
    FontSize="14"
    Content="Integer One: "
    Margin="0,0,0,3">
  </Label>

  <TextBox
    Name="Integer1TextBox"
    Grid.Row="0"
    Grid.Column="1"
    FontWeight="Bold"
    FontSize="14"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"
    VerticalContentAlignment="Center"
    Margin="0,0,0,3"
    Text="{Binding Path=TextBoxOneText}">

    <i:Interaction.Triggers>

      <i:EventTrigger
        EventName="TextChanged">

        <i:InvokeCommandAction
          Command="{Binding TextBoxOneTextChangedCommand}"
          CommandParameter="{Binding
ElementName=Integer1TextBox, Path=Text}">
        </i:InvokeCommandAction>

      </i:EventTrigger>

    </i:Interaction.Triggers>

  </TextBox>
...

  <Button
    Name="ClearButton"
    Content="Clear"

```

```

        Grid.Row="3"
        Grid.Column="1"
        FontWeight="Bold"
        Foreground="DarkBlue"
        FontSize="16"
        Margin="0,5,0,0">

        <i:Interaction.Triggers>

            <i:EventTrigger
                EventName="Click">

                <i:InvokeCommandAction
                    Command="{Binding ClearButtonClickCommand}">
                </i:InvokeCommandAction>

            </i:EventTrigger>

        </i:Interaction.Triggers>
    </Button>

    <Button
        Name="CalculateButton"
        Content="Calculate"
        Grid.Row="0"
        Grid.Column="2"
        Grid.RowSpan="4"
        FontWeight="Bold"
        Foreground="DarkBlue"
        FontSize="16"
        Margin="5,0"
        Padding="10"
        IsDefault="True">

        <i:Interaction.Triggers>

            <i:EventTrigger
                EventName="Click">

                <i:InvokeCommandAction
                    Command="{Binding CalculateButtonClickCommand}">
                </i:InvokeCommandAction>

            </i:EventTrigger>

        </i:Interaction.Triggers>
    </Button>

</Grid>

</GroupBox>

</ScrollView>

```

...

```
        </ScrollView>

        </Border>

    </Grid>

</syncfusion:TabItemExt>...
```

The first thing that you'll notice about Listing 24 is that it's a lot of markup...and this is only a partial listing! This is because the power of XAML makes it possible to do more than simply render UIs. In addition to rendering the UI, the markup in Listing 24 also exposes behaviors and states that are used to update the UI as the user interacts with it during the lifetime of the solution.

If you look closely at the markup in Listing 24, you'll notice that a standard user control is not being used. The reason for this is that if we use a regular user control, we'll add a tab to the tab control for each view that we inject, but the tab headers will be blank. There is not a property in user controls that will allow for the population of the tab's header property. The solution to this issue is to use a *Syncfusion TabItemExt* control (`<syncfusion:TabItemExt>`). This control has a header property and allows for the population of the property.

State-Based Navigation

Another interesting aspect of this view is that Prism 4 State-Based Navigation is used to update the user interface. Each state is implemented with an Expression Blend behavior and an associated State.

State-based navigation allows the user interface to transition between different states. State based-navigation changes the view without affecting data or underlying business rules.



Note: I could have used a Microsoft mask edit control or one of Syncfusion's numeric controls such as the Integer textbox control. These controls would greatly reduce the amount of validation code needed in the solution. I used regular textbox controls to show one method of validation in this type of solution.

State-Based Navigation Behaviors

In Listing 25, a **DataStateBehavior** is used to trigger one of two states. Each state executes animation storyboards that change different properties in the user interface's controls.

The **binding** property is associated with a **Boolean** property that resides in the view model. This property is used to set the behavior to one of two states. We'll talk more about this when we look at the view model.

A true state triggers the **TextBox1ExceptionStatusTrueState**. This state is triggered when there is an exception. The **TextBox1ExceptionStatusFalseState** is triggered when there is not an exception.

The **Value** property contains the initial setting for the behavior, which is **true** in this case.

Listing 25: The **TextBoxOneExceptionBehavior** Markup

```
<ei:DataStateBehavior
  x:Name="TextBox1ExceptionBehavior"
  Binding="{Binding TextBox1Exception}"
  Value="true"
  TrueState="TextBox1ExceptionStatusTrueState"
  FalseState="TextBox1ExceptionStatusFalseState">
</ei:DataStateBehavior>
```

State-Based Navigation States

Listing 26 shows the **TextBox1ExceptionStatusTrueState**. This state is triggered when the **TextBox1Exception** property is set to **true**. This partial listing shows three storyboard key frames.

The **BooleanAnimationUsingKeyFrames** is used to set the **IsEnabled** property of the **Calculate** button. When an exception is thrown, the **IsEnabled** property value is set to false.

The first **ColorAnimation** changes the foreground color of the **Integer1Textbox** to white. The second **ColorAnimation** changes the background color of the **Integer1Textbox** to red.

These changes are used to give the UI a visual indication that there is an exception. Disabling the **Calculate** button stops the user from passing invalid data for calculation.

Listing 26: A Partial Listing Of The **TextBox1ExceptionStatusTrueState** Markup

```
<VisualState
  x:Name="TextBox1ExceptionStatusTrueState">
  <Storyboard>
    <BooleanAnimationUsingKeyFrames
      Storyboard.TargetProperty="IsEnabled"
      Storyboard.TargetName="CalculateButton">
      <DiscreteBooleanKeyFrame
        KeyTime="0"
        Value="false">
      </DiscreteBooleanKeyFrame>
    </BooleanAnimationUsingKeyFrames>
    ...
  </Storyboard>
</VisualState>
```

```

        <ColorAnimation
            Storyboard.TargetProperty="Foreground.Color"
            Storyboard.TargetName="Integer1TextBox"
            To="White">
        </ColorAnimation>

        <ColorAnimation
            Storyboard.TargetProperty="Background.Color"
            Storyboard.TargetName="Integer1TextBox"
            To="Red">
        </ColorAnimation>

        ...

    </Storyboard>

</VisualState>

```

The MVVM View Model

The view model serves as an intermediary between the view and the model. It exposes public properties to the view via the data context. (We looked at the data context in Chapter 6.) The view has knowledge of the view model through the data context and bindings; the view model on the other hand, has no knowledge of the views that it services. This is a very important aspect of the MVVM pattern. The view model exposes entities to views with no real information about the views to which they provide the services. This allows views to be designed as autonomous entities that simply use the services that the view model provides.

Listing 27 shows a partial listing of the **AddTwoViewModel**.

Listing 27: The AddTwoViewModel

```

public class AddTwoViewModel : INotifyPropertyChanged
{
    #region CLASS PROPERTIES AND MEMBERS

    #region CLASS CONSTANTS

        private const string AddIntegersTabHeaderText = "Add Two Integers";
        private const string AddIntegersGroupboxHeaderText = "Add Integers";

    #endregion //END CLASS CONSTANTS REGION

    #region CLASS SINGLE MEMBERS

        private IEventAggregator EventAggregator;
        private UpdateUIModel UIModel;
        private IUpdateUIState CurrentUIState;
        private bool ClearFlag = false;
        private ObservableCollection<string> IntegerStrings = new
        ObservableCollection<string>();

        private string IntegerTotal = "";
    
```

```

private AddTwoModel AddIntegers;

private IDescription Description;
private DescriptionResult DescriptionResult;

#endregion //END CLASS SINGLE MEMBERS REGION

#region GENERAL CLASS PROPERTIES AND MEMBERS

private string TAB_HEADER_TEXT;
public string TabHeaderText
{
    get
    {
        return TAB_HEADER_TEXT;
    }

    set
    {
        if (TAB_HEADER_TEXT != value)
        {
            TAB_HEADER_TEXT = value;
            NotifyPropertyChanged("TabHeaderText");
        }
    }
}
...

#region CLASS CONSTRUCTOR

public AddTwoViewModel
(IEventAggregator EventAggregator,
UpdateUIModel UIModel,
IUpdateUIState CurrentUIState,
AddTwoModel AddIntegers,
IDescription Description,
DescriptionResult DescriptionResult)
{
    if (EventAggregator != null)
    {
        this.EventAggregator = EventAggregator;
    }

    if (UIModel != null)
    {
        this.UIModel = UIModel;
    }

    if (AddIntegers != null)
    {
        this.AddIntegers = AddIntegers;
    }

    if (Description != null)
    {

```



```

        this.Description = Description;
    }

    if (DescriptionResult != null)
    {
        this.DescriptionResult = DescriptionResult;
    }

    SetDescriptionText();
    SetHeaderValues();
    ResetUserInterface();

    TAB_LOADED_COMMAND = new DelegateCommand(TabLoad);
    TEXT_BOX_ONE_TEXT_CHANGED_COMMAND = new
    DelegateCommand<string>(TextBoxOneTextChanged);
    TEXT_BOX_TWO_TEXT_CHANGED_COMMAND = new
    DelegateCommand<string>(TextBoxTwoTextChanged);
    CLEAR_BUTTON_CLICK_COMMAND = new DelegateCommand(ClearButtonClick);
    CALCULATE_BUTTON_CLICK_COMMAND = new DelegateCommand(CalculateButtonClick);
}

#endregion //END CLASS CONSTRUCTOR REGION

```

The INotifyPropertyChanged Interface

The **AddTwoViewModel** class implements the **INotifyPropertyChanged** interface. As we noted earlier, this interface is used to reflect changes made in the view model to the view. It should be implemented in all view model classes that contain public properties. Let's take a closer look at the code.

Listing 28: A Property and The INotifyPropertyChanged Code

```

private string TAB_HEADER_TEXT;
public string TabHeaderText
{
    get
    {
        return TAB_HEADER_TEXT;
    }

    set
    {
        if (TAB_HEADER_TEXT != value)
        {
            TAB_HEADER_TEXT = value;
            NotifyPropertyChanged("TabHeaderText");
        }
    }
}

...

#region I NOTIFY PROPERTY CHANGED

```

```

    public event PropertyChangedEventHandler PropertyChanged;

    private void NotifyPropertyChanged(string PropertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));
        }
    }

#endregion //END I NOTIFY PROPERTY CHANGED REGION

```

Listing 28 shows an example of the **INotifyPropertyChanged** code. The property **TabHeaderText** calls the **NotifyPropertyChanged** method. It passes its name to the method (**PropertyName**). A guard clause determines that the **PropertyChanged** event handler is not null. If the handler is not null, the event is triggered. The view model is passed to the sender argument of the event and the **PropertyName** is passed to the **PropertyChangedEventArgs** of the event.

Each public property in the view model that is used by the view should call the **NotifyPropertyChanged** method in its setter and pass its property name to the method.

The MVVM Model

The MVVM model is where we expose services. These services can be local or global. Services can be anything ranging from Data Access Layers (DAL) to business rules. The model is designed to have no knowledge of the view or view model, and provides the services needed to execute the solutions functionally.

The only class in the MVVM pattern that should have knowledge of the model is the view model. The view model accesses services from the model and exposes their returned values as public properties to the view. This provides the necessary separation of concerns between the view and model.

Each Prism 4 module in the solution should use the MVVM pattern to keep views and models separate. There are a number of ways to incorporate this separation of concerns. Figure 10 shows an example of the folder structure used in the *Virtual Calculator* solution.

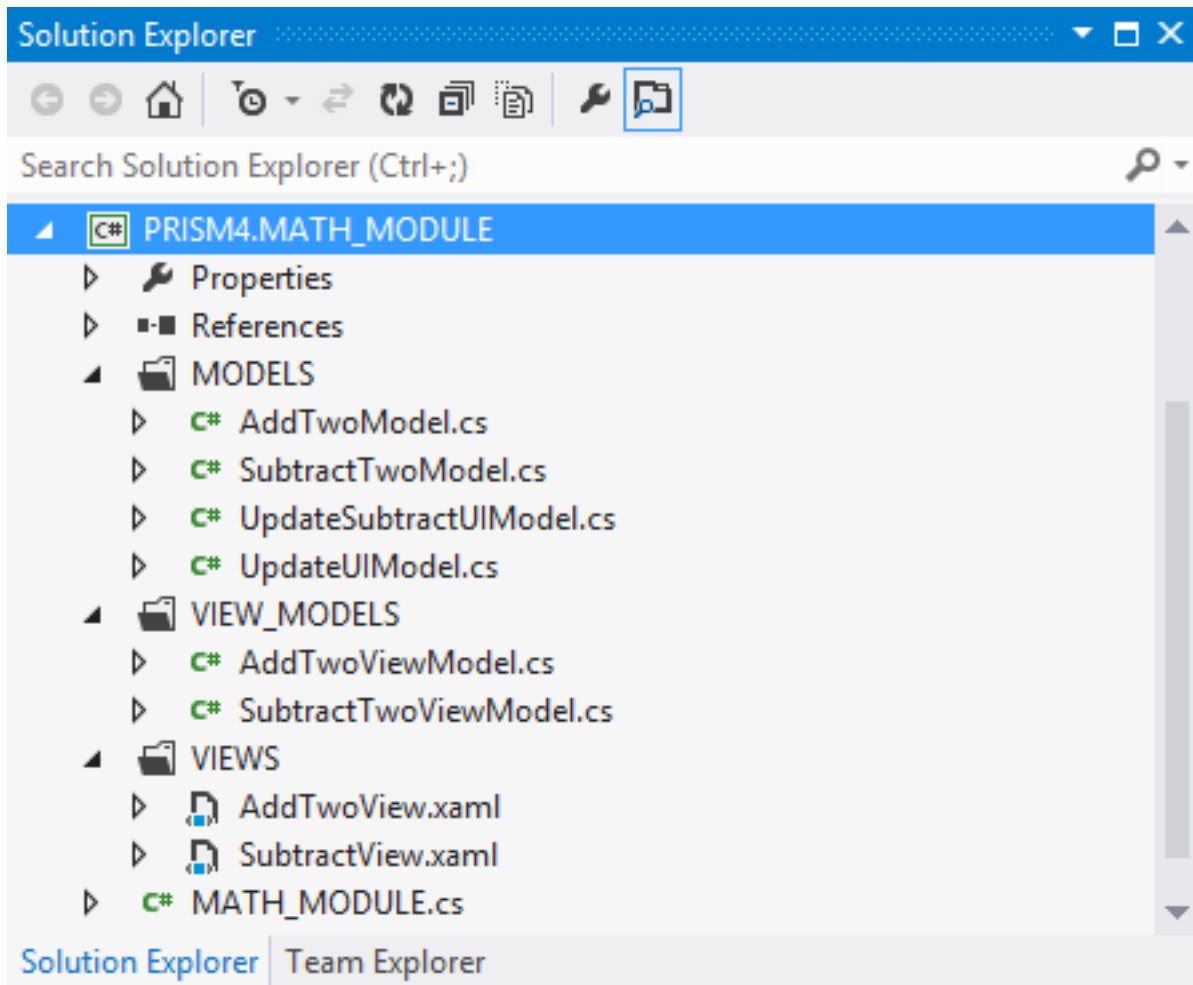


Figure 10: The Math Module Project Folder Layout

There are three folders in the module: MODELS, VIEW_MODELS, and VIEWS.

I use folders to separate the classes to better organize my solutions. This can be handy in large projects that contain numerous classes of different types.

Summary

Let's do a quick synopsis of MVVM. MVVM is not a Prism 4 construct, but is designed to work well with the framework. Its main purpose is to enforce separation of concerns between views and models. The view's data context is used to loosely couple the view model to the view. Views know about view models through the data context, but view models have little or no knowledge about views. View models know about models. Models have no knowledge of views or view models.

View models use the **INotifyPropertyChanged** interface to synchronize changes in the view model and view.

Although the MVVM design pattern is not a part of Microsoft Prism 4, it does serve as a powerful addition to solutions that use the framework

In the next chapter we'll talk about Prism 4 regions.

Chapter 8 Prism 4 Regions

Prism 4 regions are an important part of how we dynamically build User Interfaces (UI) in XAML-based solutions. In Chapter 6 we briefly looked at the `RegionManager.RegionName` property. We used this property to assign Prism 4 regions to user controls in our shell forms. In this chapter we'll take a closer look at Prism 4 regions and get a better idea of how to use them in our solutions.

What is a Prism 4 Region?

A Prism 4 region is a place holder in a window or view that identifies a section of the UI for dynamic injection of content.

How are Regions Used?

You may recall that when we talked about the shell form in Chapter 6, we added regions to the shell forms in that project.

Listing 29: The MathRegion Of The ShellLogoTop Shell Form

```
prism:RegionManager.RegionName="MathRegion"
```

The code in Listing 29 serves as a placeholder for math content in the shell form. A container control (in this case the Syncfusion `TabControlExt` control) is added to the shell form with no content. Listing 29 sets the control's `RegionName` attached property to `MathRegion`.

The `MathRegion RegionName` property will serve as an identifier that is used to inject views at runtime.



Note: We are going to defer going into detail about Prism 4 view injection at this time. We'll go into detail in the next chapter when we talk about Prism 4 modules.

Prism 4 Regions and Microsoft Controls

Adding Microsoft controls with Prism 4 regions is easy. Simply add the Prism 4 library to your window or view. Then add an appropriate container control and add a unique `RegionManager.RegionName` to that control.

There are three types of Microsoft container controls that work with Prism 4 out of the box: `ContentControl`, `ItemsControl`, and `Selector`.

The ContentControl

The content control can use a single injected view at a time. If a content control that already has a view injected is injected with a new view, the previous view is replaced with the new one. An example of content controls is the **ContentControl**.

The ItemsControl

The **ItemsControl** can use multiple injected items or views in a single container. The **ListBox** and **ComboBox** controls are examples of item controls.

The Selector

Selector controls can use multiple views in a single container also. Examples of selector controls are tab controls and dock managers.

These controls work with Prism 4 out of the box because the Prism 4 framework comes with three **RegionAdapter** implementations that are designed to work with Microsoft controls that derive from the three control types.

Built-in Prism 4 Region Adapters:

- **ContentControlRegionAdapter**
- **ItemsControlRegionAdapter**
- **SelectorRegionAdapter**

These region adapters are included when you install Prism 4. As long as you are using Microsoft controls that derive from these types, your code should work with no modifications.

Microsoft Types That Work with the Three Region Adapters:

- **System.Windows.Controls.ContentControl**
- **System.Windows.Controls.ItemsControl**
- **System.Windows.Controls.Primitives.Selector**

Using Prism 4 Regions With Third-Party Controls

Now that we've seen how region adapters work with Microsoft controls, let's look at an example where we use a third-party control. In the *Virtual Calculator* solution, two Microsoft-based content controls are used to display the company logo and application logo, and each of these modules work just fine.

In the case of the math module, we use Syncfusion's **TabControlExt** control. If we try to add the **RegionManager.RegionName** attached property to this control we get an error. The reason for this is that there is no **RegionAdapter** that works with the **TabControlExt** control in the Prism 4 library. So, how do we deal with this issue?



Tip: Most third-party control vendors supply *RegionAdapter* example code with their control suites. This is the case with at least three control suite vendors that I have worked with, including Syncfusion.

If your intention is to use Microsoft Prism 4 in your solutions with third-party controls, ensure that the control vendor's controls work with the framework before committing to the control suite.

The Syncfusion TabControlExt RegionAdapter Class

When working with third-party controls, it's necessary to create **RegionAdapter** classes for the controls that you want to use with Prism 4.



Note: This is the only area of Prism 4 where I have found it necessary to configure Prism 4 to work with third-party controls. In all other areas, I have found Prism 4 to work transparently with these controls.

You may be wondering why we create a special class in order to work with non-Microsoft controls. To understand why this is necessary, let's look at the **RegionAdapter** class for the Syncfusion **TabControlExt** control.

Listing 30: The SyncfusionTabCtrlRegionAdapter Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Practices.Prism.Regions;
using Syncfusion.Windows.Tools.Controls;
using System.Windows;

namespace PRISM4.MAIN.REGION_ADAPTERS
{
    public class SyncfusionTabCtrlRegionAdapter
        : RegionAdapterBase<TabControlExt>
    {
        public SyncfusionTabCtrlRegionAdapter
            (IRegionBehaviorFactory regionBehaviorFactory)
            : base(regionBehaviorFactory)
        {
        }

        protected override void Adapt(IRegion region, TabControlExt regionTarget)
        {
            region.Views.CollectionChanged += delegate
            {
                foreach (var tab in region.Views.Cast<FrameworkElement>())
            }
        }
    }
}
```

```

        {
            if (!regionTarget.Items.Contains(tab))
            {
                regionTarget.Items.Add(tab);
            }
        };
    }

    protected override IRegion CreateRegion()
    {
        return new Region();
    }
}

```

The first thing to take into account is that the **RegionAdapter** class inherits a generic class of type **RegionAdapterBase**. This base class takes a payload of **TabControlExt**, the Syncfusion tab control.

Next constructor injection is used to instantiate a type of **IRegionBehaviorFactory** this interface derives from **Microsoft.Practices.Prism.Regions**. The **RegionBehaviorFactory** class is then passed to the base class.

[Listing 31: Using Constructor Injection and Passing the regionBehaviorFactory to the base class](#)

```

public SyncfusionTabCtrlRegionAdapter
    (IRegionBehaviorFactory regionBehaviorFactory)
    : base(regionBehaviorFactory)
    {
    }
}

```

The inherited **RegionAdapterBase** class has two abstract methods: **CreateRegion** and **Adapt**.

The RegionAdapterBase CreateRegion Method:

The **CreateRegion** method determines the type of object to return. It returns one of three region types:

- Region
- SingleActiveRegion
- AllActiveRegion

The Region Return Type

The **Region** return type is used with controls that derive from the **Selector** class. Tab controls and dock managers are some of the controls that use this return type.

The SingleActiveRegion Return Type

The **SingleActiveRegion** return type is used with controls that derive from the content control class. Content controls are one of the controls that use this return type.

The AllActiveRegion Return Type

The **AllActiveRegion** return type is used with controls that derive from the **Items** control class. **ListBox** and **ComboBox** are some of the controls that use this return type.

The RegionAdapterBase Adapt Method

The **Adapt** method is where Prism 4 adds and removes views to and from Prism 4 regions.

Taking a look at the code in Listing 30, we see that two arguments are passed to the method: an **IRegion** object (**region**) and a **TabControlExt** object (**regionTarget**).

The **region.Views.CollectionChanged** event is triggered each time that a view is added, removed, or otherwise changed in the region's Views collection.

An iterator is used to loop through each tab in the Views collection. If the tab is not already in the collection, it is added.

In this case it is only necessary to add tabs. If required, code could be added to trigger when tabs are removed.

Adding the RegionAdapter to the Bootstrapper

Ok, we have a **RegionAdapter**. How do we let Prism 4 know about it? That's the next step in the process.

The ConfigureRegionAdapterMappings Method

One of the methods in the **bootstrapper** class is the **ConfigureRegionAdapterMappings** method. This method is used to register region adapters with Prism 4. Listing 32 is an example of the **ConfigureRegionAdapterMappings** method that is used by the *Virtual Calculator* solution.

Listing 32: The Bootstrapper ConfigureRegionAdapterMappings Method

```
protected override RegionAdapterMappings ConfigureRegionAdapterMappings()
{
    RegionAdapterMappings mappings = base.ConfigureRegionAdapterMappings();

    if (mappings != null)
    {
        mappings.RegisterMapping(typeof(TabControlExt),
            this.Container.Resolve<SyncfusionTabCtrlRegionAdapter>());
    }

    return mappings;
}
```

The first step is to create the **RegionAdapterMappings** object from the base class's mappings object.

A guard clause is used to ensure that the mappings object was created correctly. If the mappings object is not null, the **TabControlExt** and its associated **RegionAdapter** is registered in the mappings object.

As many **RegionAdapters** as needed can be registered in this manner.

The bottom line is that **RegionAdapter** classes show Prism 4 how to work with controls that do not work with the three region adapters that are included with Prism 4 out of the box. This makes it possible to extend the types of controls that can be used with the Prism 4 framework as views.

Summary

So, what have we learned so far? Prism 4 regions allow for the dynamic building of user interfaces at runtime. Regions are placed in views using the **RegionManager.RegionName** attached property. The Microsoft control library already has region adapters that work with its container controls. Third-party controls need custom **RegionAdapter** classes in order to be used as regions in Prism 4 solutions. Many third-party control companies furnish the needed **RegionAdapter** code for their controls.

Now that we have looked at Prism 4 regions in detail, let's take a look at Prism 4 modules.

Chapter 9 Prism 4 Modules

Think of a Prism 4 module as the workhorse of any Prism 4 solution. Modules are where the bulk of the work is done in your Prism 4 solutions.

What is a Prism 4 Module?

A Prism 4 module is a Visual Studio project that contains a class that implements the **IModule** interface. In more general terms, a module is a block of program functionality that can encompass UI, business rules, data, and coordination functionality between the UI and business entities.

Prism 4 modules are designed to be atomic, meaning that modules should be independent entities that have very little or no dependences on other modules. This helps to enforce loose coupling and separation of concerns.

The **IModule** interface consists of a single method named **Initialize**. The **Initialize** method is used to configure the module's classes when the module is first loaded.

Listing 33: The MATH_MODULE Module Class

```
[Module(
    ModuleName = "MATH_MODULE",
    OnDemand = false)]
public class MATH_MODULE : IModule
{
    private IRegionManager RegionManager { get; set; }
    private AddTwoView AddTwoView;
    private SubtractView SubtractView;

    //Constructor:
    public MATH_MODULE(
        IRegionManager RegionManager,
        AddTwoView AddTwoView,
        SubtractView SubtractView)
    {
        if (RegionManager != null)
        {
            this.RegionManager = RegionManager;
        }

        if (AddTwoView != null)
        {
            this.AddTwoView = AddTwoView;
        }

        if (SubtractView != null)
        {
```

```

        this.SubtractView = SubtractView;
    }
}

//Add the Module Views to the Regions here:
public void Initialize()
{
    ///Add the user controls to the region here:
    IRegion MathRegion = RegionManager.Regions["MathRegion"];

    MathRegion.Add(this.AddTwoView, "AddTwoView");
    MathRegion.Add(SubtractView, "SubtractView");
}
}

```

The Module Class Data Annotation

The data annotation that precedes the class definition in Listing 33 can be used to set properties for the class. In this case the class name is added and the **OnDemand** property is set to **false**.

The **OnDemand** property determines if the module is loaded at solution startup or if it should be loaded later with program interaction.



Note: The *OnDemand* property does not work with this solution because we are using the configuration file to load the modules. If you use the directory scan method to load modules, it works. Since in most of my Prism 4 solutions I use directory scan to load modules, I left the code in to show how to load modules at startup or on demand if this method of module loading is used. This data annotation can be removed or left as is with no adverse ramifications to the code.

Configuration files use the *startupLoaded* property to determine when a module loads. We'll look at the configuration file in Chapter 13.

Using Constructor Injection in the Module to Instantiate Objects

The **Math** module class in Listing 33 has three class-level members: **RegionManager**, **AddTwoView** and **SubtractView**. These members are instantiated by passing three arguments to the class constructor: **RegionManager**, **AddTwoView** and **SubtractView**. Guard clauses are used for each of the passed arguments in the constructor, and if the objects are not null, each is assigned to its associated local class member.

Creating the MathRegion

You'll recall that we created a **MathRegion** in both of the *Main* project's shell forms. In the Listing 33 **Initialize** method, the **IRegion** interface is used to instantiate a region object called **MathRegion**. This region is used as an injection mechanism for views.

Injecting Views into Regions

When we spoke about Prism 4 regions earlier, we didn't address the issue of how views are injected into regions. There are two methods that can be used to inject views: view discovery and view injection.

View Discovery

View discovery is used to automatically load views when the solution starts. The view or views are registered with the region manager's **RegisterViewWithRegion** method. This method takes two arguments: a string that corresponds to the *name* of the region, and the *type* of the view that will be loaded into the region.

View discovery is used in situations where it is not necessary to change the view once it is loaded.



Note: Technically view discovery could have been used to load both of the shell form logo views. Neither of these views change once loaded. I tend to use view injection for all of my views. This makes things easier if it becomes necessary to change the view as the solution changes. Take a look at the Event Aggregation and View Based Navigation solution to see examples of view discovery in a solution.

Listing 34: An Example of Prism 4 View Discovery

```
regionManager.RegisterViewWithRegion("MathRegion", typeof(AddTwoView));
```

View Injection

View injection programmatically adds views to regions either automatically, through program control, or by user interaction. Using view injection increases the amount of control that is available when creating and injecting views. This method of view creation is used in the *Virtual Calculator* solution.

Injecting Math Module Views

The two views **AddTwoView** and **SubtractView** are added to the math region in the **Initialize** method of Listing 35.

Listing 35: Prism 4 View Injection

```
Public void Initialize()
{
    ...
    MathRegion.Add(this.AddTwoView, "AddTwoView");
    MathRegion.Add(SubtractView, "SubtractView");
}
```

This code creates the object and also activates it in the region.

Notice that all that is needed is to add the two views to the correct region. Because the region adapter class uses a region return type, the region adapter takes care of adding the two tabs correctly to the Syncfusion tab control. We don't have any indication that the views are tab items.

Also note that when we use view discovery and view injection that we are actually using Prism 4 navigation to load the views. We'll go into more detail about this when we look at navigation in Chapter 12.

Loading Modules

In order to use modules in Prism 4 solutions, they must be loaded. Module loading takes place when the solution starts in most cases. There are four methods that can be used to load Prism 4 modules:

1. With Code
 - With a Configuration File
 - With a XAML file
 - With Directory Scan

Loading Modules with Code

Loading a module in code is the fastest and least complicated solution to module-loading with Prism 4. The **AddModule** method of the **ModuleCatalog** is used to add modules to the module catalog's collection.

This method of loading modules is easiest, but also has drawbacks. Because it's necessary to reference each of the solution's modules in the *Main* project, this type of loading introduces tight coupling in the solution. This of course defeats the purpose of using modules in the first place. Because of this tight coupling, I advise that one of the other three loading strategies be used.

Listing 36: Loading Prism 4 Modules With Code

```
protected override void ConfigureModuleCatalog()
{
    Type MathModule = typeof(MATH_MODULE);
    ModuleCatalog.AddModule
    (
        new ModuleInfo()
        {
            ModuleName = MathModule.Name,
            ModuleType = MathModule.AssemblyQualifiedName,
        }
    );
}
```

Loading Modules with a Configuration File

Using a configuration file has a number of advantages and drawbacks. First, configuration files allow for late binding. This means that if changes are needed to the module-loading strategy, they can be introduced without recompiling the solution. Also, all loading strategies are in a known location, making them easy to find.

Listing 37 shows an example of the bootstrapper's **CreateModuleCatalog** method.

Listing 37: Loading Prism 4 Modules With a Configuration File

```
protected override IModuleCatalog CreateModuleCatalog()
{
    ConfigurationModuleCatalog configurationCatalog = new ConfigurationModuleCatalog();
    return configurationCatalog;
}
```

The main drawback to adding loading strategies to the configuration file is that ensuring the markup is correct can become difficult with complex strategies. It also becomes progressively more difficult to ensure that the markup remains valid as the number of modules increases.

The *Virtual Calculator* solution uses this loading strategy. We will take a detailed look at the app.config file markup in Chapter 13.

Loading Modules with an XAML File

XAML files have many of the same good points and drawbacks as using a configuration file. The main point is that if large numbers of modules are used in the solution, the markup can become difficult to validate and maintain.

Listing 38 shows an example of the markup needed to load a module in an XAML file. XAML files can be added to, and accessed from, the same location as the main project's configuration file.

Listing 38: Loading Prism 4 Modules With a XAML File

```
<Modularity:ModuleInfo Ref="PRISM4.MATH_MODULE.xap" ModuleName="MathModule"
ModuleType="PRISM4.MATH_MODULE, MATH_MODULE, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null" />
```

Loading Prism 4 Modules with Directory Scan

Directory scan is my preferred method of loading Prism 4 modules. Once the solution is set up for this kind of module loading, it's fairly simple to use the system. It's also a very decoupled strategy for loading modules.

I'm not going to go into detail about loading strategies in this chapter, but will talk about module loading with configuration files in Chapter 13.

Listing 39: Loading Prism 4 Modules With a Directory Scan

```
protected override IModuleCatalog CreateModuleCatalog()
{
    return new DirectoryModuleCatalog() {ModulePath = @".\Modules"};
```

```
}
```

The @".Modules" module path in Listing 39 defines the directory in which the module DLL files will be located when the module projects are compiled. In this case the module DLL files should be located in a directory named *Modules* in the bin directory of the *Main* project.



Note: I also use a method of automatically moving the updated DLL files to this directory each time that the solution is built. I won't go into details here because this type of module loading is not used in the demo solutions.

Module Dependencies

Prism 4 module dependencies are probably not what you think. When we think of dependencies between modules, setting references comes to mind. When we speak of Prism 4 module dependencies, we are talking about the order in which modules are loaded.

If module **A** depends on module **C**, then module **C** must load before module **A**. Prism 4 exposes mechanisms that allow for module-dependency configuration.

Listing 39: Setting a Prism 4 Module Dependency

```
[Module(  
    ModuleName = "MATH_MODULE",  
    OnDemand = false),  
    ModuleDependency("PRISM4.COMPANY_LOGO_MODULE")]  
public class MATH_MODULE : IModule  
{  
    ...  
}
```

This code will load the **COMPANY_LOGO_MODULE** module before the **MATH_MODULE** module.

How MVVM Integrates with Prism 4 Module Projects

Prism 4 modules are where most of the MVVM design patterns will reside in our solutions. I use a standard folder structure in my modules.

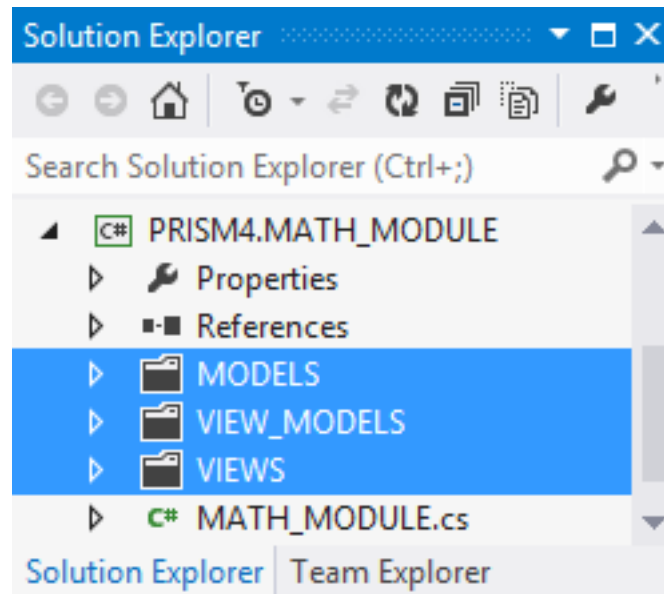


Figure 11: The Math Module MVVM Folder Layout

Summary

In this chapter we learned that Prism 4 modules are Visual Studio projects that contain a single class that implements the **IModule** interface. We learned that this interface has a single method, **Initialize**. We saw how the method can be used to add views to regions.

We also looked at different methods of loading Prism 4 modules, and talked about how dependencies can be resolved between modules.

In the next chapter we're going to look at communications in Prism 4 solutions.

Chapter 10 Prism 4 Commands

Earlier in this book we looked at how to update the user interface by setting states from the view model of the Math module. In this chapter, we'll look at how users of the UI can trigger code in the View Model by using Prism 4 commands.

What are Prism 4 Commands?

Prism 4 commands are based on the GoF command design pattern. They are designed to allow users of user interfaces to trigger actions in code. In the case of MVVM, controls in the view (buttons, combo boxes, etc.) trigger commands in the view model that carry out needed functionality in the solution.

Commands are communication mechanisms that can be used to communicate between views and view models or between Prism 4 modules. If the objective is to immediately react to an interface interaction, commands are a good choice.

WPF provides a routed command that is dependent on the visual tree when servicing commands. This type of command does not work well with Prism 4 solutions, because in these solutions, the view is separate from the entity that actually processes the UI interaction request.

Prism 4 commands, on the other hand, use a delegate command to overcome this issue. Delegate commands call a delegate method in response to a triggered action in the UI. This allows for interaction between the UI and view model, without depending on the view's visual tree. This helps to maintain loose coupling between the view and view model while allowing communications between the two entities.

Prism 4 commands consist of three required parts and one optional part:

1. Private Member and Public Property (Required)
Code that instantiates the command (Required)
Associated On method. (Required)
Associated Can method (Optional)

The Command Property

The Prism 4 command property is a public property that implements the **ICommand** interface. The command property is used to expose the view model-based command to the view through data binding. The command can then be triggered from the view by an interaction trigger or behavior. Listing 40 shows an example of a Prism 4 command property.

Listing 40: A Prism 4 Delegate Command

```
private ICommand CALCULATE_BUTTON_CLICK_COMMAND;  
public ICommand CalculateButtonClickCommand  
{  
    get  
    {
```

```

        return CALCULATE_BUTTON_CLICK_COMMAND;
    }

    set
    {
        if (CALCULATE_BUTTON_CLICK_COMMAND != value)
        {
            CALCULATE_BUTTON_CLICK_COMMAND = value;
            NotifyPropertyChanged("CalculateButtonClickCommand");
        }
    }
}

```

The only code in Listing 40 that you may not be familiar with is the **NotifyPropertyChanged** code in the property setter. If you recall from Chapter 7, this call is required to ensure that the user interface is updated when changes are made to the property. It relies on the **INotifyPropertyChanged** interface.

The Prism 4 Code that Instantiates the Command

The view model's class constructor is used to instantiate the command. This is necessary not only because the generic command class can take a payload but also because one or two associated methods are used with the delegate command type. Listing 41 shows an example of this code.

Listing 41: Instantiation of a Prism 4 Delegate Command

```

CALCULATE_BUTTON_CLICK_COMMAND = new DelegateCommand(OnCalculateButtonClick);

```

The new delegate command is applied to the associated command's private member. Note that the **OnCalculateButtonClick** method name is passed as an argument to the **DelegateCommand** constructor.

Prism 4 Command On Method

The **On** method is the execute method that is associated with the command. This is the method (or delegate) that actually carries out the functionality when the command is triggered. As we saw in the previous section, the **On** method name is the first argument that is passed to the delegate command's constructor when it is instantiated. Listing 42 shows an example of the **On** method.

Listing 42: A Prism 4 Delegate Command On Method

```

private void OnCalculateButtonClick()
{
    TextBoxTotalText = SubtractIntegers.SubtractIntegers(IntegerStrings);

    if (IntegerStrings != null && TextBoxTotalText != null)
    {
        FormulaTextboxText = SubtractIntegers.GetFormula(IntegerStrings,
        TextBoxTotalText);
    }
}

```

The code in Listing 42 is actuated when the **Calculate** button is clicked. We'll look at the XAML needed to trigger this command later in this chapter.

Prism 4 Command Can Method

The **Can** method is an optional method that is used to set the enabled state of the associated command. This is the second (optional - overloaded) argument passed to the constructor of the **DelegateCommand**. Our solution does not use this method, but it is a useful method when you want to add code to the command to change its enabled state. Listing 43 shows an example of what this method would look like.

Listing 43: A Prism 4 Delegate Command Can Method

```
private bool CanCalculateButtonClick()
{
    return true;
}
```

You can of course add code to the method to determine if the returned **Boolean** value is **true** or **false**. This method is not used in our view model because the UI uses state based navigation to set the different states.

It is also necessary to change the code in the constructor for the command if this method is used. Listing 44 shows an example.

Listing 44: A Prism 4 Delegate Command Instantiation with Can method

```
CALCULATE_BUTTON_CLICK_COMMAND = new DelegateCommand(OnCalculateButtonClick,
CanCalculateButtonClick);
```

Listing 44 simply overloads the delegate command constructor with both the **On** and **Can** method names.



Note: I use *On* and *Can* prefixes to describe these methods because this is the accepted method of naming these methods. It is not strictly necessary to use these conventions when naming command methods.

All of the code that we have used for commands so far resides in the view model.

Triggering Commands from the View

Now that we know how to set up delegate commands in the view model, how do we trigger these commands from the view?

The *Virtual Calculator* solution triggers commands by using WPF interaction triggers. Interaction triggers identify the event for a particular control that will serve as the trigger and then uses the **InvokeCommandAction** tag to identify the command to invoke. Listing 45 shows an example of the **CalculateButton** XAML markup with an interaction trigger.

Listing 45: The Calculate Button XAML Markup

```
<Button
    Name="CalculateButton"
    Content="Calculate"
    Grid.Row="0"
    Grid.Column="2"
    Grid.RowSpan="4"
    FontWeight="Bold"
    Foreground="DarkBlue"
    FontSize="16"
    Margin="5,0"
    Padding="10"
    IsDefault="True">

    <i:Interaction.Triggers>

        <i:EventTrigger
            EventName="Click">

            <i:InvokeCommandAction
                Command="{Binding CalculateButtonClickCommand}">
            </i:InvokeCommandAction>

        </i:EventTrigger>

    </i:Interaction.Triggers>
</Button>
```

First, an **Interaction.Triggers** tag is added to the **CalculateButton** control. This allows for the addition of multiple **EventTriggers** if necessary. Next, an event trigger is added and its **EventName** is set to **Click**. This name corresponds to the button's **Click** event. An **InvokeCommandAction** is added to the **EventTrigger**. The command property of this tag is bound to the **CalculateButtonClickCommand** view model property.

When the **Calculate** button is clicked, the **CalculateButtonClickCommand** is triggered. This action executes the code in the **OnCalculateButtonClick** method.

Triggering a Command with Parameters

So far we've looked at how to set up commands that operate without parameters. The delegate command class can be used as a generic class that takes a payload from the view. This allows for the passing of properties or entities that correspond to the current state of the associated control or the current state other controls in the view.

Passing parameters to the view model can be used when determining the item selected in a combo box or list box. It can also be used to pass objects to the view model when needed.

The *Virtual Calculator* solution uses this technique to pass the textbox text property to the view model when the text changed event in either one of the two textboxes is triggered. Passing parameters to the view model means that modifications need to be made to the command object, enabling it to take the parameter. Listing 46 shows an example of the **Integer1TextBox** command property.

Listing 46: The Integer1TextBox Command Property

```
private ICommand TEXT_BOX_ONE_TEXT_CHANGED_COMMAND;
public ICommand TextBoxOneTextChangedCommand
{
    get
    {
        return TEXT_BOX_ONE_TEXT_CHANGED_COMMAND;
    }

    set
    {
        if (TEXT_BOX_ONE_TEXT_CHANGED_COMMAND != value)
        {
            TEXT_BOX_ONE_TEXT_CHANGED_COMMAND = value;
            NotifyPropertyChanged("TextBoxOneTextChangedCommand");
        }
    }
}
```

The property for the **TextBoxOneTextChangedCommand** is no different from any other Prism 4 command; we'll start to see changes when we look at the command instantiation code in the constructor. Listing 47 shows an example of this code.

Listing 47: A Prism 4 Delegate Command Instantiation with Parameter

```
TEXT_BOX_ONE_TEXT_CHANGED_COMMAND = new
DelegateCommand<string>(OnTextBoxOneTextChanged);
```

In Listing 47, we are now passing a payload to the generic class of type **string**. This tells the class to expect a string value each time that the command is triggered.

The next change we see is in the **OnTextBoxOneTextChanged** method. Listing 48 shows an example of this code.

Listing 48: The OnTextBoxOneTextChanged Method

```
private void OnTextBoxOneTextChanged(string Textbox1Text)
{
    if (ClearFlag == false)
    {
        TextBoxOneText = Textbox1Text;
    }

    ResetUserInterface();

    IntegerStrings.Clear();
    IntegerStrings.Add(TextBoxOneText);
    IntegerStrings.Add(TextBoxTwoText);
}
```

```

IntegerTotal = TextBoxTotalText;

CurrentUIState = UIModel.TextBox1HasValueState(IntegerStrings, IntegerTotal);

StatusTextboxText = CurrentUIState.StatusText;
FormulaTextboxText = CurrentUIState.FormulaText;

TextBox1Exception = CurrentUIState.TextBox1ExceptionState;
TextBox2Exception = CurrentUIState.TextBox2ExceptionState;
TextBox1HasValue = CurrentUIState.TextBox1HasValueState;
TextBox1AndTextbox2HasValue = CurrentUIState.TextBox1AndTextbox2HasValueState;
BothTextBoxesBlankOrClear = CurrentUIState.BothTextBoxesBlankOrClearState;
}

```

The main change at this point is that the method is passed as an argument (**Textbox1Text**) with a type of **string**. Each time that a character is entered into the textbox, the text that is currently in the textbox is passed from the view to the view model.

We'll next look at the XAML trigger code to see how parameters are passed. Listing 49 shows an example of the **Integer1Textbox** markup.

Listing 49: The Integer1Textbox XAML Markup

```

<TextBox
    Name="Integer1TextBox"
    Grid.Row="0"
    Grid.Column="1"
    FontWeight="Bold"
    FontSize="14"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"
    VerticalContentAlignment="Center"
    Margin="0,0,0,3"
    Text="{Binding Path=TextBoxOneText}">

    <i:Interaction.Triggers>

        <i:EventTrigger
            EventName="TextChanged">

            <i:InvokeCommandAction
                Command="{Binding TextBoxOneTextChangedCommand}"
                CommandParameter="{Binding ElementName=Integer1TextBox, Path=Text}">
            </i:InvokeCommandAction>

        </i:EventTrigger>

    </i:Interaction.Triggers>

</TextBox>

```

The trigger markup in Listing 49 is almost identical to the markup in Listing 45. The only exceptions are that the **EventName** is set to the **TextChanged** event, the **Command** property is set to the **TextBoxOneTextChangedCommand** property, and that an extra property is added to the **InvokeCommandAction** tag. The **CommandParameter** property is used to pass a payload to the command object. This parameter is then passed as an argument to the command object's **On/Can** methods.

In this case, element binding is used to pass the text value of the **Integer1Textbox** to the view model command property.

We could have just as easily passed the textbox object by removing the path property and passing the **Integer1Textbox** element as the parameter. Then we'd just need to change the type that the delegate object and the **On** method argument expects for the textbox. We would also need to change the command class payload to a textbox control.

Global Commands

Prism 4 global commands are command classes that are visible to all of the other projects in the solution. These classes are one of the methods that is used for intra-module communications in Prism 4 solutions. The classes can be used in the same manner as the property-based command classes that we spoke about earlier, with the added value of being visible between Prism 4 modules.

Prism 4 Composite Commands

Prism 4 composite commands are used to allow a single, parent command to trigger multiple child commands. Composite commands work well in situations where it is necessary to call the same type of command across different modules. A good example is where it is necessary to implement "save all" functionally across a number of different modules. A single composite command can be used to trigger save commands in each of the associated modules.

I'm not going to talk in detail about global and composite commands because they are not used in the *Virtual Calculator* solution. But keep in mind that these two command types are also included in the Prism 4 framework.

Summary

In this chapter we looked at Prism 4 commanding. We learned about the different parts of a command and how to use them.

We also looked at WPF triggers. We learned how to integrate triggers with commands. We also worked with both types of triggers: triggers without parameters and triggers with parameters.

In the next chapter we'll continue to look at Prism 4 communications. This time we'll work with event aggregation.

Chapter 11 Prism 4 Event Aggregation

Prism 4 event aggregation is another Prism 4 communications mechanism. Rather than being built on the GoF command design pattern, the event aggregator uses the *observer design pattern*. The observer design pattern is a publish/subscribe pattern that enables communication between multiple publishers and subscribers.



Note: The GoF definition of the observer design pattern is a one-to-many construct that has a single publisher to many subscribers. The Prism 4 event aggregator uses a many-to-many construct that allows for as many subscribers and publishers as needed.

Event aggregation is an excellent method of communication between Prism 4 modules, and its versatile interface makes it suitable for a number of different situations.

In the next two chapters we'll work with the *Event Aggregation and View-Based Navigation* (EA_VBN) solution to explain event aggregation and view-based navigation. These two Prism 4 constructs are excellent examples of using event aggregation to access views in different modules while keeping code loosely coupled.

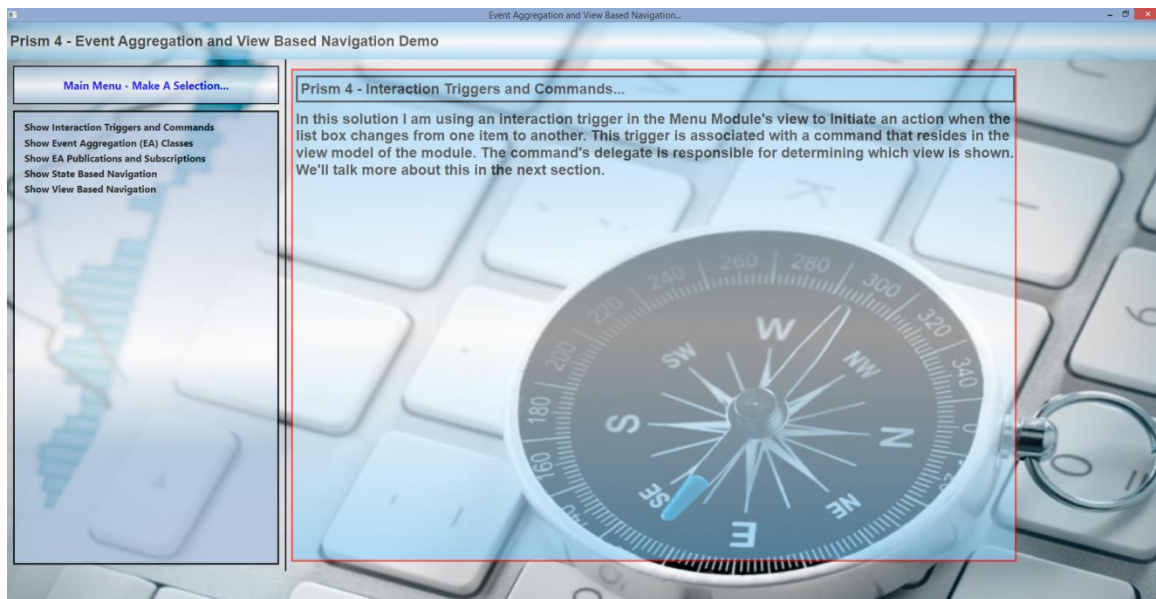


Figure 12: The Prism 4 *Event Aggregator and View-Based Navigation* Demo Solution

The solution shown in Figure 12 uses event aggregation to navigate between five different views in four modules. A Listbox in the menu module is used to select a menu item. The string value of the selected menu item is passed as a parameter to the view model via an interaction trigger in the view's XAML markup. This value is used to determine which view is shown.

An event aggregator is made up of:

- The event aggregation class
- Subscribers
- Publishers
- Trigger mechanisms

The Event Aggregation Class

The event aggregation class is used to identify and create Prism 4 event aggregation objects. As a rule I place these classes in the infrastructure project of the solution so that the other classes have access the object. Figure 13 shows an example of the EA_VBN solution's structure.

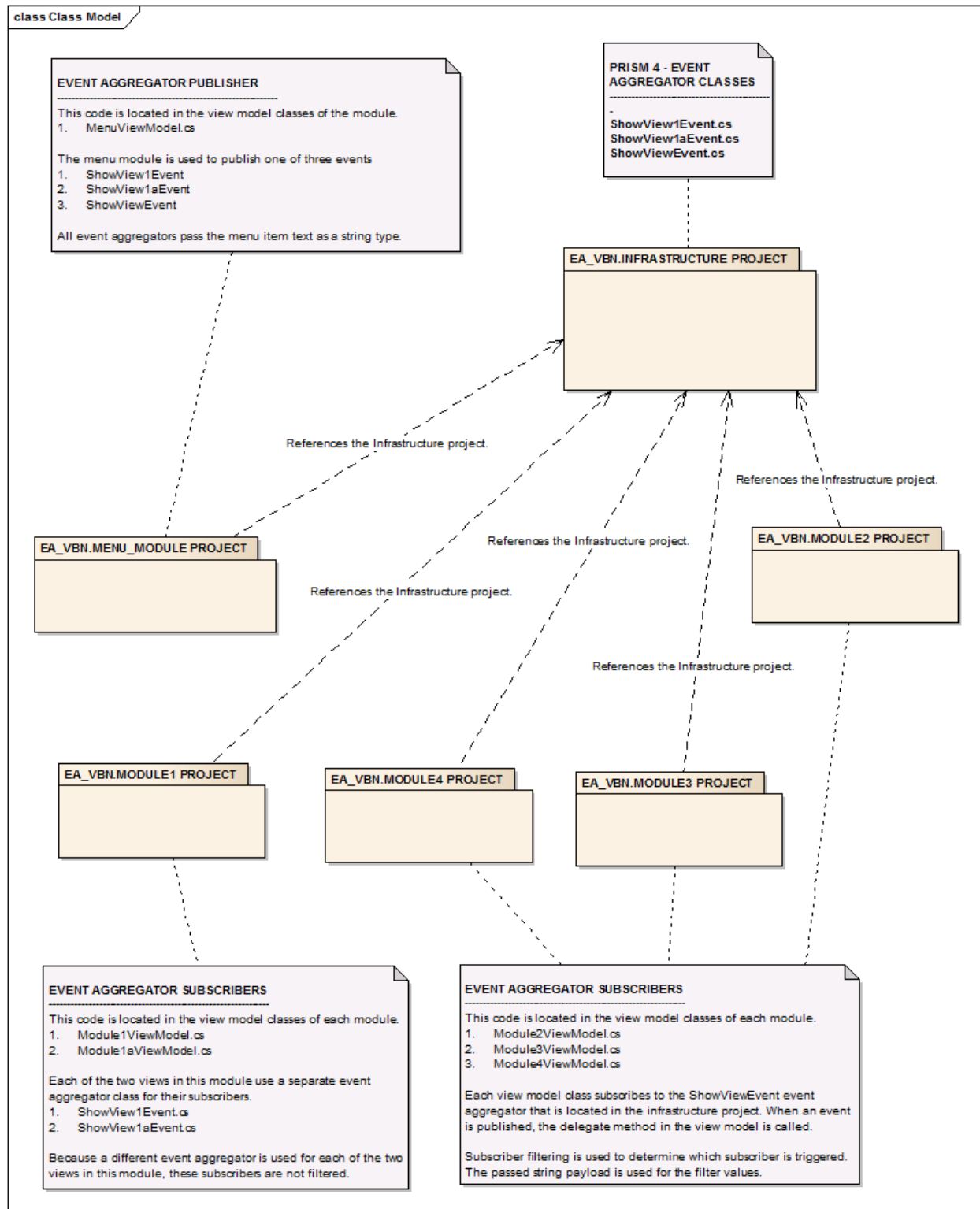


Figure 13: The EA_VBN Event Aggregation Structure.

Listing 50 is an example of the **ShowViewEvent** event aggregation class. I usually place an Event Aggregation folder in the *Infrastructure* project for these classes.

Listing 50: A Prism 4 Event Aggregation Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Microsoft.Practices.Prism.Events;

namespace EA_VBN.INFRASTRUCTURE.EVENT_AGGREGATORS
{
    public class ShowViewEvent : CompositePresentationEvent<string>
    {
    }
}
```

These classes inherit from the **CompositePresentationEvent** generic class, which takes a payload of the type that is passed to the class. In this case a string is passed. The string's value is passed from the publisher to the subscriber.

Event Aggregation Subscribers

Once the event aggregation class is created, subscribers can subscribe to the class to monitor for events being raised. Figure 13 shows five subscribers that are associated with three event aggregation classes.

Subscriber Overloads

The Prism 4 event aggregator subscribe method has four overload types:

1. The default subscription
- The UI thread subscription
- Subscription filtering
- Strong reference subscription

The Default Subscription

Notice that in Listing 51, I pass the event aggregator to the view model's constructor using constructor injection. This ensures that the event aggregator is instantiated by the unity Dependency Injection Container (DIC). A guard clause is used to confirm that the event aggregator is not null. The passed event aggregator is then assigned to a private local member, **EventAggregator**.



Note: *It's not necessary to register the event aggregator with the Unity DIC because Prism 4 automatically creates an event aggregator when the bootstrapper is started.*

The default subscription first uses the **GetEvent** method to get the event aggregator that is to be used. This class assigns the event aggregator to a callback method that is passed to the **Subscribe** method. This method is where the code for the event aggregator is executed. As a rule, I add this code to the constructor of the view model classes. Listing 51 shows an example of the view model's constructor and the event aggregator-based **GetEvent** method.

Listing 51: The Prism 4 View Model Constructor and Event Aggregation GetEvent Class

```
private IEventAggregator EventAggregator;

public Module1ViewModel(IEventAggregator eventAggregator)
{
    if(eventAggregator != null)
    {
        EventAggregator = eventAggregator;
    }

    EventAggregator.GetEvent<ShowView1Event>().Subscribe(ShowViewOne);

    ...
}
```

The payload in Listing 51, **ShowView1Event**, is the event aggregator that this subscriber will use. This is the event aggregator that was added to the infrastructure project. The callback method is the **ShowViewOne** argument that is passed to the **Subscribe** method. The **ShowViewOne** method is added to the **ViewModel** as a stand-alone, private method in most cases. Listing 52 shows an example of the **ShowViewOne** method.

Listing 52: A Prism 4 Event Aggregation ShowViewOne Method

```
private void ShowViewOne (string View)
{
    this.RegionManager.RequestNavigate("ViewsRegion", new
    Uri("Module1View", UriKind.Relative));
}
```

Don't be concerned with the **RequestNavigate** code in the **ShowViewOne** method; we will talk about that in the next chapter.

Note that the string type payload that was passed to the event aggregator class in Listing 50 is resolved as the view argument in the **ShowViewOne** method.

The UI Thread Subscription

The subscriber method UI thread overload allows the publisher of the event to update the user interface. As a default, the publisher thread is used when an event is published. If this thread is a background thread, the UI won't be updated. Using the subscriber method's UI thread overload addresses this issue. Listing 53 shows an example that uses this overload.

Listing 53: The Subscribe UI Thread Overload

```
this.eventAggregator.GetEvent<ShowView1Event>().Subscribe( ShowViewOne ,  
ThreadOption.UIThread);
```

The **ThreadOption** can use one of three selections:

1. PublisherThread
BackgroundThread
UIThread

PublisherThread

This default setting allows the event to be received on the thread of the publisher.

BackgroundThread

This type of thread is an asynchronous thread that works with .NET framework thread pool threads.

UIThread

The UI thread allows for the update of the UI by receiving a thread that is capable of UI updates.

Subscription Filtering

The subscription filtering overload allows the subscriber to determine if its associated callback method is executed when an event is published. You can use filtering for single or multiple subscribers.

The EA_VBN solution uses subscription filtering. A single event aggregator triggers selected single and subscriber delegates. The filter data is passed as the menu item text string value from the menu list box in the menu module project. The string payload that we designated when we created the event aggregator is how that value is passed. Modules 2 through 4 use subscriber filtering to ensure that only one subscriber is triggered per menu selection.

Listing 54 shows an example of subscription filtering.

Listing 54: The Subscribe Filter Overload

```
EventAggregator.GetEvent<ShowViewEvent>().Subscribe(ShowViewTwo,ThreadOptio  
n.UIThread,false,view => view == "Show EA Publications and Subscriptions");
```

This filter will only show the view if the **ShowViewOne** argument is equal to **Show EA Publications and Subscriptions**. The false setting after the **ThreadOption** is the **KeepSubscriberReferenceAlive** argument. We'll talk more about this setting in the next section.

Strong Delegate Reference Subscription

Sometimes it's necessary to publish a number of events in a short period of time. This can cause performance issues in your solution. To address this issue, you can use strong delegate references.

As a rule, weak delegate references are used by default. This kind of reference allows the garbage collector to remove objects without any action on your part. The issue with this kind of cleanup is that it may take a good deal of time before objects are destroyed.

On the other hand, strong delegate references stop the automatic destruction of objects. This keeps the resources from being destroyed. When strong delegate references are used, it's important that the code manually unsubscribe from the event.

Listing 55 shows an example of using strong delegate references with event aggregator subscriptions.

Listing 55: The Subscribe Strong Delegate Reference Overload

```
bool KeepSubscriberReferenceAlive = true;

this.eventAggregator.GetEvent<ShowViewEvent>().Subscribe( ShowViewTwo ,
ThreadOption.UIThread, KeepSubscriberReferenceAlive , DisplayMessage =>
ShowMessage == "Show EA Publications and Subscriptions");
```

To stop the garbage collector from destroying the object, set the **KeepSubscriberReferenceAlive** member to **true**.

Event Aggregation Publishers

Our last step when working with event aggregation is to publish an event. Publishing starts in the same manner as subscribing to an event. **GetEvent** is called from the event aggregator, but in this case, the **Publish** method is called. The method takes the payload that was created when we added the event aggregation class to the infrastructure project. As a rule, this code is added to view model code in the method where the event is to be triggered. This code could be, for instance, included in a command method and triggered from the UI or a timer control. Listing 56 shows an example of an event aggregator publisher that is triggered from a Prism 4 command method.

Listing 56: The Event Aggregator Publish Method

```

private void GetMenuItem(string menuItem)
{
    switch (menuItem)
    {
        case "Show Interaction Triggers and Commands":
            EventAggregator.GetEvent<ShowView1Event>().Publish(menuItem);
            break;

        case "Show Event Aggregation (EA) Classes":
            EventAggregator.GetEvent<ShowView1aEvent>().Publish(menuItem);
            break;

        default:
            EventAggregator.GetEvent<ShowViewEvent>().Publish(menuItem);
            break;
    }
}

```

In this case, the event aggregator would be triggered from the UI with a command interaction trigger.

Listing 57: The Event Aggregator Interaction Trigger

```

...
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
...

<ListBox.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="LightSteelBlue" Offset="0.004"/>
        <GradientStop Color="LightSteelBlue" Offset="1"/>
        <GradientStop Color="White" Offset="0.537"/>
        <GradientStop Color="#FFA9C5DC" Offset="0.748"/>
    </LinearGradientBrush>
</ListBox.Background>

<i:Interaction.Triggers>

    <i:EventTrigger EventName="SelectionChanged">

        <i:InvokeCommandAction
            Command="{Binding GetMenuItemCommand}"
            CommandParameter=
                "{Binding ElementName=ViewNames,
                Path=SelectedValue}">
        </i:InvokeCommandAction>
    </i:EventTrigger>
</i:Interaction.Triggers>

```



```
        </i:EventTrigger>

        </i:Interaction.Triggers>

    </ListBox>

    ...
```

Listing 57 shows the interaction trigger that is used to actuate the **GetMenuItemCommand** command in the module's view model. It is first necessary to set a namespace to the Microsoft Expression interactivity code library. The first line of XAML creates this namespace with a name of **i**.

Next we create an interaction trigger in the **ListBox** control and nest an event trigger into the interaction trigger. The event trigger is set to the **SelectionChanged** event of the parent **ListBox** control.

We next add an **InvokeCommandAction** tag. This tag is used to access the command property in the view model. The first tag, **Command**, is the name of the command property that will be executed—in this case, **GetMenuItemCommand**. In many cases, this is all that you need to execute a command. We also add an optional command parameter to the **InvokeCommandAction**. This is how we pass object or property values to the command object. In this case, we pass the **SelectedValue** property of a **ListBox** control to the command using element binding.

When a menu item is selected the command is invoked, which in turn publishes to the event aggregator. The view that we want to display is passed as a menu value parameter to the command object, and then to the event aggregator.

Summary

Event aggregation is one of a number of methods that can be used to communicate between loosely coupled components in Prism 4 solutions. In this chapter we talked about how to create event aggregators, how to use the **GetEvent** method to identify and use the event class, and how to subscribe to and publish events. We also took a close look at the **Subscribe** method and its overloads.

In the next chapter we'll look at Prism 4 navigation.

Chapter 12 Prism 4 Navigation

Before we talk about Prism 4 navigation, let's talk about state. *State* determines how the user interface changes as the user interacts with the solution over time. State also can determine how the UI reacts to internal (programmatic) changes. The state is a status that the solution is in, at a particular point in its lifetime.

The best way to explain this is with an example. When you first start most solutions that have a menu control, you'll find that the FILE-New and the FILE-Open menu items are enabled. You may also find that the FILE-Save and FILE-Print menu items are disabled. If you think about it, this makes sense. In the case of Microsoft Word, you can't save or print a document until a document is either created or loaded. So when the solution is in a just-started state, these options are not available. After you load a document, the state or status of the solution changes (to perhaps a document-loaded state) and the solution's menu may reflect those changes. So again, state determines how the user interface changes as the user or internal code interacts with the solution over its lifetime.

Navigation in Prism 4 is a means of reacting to state changes over the lifetime of a solution, and covers a large number of situations. The following is a list of some of these situations:

- Updates to user control properties depending on the current state of the solution over time.
- Updates to the layout of user interfaces depending on the user's interactions with the solution.
- Changes from one UI to another through user interaction or code.
- The addition or removal of user controls/views through user interaction or code.
- The update of displayed data through user interaction or code.

Navigation in Prism 4 solutions is not as straightforward as in WPF solutions because of the necessity of integrating navigation with MVVM. Prism 4 supplies the tools needed to reduce the complexity of navigating in loosely coupled solutions.

Prism 4 uses two kinds of navigation: view-based navigation, and state-based navigation.

View-Based Navigation

View-based navigation is navigation that switches from one view to another, or that adds or removes a view to or from the visual tree.

In the *Virtual Calculator* solution, a simple form of Prism 4 navigation is used to add the company logo and application logo views to their respective regions in the shell form. The *Math* module uses view-based navigation to add the two tabs (Add Two Integers and Subtract Two Integers) to the Syncfusion tab control. Listing 58 shows an example of loading the two *Math* module tabs.

Listing 58: Loading Math Views with Prism 4 Navigation

```
[Module(
```

```

ModuleName = "MATH_MODULE",
OnDemand = false)]
public class MATH_MODULE : IModule
{

    private IRegionManager RegionManager { get; set; }
    private AddTwoView AddTwoView;
    private SubtractView SubtractView;

    //Constructor:
    public MATH_MODULE(
        IRegionManager RegionManager,
        AddTwoView AddTwoView,
        SubtractView SubtractView)
    {
        if (RegionManager != null)
        {
            this.RegionManager = RegionManager;
        }

        if (AddTwoView != null)
        {
            this.AddTwoView = AddTwoView;
        }

        if (SubtractView != null)
        {
            this.SubtractView = SubtractView;
        }
    }

    //Add the Module Views to the Regions here:
    public void Initialize()
    {
        ///Add the user controls to the region here:
        IRegion MathRegion = RegionManager.Regions["MathRegion"];

        MathRegion.Add(this.AddTwoView, "AddTwoView");
        MathRegion.Add(SubtractView, "SubtractView");
    }
}

```

The views and region manager are first instantiated by using constructor injection. Guard clauses ensure that the objects are not null; if they are not null, they are assigned to private members in the class. The **Initialize** method creates a region (**MathRegion**) that will serve as the loading point for the two views. The two tabs (**AddTwoView** and **SubtractView**) are then added to the region.

This type of navigation is considered to be a limited form of Prism 4 navigation. The region class of Prism 4 has been extended to allow for more dynamic navigation between views. The previous code listing shows an example of loading the view automatically when the module is loaded. This type of view loading works fine when the module is being initialized, but how are views changed as the solution runs?

Using a Region Manager to Navigate Between Views

The *Event Aggregation and View-Based Navigation* (EA_VBN) solution uses view-based navigation to show different views in the **ViewsRegion** of the solution's shell form.

The **RequestNavigate** method of the **RegionManager** class allows for navigation during runtime of the solution. Listing 59 shows an example of the **RequestNavigate** method in the Module 1 view model class.

Listing 59: Using IRegionManager to Navigate

```
IRegionManager RegionManager = new RegionManager;  
  
...  
  
this.RegionManager.RequestNavigate("ViewsRegion", new Uri("Module1View",  
UriKind.Relative));  
  
...
```

After creating a region manager, the **RequestNavigate** method of the region manager is called. Two arguments are passed to the method. The first argument is the name of the region that will be populated. The second argument is a URI that contains the name of the view that will populate the region and a **UriKind** enumeration. Navigation code is usually called from the module's view model.

Using a Region to Navigate Between Views

A region can also be used to navigate to a view, as shown in Listing 60.

Listing 60: Using IRegion to Navigate

```
IRegion ViewsRegion = RegionManager.Regions["ViewsRegion"];  
  
...  
  
this.ViewsRegion.RequestNavigate (new Uri("Module1View", UriKind.Relative));  
  
...
```

It's not necessary to pass the region name to the **RequestNavigate** method when a region is used to navigate because the region is known.

It's also possible to identify a callback method when navigating with the **RequestNavigate** method. This callback method will execute after the navigation is complete. Listing 61 shows an example of the **RequestNavigate** method with a callback.

Listing 61: Using **IRegion** to Navigate with a Callback Method

```
IRegion ViewsRegion= RegionManager.Regions["ViewsRegion"];

ViewsRegion.RequestNavigate (new Uri("Module1View", UriKind.Relative),
NavigationCompleted);

private void NavigationCompleted (NavigationResult Result)
{
}
}
```

The callback method is added as the last argument of the **RequestNavigate** method. A method with the callback name is then created. This method must take a **NavigationResult** type as an argument. The **NavigationResult** is comprised of a number of properties that return the status of the navigation call.

State-Based Navigation

State-based navigation is navigation that is implemented through state changes in the solution's modules. The changes are made to the visual tree of the solution, through user interaction, or by program code. State-based navigation works best in the following scenarios:

- The view needs to change properties of its existing controls to reflect state changes as the user interacts with the user interface.
- The view needs to change properties of its existing controls to reflect state changes made by the solution's internal code.
- Existing data needs to be shown in different layouts (List View, Details View, etc.).

The View Changes State Through User Interaction

The *Virtual Calculator* solution uses a combination of XAML markup, view model, and model code in the *Math* module to update the user interface depending on the current state of the module. Behaviors in the views are used to trigger specific states, which are also located in the view as XAML markup. The states run animation storyboards that in turn set the properties of controls in the view. Each state has a unique set of properties that are set to specific values when the state is activated.

States are triggered from the view models of the module. As the user interacts with the UI, interaction triggers in the view fire when events are triggered. These triggers are associated with commands in the view model that determine the current state. The model exposes services from the infrastructure project that actually apply the necessary validation and UI update business rules to reflect the current state.

The View Changes State Through Internal Code

The view can also be updated by code that uses internal criteria to set states. Examples are timers, random number generators, date/time triggers, and any other code-based mechanism that does not depend on user interaction with the solution.

Existing Data Needs to Be Shown in Different Layouts

It is sometimes necessary to show data in multiple ways. For instance, it may be necessary to show data in a tab view, carousel view, and list view. As long as it's not necessary to update the data, state-based navigation will work well. Prism 4 provides functionality that simplifies transitions between views when it's not necessary to interact between the view and the view model.

We looked at some state-based navigation constructs in Chapter 7, so we won't go into detail here. Revisit Chapter 7 if you need to see examples from the *Virtual Calculator* solution.

Summary

In this chapter we took a brief look at Prism 4 navigation. We learned that there are two kinds of navigation: view-based navigation and state-based navigation. We saw that view-based navigation is used to switch views with view model interaction. We also learned that state-based navigation is used to modify controls and views that already exist in the visual tree. We also saw that these state changes can be triggered through user interaction with the UI or by the solution's internal code.

In the next chapter we're going to take a closer look at the *Virtual Calculator* solution.

Chapter 13 The Virtual Calculator Solution

In this chapter we're going to look at the *Virtual Calculator* solution with an eye toward the parts that we have yet to explain. I'll talk about some of the reasons why things were done the way they were and talk about some of the ways that the solution could be refactored.

The Virtual Calculator Solution

The first step in building a Prism 4 solution is to create a Visual Studio solution. I prefer to create a solution only. This allows me to use the naming conventions for my projects that conform to my standards.



Note: *You can, of course, simply add the correct solution name to the solution textbox. I always forget!*

If a project is selected (such as a WPF Application), both the solution and WPF project is created. The Project name is the same as the solution name, for instance, PRISM4.PRISM4. Dot notation is used to separate the solution name from the project name. I prefer to name my solutions and projects in the following manner: PRISM4.MAIN, PRISM4.INFRASTRUCTURE, etc. I feel that using this type of notation better describes each project in the solution.



Tip: *The techniques and layout that I use in this book are those that work best for me. If you are new to Prism 4, I suggest that you start out by using my techniques and layout. As you become more proficient, you can set up your solutions to your taste.*

The Main (Startup) Project

The *Main* project is the project where WPF and Prism 4 solutions start. The *Main* project is a WPF Application project. This project is where the composition root and bootstrapper reside. It's also where the shell form for the solution is located.

Other necessary classes and files also reside in this project. Region adapters are typically located in the main project and images and other files such as the *App.config* file are located in this project.

Loading DLL References

Before we can use Prism 4 with the project, we'll need to load the Prism 4 library. When we downloaded Prism 4, the required DLLs were included in the files.



Tip: When working with Prism 4 solutions, I move all of the needed DLL files (Prism 4 and other necessary files) into a central folder. This allows me to quickly find the needed DLLs. The Prism 4 WPF DLLs are located in the \Bin\Desktop folder of the Prism 4 download.

You also have the option to automatically load Prism 4 DLL files into your Visual Studio solutions by using the RegisterPrismBinaries tool. This tool is located in the downloaded Prism 4 folder.

The following selected DLL files need to be added to the project:

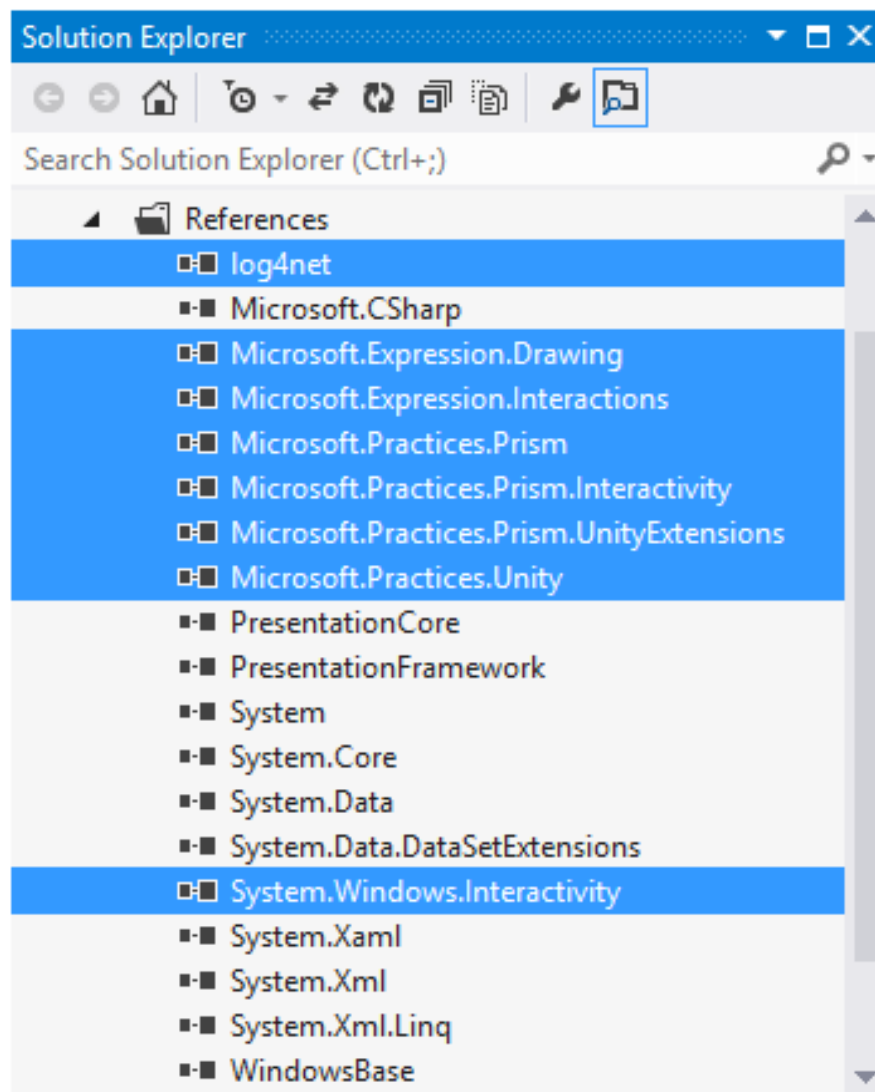


Figure 14: The Main project DLL References

Although we won't talk about using the Log4Net DLL, this is a frequently used library in my Prism 4 solutions, so it was added also. Prism 4 has a default logging mechanism that logs all activity to the Output Console. You don't have to really do anything to get this functionally in Prism 4 solutions. You also have the option of using a third-party logger, and that's where the Log4Net logger comes into the picture.

The Microsoft.Expression.Drawing DLL is used to draw Vector graphics. This and all of the Microsoft.Expression DLLs are from Expression Blend and Expression Design.

We need the Microsoft.Expression Interactions and Interactivity libraries to work with triggers and behaviors in our solutions.

The Microsoft.Practices.Prism DLL is the main Prism 4 library. It contains classes that allow for working with event aggregation, regions, modules, navigation, and any other Prism 4 features.

The Microsoft.Practices.Unity and UnityExtensions DLLs are used with the Unity DI container. This is the DI container that we work with in this solution. There are libraries for the Managed Extensibility Framework (MEF) DI container, but it's not necessary to add the DLL, because we won't use MEF.

These are the minimum libraries that are needed to work with all of the features of Prism 4. There are other libraries, such as Service Location, but they are not necessary to use the Prism 4 framework in the manner in which it is used in this book.

Setting Up the Main Project Folder Structure

Now that we have the necessary project references, our next step is to set up the project folders. There are any number of ways that this can be done; I have found that the following folder structure works best for me.

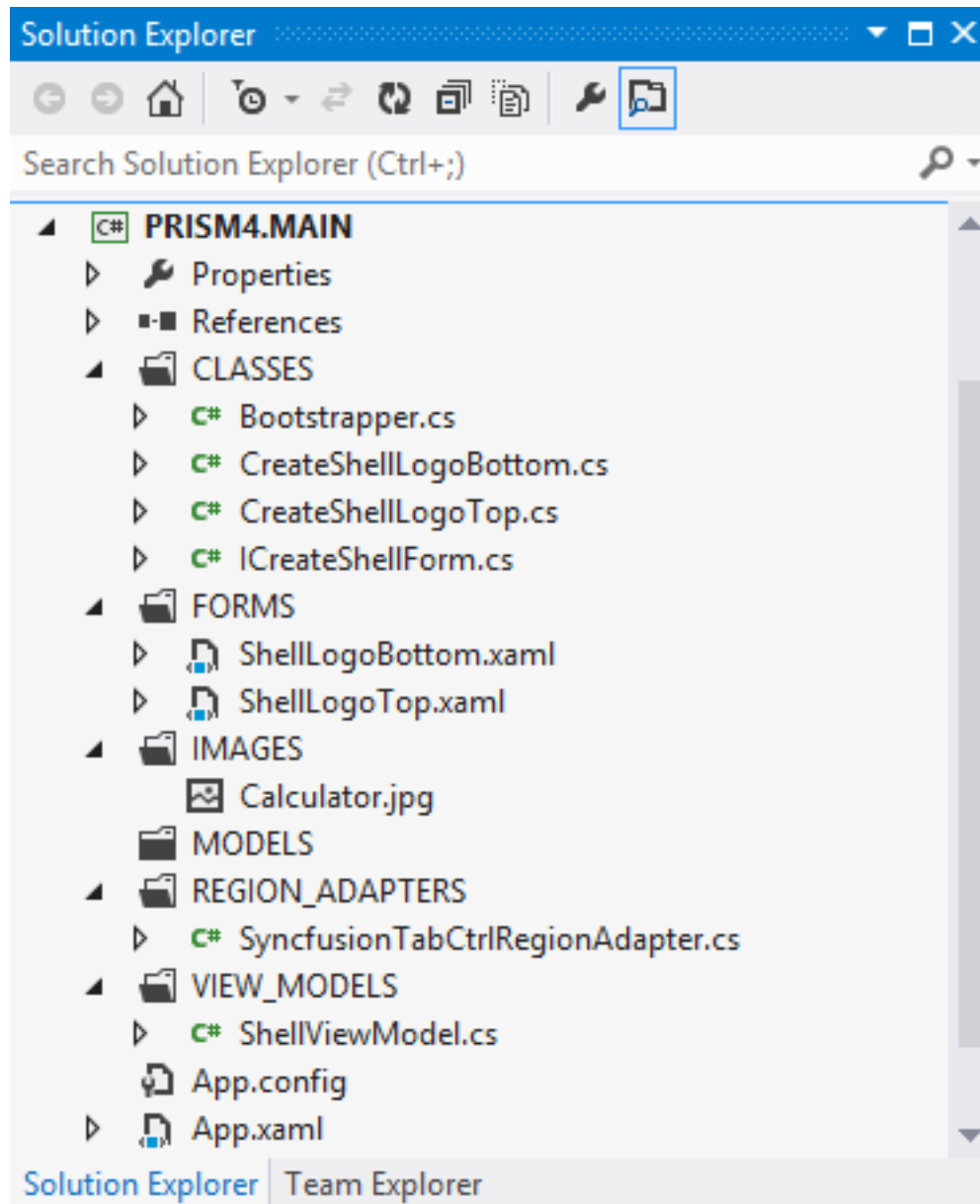


Figure 15: The Main Project Folder Layout

Notice that I use all upper case for the folder names. This differentiates folders that I have added from Visual Studio folders.

The Shell Form

For now, think of the Prism 4 shell form as the main display that the solution will use. This Windows form is where we will compose the views that make up the solution. We talked about the *Virtual Calculator* shell forms in Chapter 7.

Unlike regular WPF forms, shell forms don't contain a lot of content at design time. In fact, shell forms can be designed without any content initially. In the *Virtual Calculator* solution we dynamically added views to the shell form from modules at run time. We went into detail about this process in Chapter 9 when we talked about modules.

The App.xaml File

The App.xaml file is the normal starting point (Composition Root) for WPF solutions. The App.xaml file uses the **StartupUri** tag to point to the Windows form (MainWindow.xaml in this case) in the *Main* project. As we progressed with setting up our Prism 4 WPF solution, we changed this file in order to have the project conform with Prism 4. In Listing 60, we removed the bold **StartupUri** property.

Listing 62: The App.xaml File Markup

```
<Application
  x:Class="PRISM4.MAIN.App"

  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

  StartupUri="MainWindow.xaml">

  <Application.Resources>

  </Application.Resources>

</Application>
```

The App.xaml.cs File

While the App.xaml file works well for simple WPF solutions, Prism 4 solutions are started by calling a **Bootstrapper** class. The simplest way to do this is to remove the **StartupUri** from the App.xaml file and call the **Bootstrapper** class from the App.xaml.cs file. This file serves as a code-behind file for the App.xaml file, and can replace it as the startup file for WPF solutions. We talked about this file when we built the Hello Prism 4 solution in Chapter 3.

The App.config File

The App.config file is the configuration file for the *Main* project and is also used by all of the other projects in the solution. This file is used to provide late binding to the solution and can also be used to create appenders for third-party loggers used with Prism 4. If Microsoft Entity Framework is used, connection strings and many other string-based items can be added to this file.

The App.config file is used in the *Virtual Calculator* solution to load modules at solution startup. Each of the three modules in the solution have XAML markup that defines the module and is used to load the module. Listing 63 shows an example of the App.config file for the *Virtual Calculator* solution.

Listing 63: The Virtual Calculator App.config File Markup

```
<?xml
  version="1.0"
  encoding="utf-8" ?>

<configuration>

  <configSections>

    <section
      name="modules"
      type=
        "Microsoft.Practices.Prism.Modularity.ModulesConfigurationSection,
        Microsoft.Practices.Prism"/>

  </configSections>

  <modules>

    <module
      assemblyFile="MODULES/PRISM4.APPLICATION_LOGO_MODULE.dll"

      moduleType=
        "PRISM4.APPLICATION_LOGO_MODULE.APPLICATION_LOGO_MODULE,
        PRISM4.APPLICATION_LOGO_MODULE,
        Version=1.0.0.0,
        Culture=neutral,
        PublicKeyToken=null"

      moduleName="APPLICATION_LOGO_MODULE"

      startupLoaded="True">

    </module>

    <module
      assemblyFile="MODULES/PRISM4.COMPANY_LOGO_MODULE.dll"

      moduleType=
        "PRISM4.COMPANY_LOGO_MODULE.COMPANY_LOGO_MODULE,
        PRISM4.COMPANY_LOGO_MODULE,
        Version=1.0.0.0,
        Culture=neutral,
        PublicKeyToken=null"

      moduleName="COMPANY_LOGO_MODULE"

      startupLoaded="True">

    </module>

    <module
      assemblyFile="MODULES/PRISM4.MATH_MODULE.dll"
```

```

    moduleType=
      "PRISM4.MATH_MODULE.MATH_MODULE,
      PRISM4.MATH_MODULE,
      Version=1.0.0.0,
      Culture=neutral,
      PublicKeyToken=null"

    moduleName="MATH_MODULE"

    startupLoaded="True">

  </module>

</modules>

<startup>
  <supportedRuntime
    version="v4.0"
    sku=".NETFramework,Version=v4.5" />
</startup>

</configuration>

```

First, a module section is created in the App.config file's **configSections** section. The modules section is where all of the Prism 4 module markup for the solution will reside. The *Virtual Calculator* solution has three modules:

1. Company Logo Module
- Application Logo Module
- Math Module

Each of the three modules have markup in the App.config file that identifies the module. The markup is used to load each module when the solution is started. Let's take a look at one of the module loaders.

Listing 64: The App.config File Math Module Markup

```

<module
  assemblyFile="MODULES/PRISM4.MATH_MODULE.dll"

  moduleType=
    "PRISM4.MATH_MODULE.MATH_MODULE,
    PRISM4.MATH_MODULE,
    Version=1.0.0.0,
    Culture=neutral,
    PublicKeyToken=null"

    moduleName="MATH_MODULE"

    startupLoaded="True">

</module>

```

The first property that we encounter after the module tag is the **assemblyFile** property. This property points to the path and filename of the module's DLL file. The *Virtual Calculator* solution stores all of its module DLL files in the MODULES folder, which is located in the debug or release folder of the solution.

The **moduleType** property is the namespace and name of the module class that is associated with the module project. The solution name and project name is the second part of this property. The **Version**, **Culture**, and a null **PublicKeyToken** are also added to this property.

The **moduleName** property is the name of the module class.

The **startupLoaded** property is a **Boolean** type that will load the module at solution startup if **true** and will not load the module if set **false**.

Each module in the solution uses the same markup with its associated module data.



Tip: Keep in mind that the App.config file is not a secure file! Don't save data that could compromise security, such as passwords, in this file.

The Bootstrapper CreateModuleCatalog Method

Even though each module is added to the App.config file, this is not enough to load the Prism 4 modules. The **Bootstrapper** class needs to know how modules will be loaded. This is done with the bootstrapper **CreateModuleCatalog** method. Listing 65 shows an example of this method.

Listing 65: The Bootstrapper CreateModuleCatalog Method

```
protected override IModuleCatalog CreateModuleCatalog()
{
    ConfigurationModuleCatalog configurationCatalog = new ConfigurationModuleCatalog();
    return configurationCatalog;
}
```

First, the **configurationModuleCatalog** is created. This type of catalog defaults to the App.config file in the startup project, and is then returned. This is all that is needed to configure the solution for App.config file loading.

This method of loading modules works well for solutions with a small number of modules. As the number of modules increases, this kind of module loading can become error-prone. Working with App.config files (or XAML files for that matter) and large numbers of modules are problematic because these files are case-sensitive, and therefore brittle. If you anticipate that large numbers of modules will be used in a solution, directory scan is a better option for module loading.

The Project Post-Build Event

There is one more issue that needs to be addressed before we leave the main project. How do we ensure that the DLLs that are generated for modules are moved to the MODULES folder?

Each time that we build the solution, the added changes are appended to an updated DLL file for each project. These updated files need to be synchronized with the DLL files in the MODULES folder that Prism 4 uses. There is no reason why these files can't be manually copied each time that a build is completed, but using this method to update the DLL files would be error-prone and time-consuming.

An automatic method of updating the DLL files can be implemented by using each module project's post-build event.

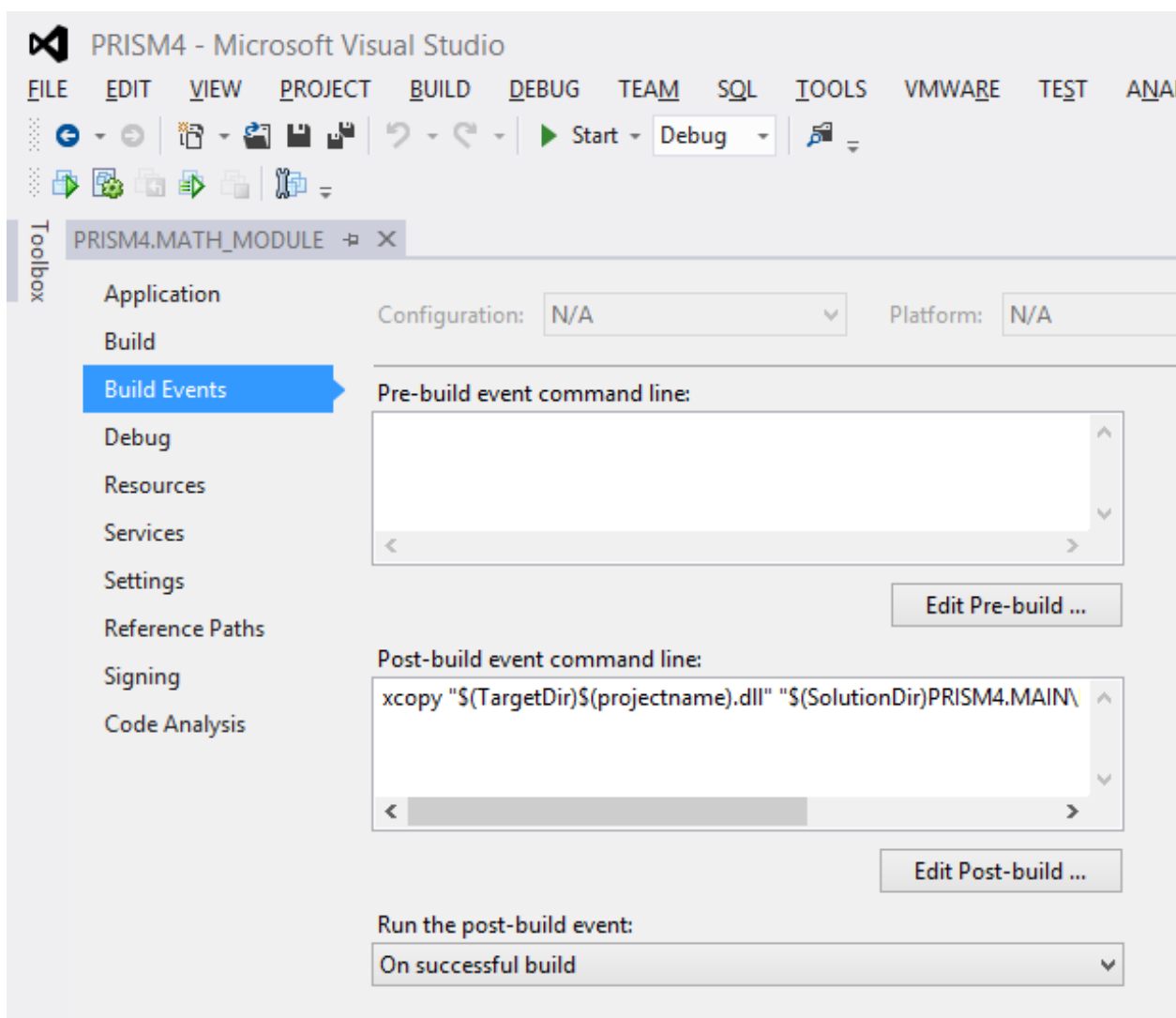


Figure 16: The Post-Build Event for the Math Module project

Listing 66 shows an example of the command line that I use to update the DLL files.

Listing 66: Using the XCopy command to Automatically Synchronize Module DLL Files

```
xcopy "$(TargetDir)$(projectname).dll"
"$(SolutionDir)PRISM4.MAIN\bin\$(ConfigurationName)\MODULES\" /Y
```

This command line simply copies the DLL file that is located in the module project to the MODULES folder that is located in the debug or release folder of the main (Startup) project. The **ConfigurationName** placeholder copies the DLL file to the correct folder (debug or release), depending on the solution's configuration setting.

Each time that a Build Solution or Rebuild Solution is executed in Visual Studio, the post-build event copies the project's updated DLL file to the MODULES folder. It's important to use the post-build event because this event is triggered after the DLL file has been updated.

The Infrastructure Project

The *Infrastructure* project is a global project that can be safely referenced by all other projects in the solution. This type of project is a *Class Library* project.

One of Prism 4's main objectives is to promote loose coupling by creating an environment where modules and the main project are autonomous entities that are not tightly coupled to one another. One way to achieve this is to eliminate references between projects.

Why is the Infrastructure Project Needed?

In the case of the *Infrastructure* project, items that need to be accessed by multiple modules and other projects in the solution can use this project as a central location for entities, services, or any other items that may need to be shared.

Loading Prism 4 DLL Files in the Infrastructure Project

The *Infrastructure* project should have the same Prism 4 references loaded as the *Main* project. This will ensure that the needed libraries are available for event aggregation, commanding, and any other Prism 4 features that need to be used by this project.

Loading Other DLL Files in the Infrastructure Project

Because the *Infrastructure* project will serve as a central warehouse for global entities, there is no set group of libraries that need to be added to this project. For instance, if the Entity framework is used in your solution, you'll want to load the *System.Data.Entity* library. If you use ADO .NET, on the other hand, you'll want the *System.Data.SqlClient* library. So the basic rule with the *Infrastructure* project references is to load whatever is needed.



Tip: As I stated earlier, how you design this project should reflect what works best for you. I use Microsoft Entity Framework - Code First with the Fluent API, and my folder layout reflects this organizational structure. The bottom line is to use what works best for you.

The Infrastructure Project Folder Structure

Figure 16 displays an example of the initial folder layout that I use for my Prism 4 *Infrastructure* projects.

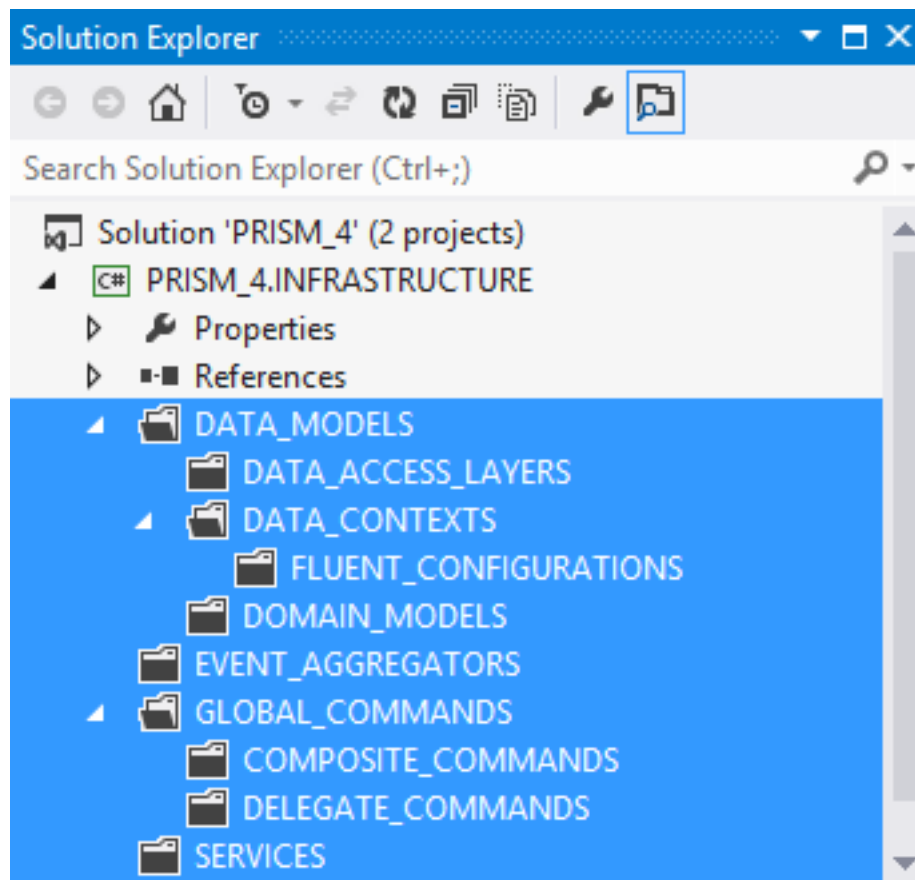


Figure 17: The Infrastructure Project Folder Structure

The highlighted items are the added folders.

The DATA_MODELS folder contains all of the folders and entities that are needed to use Microsoft Entity Framework – Code First. I group my data access layers, data contexts, and domain models into the root DATA_MODELS folder. If it becomes necessary to split the different folders into separate projects, I can just move the folder and modify the code. Notice that the FLUENT_CONFIGURATIONS folder is nested in the DATA_CONTEXTS folder. I prefer to use separate fluent classes for each **DbSet** in the data context, and I find that this is a good way to organize the classes. Relational databases are not used in the *Virtual Calculator* solution, so these folders are empty.

The EVENT_AGGREGATORS folder is used to hold event aggregator classes that must be global to all modules in the solution. This is one way to implement module-to-module communication in a loosely coupled manner. See Chapter 11 for more about event aggregation. If you want to see an example of classes in this folder, take a look at the EA_VBN demo solution.

The `GLOBAL_COMMANDS` folder is used for two types of commands: composite commands and global delegate commands. I tend to use delegate commands locally in module View Models, but they can also be used for module-to-module communication when created globally in the Infrastructure project. I try to use event aggregation exclusively for module-to-module communication, but find that global commands make sense in some cases. We covered local delegate commands in more depth in Chapter 10.

The `COMPOSITE_COMMANDS` folder contains all of the solutions' composite commands. A composite command is a single command that is used to trigger any number of other commands. This type of command can come in handy when you need to (for instance) save data across more than one module. We won't be using composite commands in this book, but there is a wealth of information about it on the internet and in the Prism 4 manual.

The `SERVICES` folder contains all of the global services that are used in the solution. This folder is used to supply each project with entities that may be used by any module or project in the solution.

Prism 4 Module Projects

Prism 4 module projects are the most prevalent type of project in Prism 4 solutions. These are the projects that supply the views and provide the business rules for the solution. They are also where the Model–View–ViewModel (MVVM) design pattern is implemented. Module projects are the entities that provide loose coupling and separation of concerns in Prism 4 projects.

These are the projects that define the solution's appearance and actions. They populate the shell form with the user interfaces (views) that the users of the solution interact with and provide an infrastructure for view navigation. In a nutshell, a Prism 4 solution without at least one module doesn't make a lot of sense.

How are Modules Used?

Prism 4 Module projects are Class Library projects. When adding modules, use the **Class Library project** selection in the **Add New Project** dialog.

Prism 4 modules are designed to be standalone entities. Although modules can set dependencies between one another, this is limited to the order in which the modules are created. Tightly bound dependencies (references) are discouraged.

A module's job is to define a discrete part of a Prism 4 solution. The module can include a view or views (the UI), validation, data access, and business rules (through the model) coordination between views and models, supplied through the view model. The module can also allow the user to take actions (commands, triggers, behaviors, and event aggregation, to name a few).

So modules are used to define a self-contained but complete part of a Prism 4 solution. Internally, separation of concerns are enforced with the use of the MVVM design pattern.

Loading Prism 4 DLL Files into each Module Project

The Module project should have the same Prism 4 references loaded as the Main project. This will ensure that the needed libraries are available for the **IModule** interface, regions, and any other Prism 4 functionality that needs to be used by the project.

Loading Other DLL Files in the Module Project

The **System.Xaml** and **WindowsBase** libraries should be added to all Prism 4 Module projects. Add other libraries as needed.

The Module Project Folder Structure

Figure 18 shows the initial folder layout that I use for my *Prism 4 Module* projects. The highlighted items are the added folders.

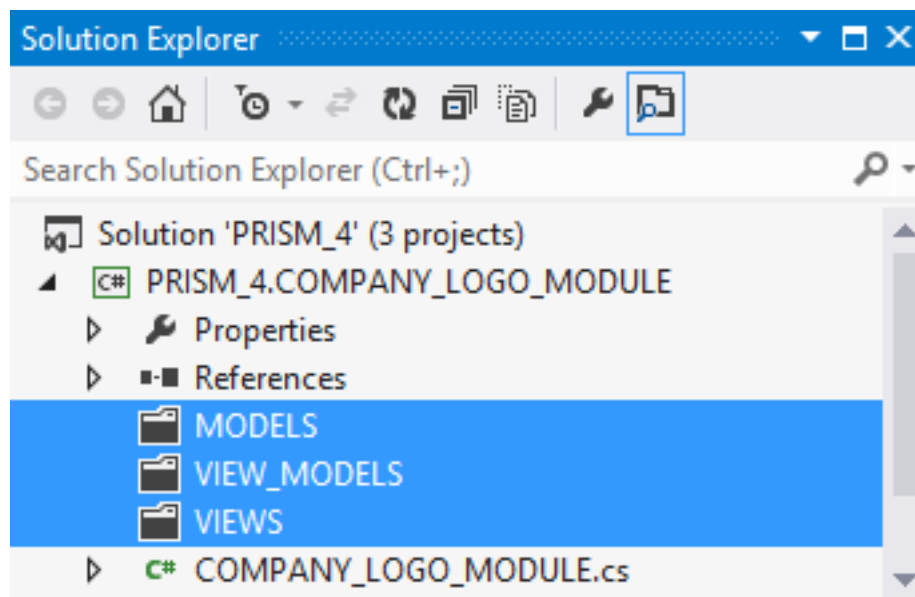


Figure 18: The Module Project Folder Structure

The MODELS folder is used for business rule and data access classes. Although the VIEW_MODELS classes are aware of the classes in this folder, the VIEWS user controls have no knowledge of this folder's classes. This ensures that the separation of concerns between the view and model are enforced.

The VIEW_MODELS folder is used as an intermediary between views and models. The view model has knowledge of the model classes, but no knowledge of any views. The view model class is where most of the work is done in each module. Its job is to expose functionality and data to the view even though it has no knowledge of the view. It's also responsible for calling methods in the model that enforce business rules and gathering data from remote data stores.

The VIEWS folder is where user interfaces reside. These user interfaces can be user controls, data templates, or Windows forms. XAML is used to build and format each view. Views use bindings to access functionality and data from the view model by using the view's data context. We talked about the data context in Chapter 6. The view has knowledge of the view model through the data context, but has no knowledge of the module's model.

The Module Class

The module class is what makes a project a Prism 4 module. This class is responsible for initialization of each module.

The IModule Interface

The module class implements the **IModule** interface; this interface is used to convert the project into a Prism 4 module. The **IModule** interface exposes a single method: **Initialize**. This method is used by the module to set up its initial environment. This usually entails loading views contained in the module to regions in the shell form or to regions in other views.

The Company Logo Module Project

The *Company Logo Module* project is responsible for loading the animated company logo graphic into the shell form.

The graphic was created with Microsoft Expression Design. The animations were added using Expression Blend. Expression Design and Expression Blend are components of the Microsoft Expression Studio Ultimate suite. Expression Design is used in the same manner as Adobe Illustrator; it's a vector-based graphic design tool. Expression Design can also be used with photos. Expression Blend, on the other hand, is an XAML code generator. These two tools in combination allow for the fast creation of powerful graphic and XAML constructs.

One interesting aspect of the *Company Logo Module* is that instead of using a local user control, I opted to use a DLL file for the graphic and animation content. This allows the company logo DLL to be used across different solutions. Using the DLL file in this manner allows for a consistent look and layout across any number of applications that the company creates.

One important fact to remember: if a DLL file is used in a module that renders a view, you must also reference that file in the *Main* project. Failure to do so will result in an exception.

The Application Logo Project

The *Application Logo* project is responsible for loading the application logo into the shell form. This module uses a local user control to display the application logo. A local user control was used because the application logo is specific to each application and does not need to be shared.



Tip: It's not really necessary to create two modules (one for each logo view) in the solution. I could have just as easily created a Logo Module project and loaded each module (Company Logo and Application Logo) from the single

module. I added the two modules in the demonstration solution to show more examples of module loading in Prism 4 solutions.

Another interesting fact about the two logo modules is that there is no view model or model code in either project. Each view has no need for the other classes because they use XAML exclusively; even the automation code in the company logo view is XAML.

The Math Project

The *Math Module* project is where most of the work is done in the solution. This project is where the two math views (AddTwoView and SubtractView) are located. The views not only define the look and layout, but also contain the behaviors and states that are used to update the UI as user interaction progresses during the solution's lifetime.

The view model exposes an interface that allows the view to set the different states and provides properties that make the data entered into the view available to the model for processing.

I use the model classes as intermediaries between the services in the infrastructure project and the view model and views. I find that using intermediate classes allows me to better use design patterns such as decorator to add functionality to my existing classes without violating design principles (such as the open/closed principle).

The Unit Test Project

The *Unit Test* project uses NUnit to test most of the service classes in the solution. I won't go into detail about this project because testing is not within the purview of this book. Feel free to look at my tests to get a feel for how you may want to set up your test projects.

Chapter 14 Conclusion

What We Learned

In this e-book, we explored Microsoft Prism 4 and how it relates to Windows Presentation Foundation (WPF). We learned the different features that Microsoft Prism 4 exposes and talked about their functionality. We worked our way through the *Hello World*, *Virtual Calculator*, and the *Event Aggregation* solutions to deepen our understanding.

Microsoft Prism 4 is designed to help developers to build well architected XAML-based solutions. Hopefully this book will help in your study of Prism 4 and WPF.

References

I have added some resources that I have found to be useful in my studies. I hope that they will be of use to you also.

Microsoft Prism 4

Brumfield, Bob, Geoff Cox, David Hill, et al. *Developer's Guide to Microsoft Prism 4: Building Modular MVVM Applications Using Windows Presentation Foundation and Microsoft Silverlight*. Microsoft Press, 2001.

Microsoft WPF

MacDonald, Matthew. *Pro WPF in C# 2010: Windows Presentation Foundation in NET 4*. APress, 2010

Nathan, Adam. *WPF 4 Unleashed*. Sams Publishing, 2010

Dependency Injection

Seeman, Mark. *Dependency Injection in .NET*. Manning Publications Co., 2011.

Design Patterns

Gamma, Erich, Richard Helm, Ralph Johnson, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

Freeman, Eric, Bert Bates, Kathy Sierra, et al. *Head First Design Patterns*, O'Reilly Media, 2004.

Object-Oriented Programming

McLaughlin, Brett, Gary Police, and Dave West. *Head First Object-Oriented Analysis and Design*. O'Reilly Media, 2006

Enterprise Architecture

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002