



Developer's Guide to Microsoft Prism Library 5.0 for WPF

Microsoft patterns & practices

April 2014



patterns & practices

Copyright

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

© 2014 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, Windows Vista, Silverlight, Expression Blend, MSDN, IntelliSense, Visual C#, Visual C++, and Visual Studio are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Contents

Download and Setup Prism.....	8
Documentation	8
NuGet Packages	8
Download and Setup the Prism Source Code	9
Adding Prism Library Source Projects to Solutions.....	13
Related Downloads	13
What's New in Prism Library 5.0 for WPF.....	14
New Guidance.....	14
Changes in the Prism Library.....	14
CodePlex Issues Resolved	16
Example Code Changes.....	17
NuGet Packages Now Available	18
The Team Who Brought You This Guide.....	19
1: Introduction	20
Why Use Prism?	20
Prerequisites	23
An Overview of Prism.....	24
2: Initializing Applications	37
What Is a Bootstrapper?	37
Key Decisions	38
Core Scenarios	38
3: Managing Dependencies Between Components.....	45
Key Decision: Choosing a Dependency Injection Container	46
Core Scenarios	47
Using Dependency Injection Containers and Services in Prism.....	51
IServiceLocator.....	52
Considerations for Using IServiceLocator	54
4: Modular Application Development	55
Benefits of Building Modular Applications	55

Core Concepts	56
Key Decisions	60
Core Scenarios	65
5: Implementing the MVVM Pattern	77
Class Responsibilities and Characteristics.....	77
Class Interactions	82
Construction and Wire-Up.....	93
Key Decisions	96
6: Advanced MVVM Scenarios.....	98
Commands	98
Interaction Triggers and Commands.....	104
Handling Asynchronous Interactions	106
User Interaction Patterns.....	108
Advanced Construction and Wire-Up	114
Testing MVVM Applications.....	117
7: Composing the User Interface	123
UI Layout Concepts	124
UI Layout Scenarios.....	133
UI Design Guidance.....	150
UI Layout Key Decisions	165
8: Navigation	167
Navigation in Prism	167
9: Communicating Between Loosely Coupled Components.....	190
Solution Commanding.....	190
Region Context.....	193
Shared Services	194
Event Aggregation.....	195
10: Deploying Applications	202
Deploying WPF Prism Applications	202
11: Glossary.....	207
12: Patterns in the Prism Library	210

Adapter	210
Application Controller Pattern.....	211
Command Pattern.....	211
Composite and Composite View.....	211
Dependency Injection Pattern	212
Event Aggregator Pattern	212
Façade Pattern	212
Inversion of Control Pattern	213
Observer Pattern.....	213
Model-View-ViewModel Pattern.....	213
Registry Pattern	214
Repository Pattern.....	214
Separated Interface and Plug-In	214
Service Locator Pattern.....	214
13: Prism Library	217
Add Reference using NuGet and Accessing the Library Source Code.....	218
Organization of the Prism Library	218
The Prism Library Source	219
Modifying the Library.....	219
Running the Tests.....	220
14: Upgrading from Prism Library 4.1.....	221
15: Extending the Prism Library	224
Guidelines for Extensibility	224
Recommendations for Modifying the Prism Library.....	225
Extensibility Points in the Prism Library.....	226
Container and Bootstrapper	227
Logging	231
Modules	232
Regions.....	235
Region Navigation.....	240
View Model Locator	242

16: Code Samples.....	244
Installing Prism.....	244
Stock Trader Reference Implementation.....	248
Building and Running the Reference Implementation	249
The Scenario.....	251
Stock Trader RI Features.....	255
Logical Architecture	256
Implementation View	257
How the Stock Trader RI Works.....	257
Technical Challenges.....	265
Modularity QuickStarts.....	268
Building and Running the QuickStarts	270
Walkthrough	271
Implementation Details	275
Key Modularity Classes	280
Acceptance Tests	280
Interactivity QuickStart.....	281
Building and Running the QuickStart.....	282
Implementation Details	282
Key Interactivity Classes.....	292
Acceptance Tests	293
MVVM QuickStart	294
Building and Running the QuickStart.....	295
Implementation Details	295
Commanding QuickStart.....	302
Building and Running the QuickStart.....	302
Implementation Details	303
Acceptance Tests	307
UI Composition QuickStart.....	308
Building and Running the QuickStart.....	308
Implementation Notes.....	309

Acceptance Tests	314
State-Based Navigation QuickStart.....	315
Building and Running the QuickStart.....	316
Implementation Details	316
Acceptance Tests	325
View-Switching Navigation QuickStart	326
Building and Running the QuickStart.....	327
Implementation Details	327
Navigation Support in the Prism Library.....	328
Using the Prism Library for Navigation	331
Unit and Acceptance Tests.....	338
Event Aggregation QuickStart.....	340
Building and Running the QuickStart.....	340
Implementation Details	341
Unit and Acceptance Tests.....	344
17: Getting Started Using the Prism Library Hands-on Lab	345
System Requirements	345
Procedures	346
Task 1: Creating a Solution Using the Prism Library	346
Task 2: Adding a Module.....	353
Task 3: Adding a View	357
18: Publishing and Updating Applications Using the Prism Library Hands-on Lab.....	361
System Requirements	361
Preparation	361
Procedures	362
Task 1: Publishing an Initial Version of the Shell Application	362
Task 2: Updating the Manifests to Include Dynamically Loaded Module Assemblies	369
Task 3: Deploying the Initial Version to a Client Computer	373
Task 4: Publishing an Updated Version of the Application and Updating the Manifests.....	375
Task 5: Deploying the Updated Version to a Client Computer	377
Bibliography	378

Download and Setup Prism

Learn what's included in Prism 5.0 including the documentation, WPF code samples, and libraries. Additionally find out where to get the library and sample source code and the library NuGet packages.

For a list of the new features, assets, and API changes, see [What's New in Prism 5.0](#).

Documentation

Prism includes the following documentation:

- [Developer's Guide to Microsoft Prism 5.0 on MSDN](#).
 - [Prism Reference Documentation on MSDN](#).
 - [Developer's Guide to Microsoft Prism 5.0 in .pdf format](#).
 - [Prism Reference Documentation in chm format](#).
-

NuGet Packages

- [Prism](#): Downloads NuGet dependency packages—Prism.Composition, Prism.Interactivity, Prism.Mvvm, and Prism.PubSubEvents NuGet Packages.
 - [Prism.Composition](#): Modularity, UI Composition, Bootstrapping, Interactivity, IActiveAware, Navigation, and deprecated NotificationObject and PropertySupport.
 - [Prism.Interactivity](#): Interactivity.
 - [Prism.Mvvm](#): The Portable Class Library for MVVM and the associated platform specific code to support MVVM. Includes Commanding, BindableBase, ErrorsContainer, IView, and ViewModelLocationProvider.
 - [Prism.PubSubEvents](#): The Portable Class Library for PubSubEvents.
 - [Prism.UnityExtensions](#): Use these extensions to Prism to build Prism applications based on Unity.
 - [Prism.MefExtensions](#): Use these extensions to Prism to build Prism applications based on Managed Extensibility Framework (MEF).
-

The following table shows common Prism namespaces and in which assemblies and NuGet packages they can be found.

Namespace	Assembly	NuGet Package
Microsoft.Practices.Prism.Logging	Microsoft.Practices.Prism.Composition	Prism.Composition
Microsoft.Practices.Prism.Modularity		
Microsoft.Practices.Prism.Regions		
Microsoft.Practices.Prism.Interactivity	Microsoft.Practices.Prism.Interactivity	Prism.Interactivity
Microsoft.Practices.Prism.Commands	Microsoft.Practices.Prism.Mvvm	Prism.Mvvm
Microsoft.Practices.Prism.Mvvm		
Microsoft.Practices.Prism.ViewModel		
Microsoft.Practices.Prism.PubSubEvents	Microsoft.Practices.Prism.PubSubEvents	Prism.PubSubEvents

Download and Setup the Prism Source Code

This section describes how to install Prism. It involves the following three steps:

1. Install system requirements.
2. Download and extract the Prism source code and documentation.
3. Compile and run the QuickStarts, Reference Implementation, or Prism Library source code.

Step 1: Install System Requirements

Prism was designed to run on the Microsoft Windows 8 desktop, Microsoft Windows 7, Windows Vista, or Windows Server 2008 operating system. WPF applications built using this guidance require the .NET Framework 4.5.

Before you can use the Prism Library, the following must be installed:

- Microsoft .NET Framework 4.5 (installed with Visual Studio 2012) or Microsoft .NET Framework 4.51.
- Microsoft Visual Studio 2012 or 2013 Professional, Premium, or Ultimate editions.

Note: Visual Studio 2013 Express Edition can be used to develop Prism applications using the Prism Library.

Optionally, you should consider also installing the following:

- [Microsoft Blend for Visual Studio 2013](#). A professional design tool for creating compelling user experiences and applications for WPF.

Step 2: Download and Extract the Prism Source Code and Documentation

You can download the source code for the Prism library, the reference implementation and the QuickStarts from the following link:

- [Prism 5.0](#)

To install the Prism assets, right-click the exe file or zip file, and then click **Run as administrator**. This will extract the source code into the folder of your choice.

Note: The Stock Trader Reference Implementation and the QuickStarts can also be downloaded separately. The table below provides links to the source code for each.

Sample	Category	Description
Stock Trader Reference Implementation	Prism	<p>The Stock Trader RI application is a reference implementation that illustrates the baseline architecture. Within the application, you will see solutions for common, and recurrent, challenges that developers face when creating composite WPF applications.</p> <p>The Stock Trader RI illustrates a fictitious, but realistic financial investments scenario. Contoso Financial Investments (CFI) is a fictional financial organization that is modeled after real financial organizations. CFI is building a new composite application to be used by their stock traders.</p>
Hello World Hands-on Lab	Get Started	<p>The Hello World Hands-on Lab demonstrates the end solution for the hands-on lab "Getting Started Using the Prism Library 5.0 for WPF Hands-on Lab." In this lab, you will learn the basic concepts of Prism and apply them to create a Prism Library solution that you can use as the starting point for building a composite WPF.</p>

<ul style="list-style-type: none"> • Modularity QuickStarts for Unity • Modularity QuickStarts for MEF 	Modularity	The Modularity QuickStarts demonstrate how to code, discover, and initialize modules using Prism. These QuickStarts represent an application composed of several modules that are discovered and loaded in the different ways supported by the Prism Library using MEF and Unity as the composition containers.
MVVM QuickStart	MVVM	The MVVM QuickStart demonstrates how to build an application that implements the MVVM presentation pattern, showing some of the more common challenges that developers can face, such as wiring a view and view model using the ViewModelLocator, validation, UI interactions, and data templates.
Commanding QuickStart	Commanding	The Commanding QuickStart demonstrates how to build a WPF UI that uses commands provided by the Prism Library to handle UI actions in a decoupled way.
UI Composition QuickStart	UI Composition	This QuickStart demonstrates how to build WPF UIs composed of different views that are dynamically loaded into regions and that interact with each other in a decoupled way. It illustrates how to use both the view discovery and view injection approaches for UI composition.
State-Based Navigation QuickStart	Navigation	This QuickStart demonstrates an approach to define the navigation of a simple application. The approach used in this QuickStart uses the WPF Visual State Manager (VSM) to define the different states that the application has and defines animations for both the states and the transitions between states.

<u>View-Switching Navigation QuickStart</u>	Navigation	This QuickStart demonstrates how to use the Prism Region Navigation API. The QuickStart shows multiple navigation scenarios, including navigating to a view in a region, navigating to a view in a region contained in another view (nested navigation), navigation journal support, just-in-time view creation, passing contextual information when navigating to a view, views and view models participating in navigation, and using navigation as part of an application built through modularity and UI composition.
<u>Event Aggregation QuickStart</u>	Event Aggregation	This QuickStart demonstrates how to build a WPF application that uses the Event Aggregator service. This service enables you to establish loosely coupled communications between components in your application.
<u>Interactivity QuickStart</u>	Interactivity	This QuickStart demonstrates how to create a view and view model that work together when the view model needs to interact with the user or user gesture needs to raise an event that invokes a command. In each of these scenarios the view model should not need to know about the view. The first scenario is handled by using InteractionRequests and InteractionRequestTriggers . The second scenario is handled by InvokeCommandAction .

Step 3: Compile and Run QuickStarts, Reference Implementation, or Prism Library Source Code

In order to build and run the reference implementation and the QuickStarts, select the appropriate shortcut file and press F5 to build and run.

The reference implementation and QuickStarts use NuGet references to the Prism library assemblies so you can compile and run each solution directly.

Adding Prism Library Source Projects to Solutions

As part of shipping the Prism Library as NuGet packages, the Prism Library projects were removed from the solutions of all QuickStarts and reference implementation projects. If you are a developer accustomed to stepping through the Prism Library code as you build your application, there are a couple of options:

- **Add the Prism Library Projects back in.** To do this, right-click the solution, point to **Add**, and then click **Existing project**. Select the Prism Library projects. Then, to prevent inadvertently building these, click **Configuration Manager** on the **Build** menu, and then clear the **Build** check box for all Prism Library projects in both the debug and release configurations.
- **Set a breakpoint and step in.** Set a break point in your application's bootstrapper, and then step in to a method within the base class (F11 is the typical C# keyboard shortcut for this). You may be asked to locate the Prism Library source code, but often, the full program database (PDB) file is available and the file will simply open. You may set breakpoints in any Prism Library project by opening the file and setting the breakpoint.

Related Downloads

- [ManifestManagerUtility for ClickOnce](#)
 - [Getting Started Using the Prism Library Hands-on Lab](#)
 - [MVVM Training](#)
-

What's New in Prism Library 5.0 for WPF

Prism 5.0 includes guidance in several new areas, resulting in new code in the Prism Library for WPF, new and updated QuickStarts, and updated documentation. Parts of the Prism Library changed between Prism 4.1 and Prism 5.0 to support the new guidance, fix existing issues, and respond to community requests.

Note: For Silverlight applications use [Prism 4.1](#).

New Guidance

Prism 5.0 contains several areas of new guidance as follows:

- **Prism.Mvvm** is a portable class library that encapsulates Prism's MVVM support. This library includes **ViewModelLocationProvider**. Views and view models can be wired up together using the new **ViewModelLocationProvider**'s convention-based approach as explained in [Implementing the MVVM Pattern](#). View model construction can be accomplished using a dependency injection container. The ViewModel Locator's extensibility points are discussed in [Extending the Prism Library](#). **DelegateCommands** are now extendable and provide Async support. A new implementation of the **INotifyPropertyChanged** interface, the **BindableBase** class, was added.
- **The PopupWindowAction** class was added to the **Prism.Interactivity** assembly to open a custom window in response to an interaction request being raised.
The **InvokeCommandAction** action provided by Prism now passes trigger parameters to the associated command.
For more information see [Advanced MVVM Scenarios](#).
- The **EventAggregator** classes have been moved to the **Prism.PubSubEvents** portable class library.
- The **NavigationParameters** class can now be used to pass object parameters during navigation, using the overloads of the **RequestNavigate** method of a **Region** or **RegionManager** instance.

Changes in the Prism Library

Prism Library 5.0 includes changes related to new functionality, code organization, and APIs.

Code Organization

The following organizational changes were made to the library:

- The **Microsoft.Practices.Prism.PubSubEvents** portable class library contains **PubSubEvents**, **EventAggregator** and related classes.

- The **Microsoft.Practices.Prism.Mvvm** portable class library contains **ViewModelLocationProvider**, **BindableBase**, **ErrorsContainer**, **PropertySupport**, **CompositeCommand**, **DelegateCommand**, **DelegateCommandBase**, and **WeakEventHandlerManager**.
 - The **Microsoft.Practices.Prism.Mvvm.Desktop** assembly contains the WPF **ViewModelLocator** attach property.
 - The **EventAggregator** classes are marked obsolete in the Prism assembly.
 - The Silverlight and phone versions of the Prism Library were taken out. If you need these libraries download the Prism 4.1 assemblies from NuGet.
 - Prism and **Prism.Mvvm** are independent of each other but share **IActiveAware**. Therefore **IActiveAware** has been moved to **Prism.SharedInterfaces**.
-

In version 5.0 of Prism, Pub Sub Eventing functionality was moved into a separate assembly (**Prism.PubSubEvents**). **DelegateCommand**, **CompositeCommand**, and **ViewModel** support were moved into another assembly (**Prism.Mvvm**). There are many advantages to separating **PubSubEvents** and **Mvvm** from the core Prism assembly.

- You can select only the functionality that you need. If you want Regions and Modularity, you use the core Prism assembly. If you want only **ViewModel** and commanding support, you use **Prism.Mvvm**. If you only want Pub Sub Eventing, you use **Prism.PubSubEvents**. Each assembly is smaller and easier to understand.
 - You can now build your **ViewModel** code in a portable class library that leverages **Prism.Mvvm** and/or **Prism.PubSubEvents** since both are PCLs. By putting your **ViewModel** code in a PCL, your **ViewModel** code is constrained to using dependent libraries that are platform agnostic and thus can target multiple platforms.
 - Updates to these smaller libraries can be made more easily and quickly.
-

API Changes

The Prism Library API changed in several key areas. The bootstrapper was heavily modified and reusable code was added to support the new areas of guidance provided in Prism.

MVVM and Event Aggregator Changes.

Moving **ViewModel** and **EventAggregator** to a PCL causes several changes to the Prism Library. These changes include the following:

- The **BindableBase** class in **Prism.Mvvm** should be used instead of **NotificationObject**. The **NotificationObject** and **PropertySupport** classes are marked obsolete in the Prism assembly.

- When inheriting from the **BindableBase** class, use the **SetProperty** method to update the property's backing field and raise the corresponding property change event. A new **OnPropertyChanged** method that takes a lambda expression as a parameter has been added.
 - Use the **PubSubEvents** class in the **Microsoft.Practices.PubSubEvents** portable class library instead of **CompositePresentationEvents**. The classes from the Events solution folder in the Prism assembly are marked obsolete.
 - The **UriQuery** class was renamed to **NavigationParameters**, it keeps the same functionality as before, and adds support for passing object parameters.
 - **DelegateCommand** includes support for async handlers and has been moved to the **Prism.Mvvm** portable class library. **DelegateCommand** and **CompositeCommand** both use the **WeakEventHandlerManager** to raise the **CanExecuteChanged** event. The **WeakEventHandlerManager** must be first constructed on the UI thread to properly acquire a reference to the UI thread's **SynchronizationContext**.
 - **EventAggregator** now must be constructed on the UI thread to properly acquire a reference to the UI thread's **SynchronizationContext**.
 - The **WeakEventHandlerManager** is now public.
 - The **Execute** and **CanExecute** methods on **DelegateCommand** are now marked as virtual.
-

Additions to the Prism Library Core API

The following namespaces were added to the Prism Library to support the new areas of guidance added in Prism 5.0:

- **Microsoft.Practices.Prism.PubSubEvents** was added to help you send loosely coupled message using a portable class library.
 - **Microsoft.Practices.Prism.Mvvm** was added to assist you in implementing MVVM using a portable class library and several platform specific libraries.
 - **Microsoft.Practices.Prism.SharedInterfaces** has been added to share the **IActiveAware** interface between **Prism** and **Prism.Mvvm** assemblies, therefore the **IActiveAware** interface has been moved to this assembly. It is also intended for future use.
-

CodePlex Issues Resolved

- **8532: InteractionRequestTrigger** can cause memory leaks with some implementations.
- **9153: 'Notification'** really should be an interface.
- **9438: Navigation** to an existing view.
- **5495: Event to Command.**

- **8101:** `DelegateCommand` is not extendable.
 - **5623:** Make `WeakEventHandlerManager` public.
 - **9906:** A bug when using XAML module catalog in WPF.
 - **7215:** Issue with `ModuleCatalog.CreateFromXaml` for WPF application with MEF Bootstrapper.
 - **8703:** `RegionManager::IsInDesignMode`.
 - **4349:** Default Region Behavior Order Problem.
 - **3552:** Region manager in V2 fails to recognize non-WPF applications.
-

Example Code Changes

Prism 5.0 contains eleven separate code samples that demonstrate portions of the provided guidance. Several samples from Prism 4.1 were removed or replaced, and new samples added.

The following samples were added for Prism 5.0:

- **Basic MVVM QuickStart.** This QuickStart shows a very simple MVVM application that uses the `ViewModel` Locator and show a parent and child `ViewModels`. For more information, see the [MVVM QuickStart](#).
- **MVVM QuickStart.** This QuickStart was removed for this version.
- **MVVM Reference Implementation.** This reference implementation was removed for this version.
- **View-Switching Navigation QuickStart.** This QuickStart now supports WPF. It demonstrates how to use the Prism region navigation API. For more information, see [View-Switching Navigation QuickStart](#).
- **State-Based Navigation QuickStart.** This QuickStart now supports WPF. It shows an approach that uses the Visual State Manager to define the views (states) and the allowed transitions. For more information, see [State-Based Navigation QuickStart](#).
- **UI Composition QuickStart.** This QuickStart now supports WPF. It replaced the View Injection QuickStart and the View Discovery QuickStart from Prism 2.0. In the current versions, both concepts are shown in one example application. For more information, see [UI Composition QuickStart](#).
- **Interactivity QuickStart.** This new QuickStart demonstrates how to exposes an interaction request to the view through the view model. The interactions can be a popup, confirmation, custom popup, and a more complex case where the popup needs a custom view model. It also shows Prism's `InvokeCommandAction` action that passes the `EventArgs` from the trigger, as a command parameter. For more information, see [Interactivity QuickStart](#).

NuGet Packages Now Available

In your application, you can now use NuGet to add references to the Prism assemblies. These packages include:

- [Prism](#)
- [Prism.Composition](#)
- [Prism.Interactivity](#)
- [Prism.Mvvm](#)
- [Prism.PubSubEvents](#)
- [Prism.UnityExtensions](#)
- [Prism.MEFExtensions](#)

The Prism NuGet package will download the Prism.Composition, Prism.Interactivity, Prism.Mvvm, and Prism.PubSubEvents packages. You only need to add references to the Prism.Interactivity, Prism.Mvvm, and Prism.PubSubEvents if you need finer granularity.

More Information

For more information about how to upgrade a solution from version 4.1 to version 5.0 of the Prism Library, see [Upgrading from Prism Library 4.1](#).

The Team Who Brought You This Guide

Prism was produced by the following individuals:

patterns & practices Team:

Microsoft Corporation	Blaine Wastell, Francis Cheung, Nelly Delgado, Rohit Sharma, RoAnn Corbisier
Southworks SRL	Diego Poza
Icertis Inc.	Poornimma Kaliappan

Contributors to the previous release of this guidance:

Microsoft Corporation	Blaine Wastell, Bob Brumfield, David Hill, Karl Shifflett, Larry Brader, Michael Puleio, Nelly Delgado
Clarius Consulting	Fernando Simonazzi
Infosys Technologies Ltd	Mani Krishnaswami, Meenakshi Krishnamoorthi, Rathi Velusamy, Ravindra Varman, Sangeetha Manickam, Sanghamitra Chilla
Software Insight	Brian Noyes
Southworks SRL	Diego Poza, Fernando Antivero, Geoff Cox, Matias Bonaventura
TinaTech, Inc.	Tina Burden
Modeled Computation	Sharon Smith, Katie Niemer

Many thanks to the following advisors who provided invaluable assistance:

Bill Wilder of Fidelity Investments, Brian Noyes of Solliance, Brian Lagunas of Infragistics, Clifford Tiltman of Morgan Stanley, Rob Eisenberg of Blue Spire, Norman Headlam, Ward Bell of IdeaBlade, Paul Jackson of CM Group Ltd., John Papa of Microsoft, Julian Dominguez of Clarius Consulting, Ted Neveln of Ballard Indexing Services, Glenn Block of Microsoft, Michael Kenyon of IHS, Inc., Terry Young of PEER Group, Jason Beres of Infragistics, Peter Lindes of The Church of Jesus Christ of Latter-day Saints, Mark Tucker of Neudesic, LLC, David Platt of Rolling Thunder Computing, Steve Gentile of Strategic Data Systems, Markus Egger of EPS Software Corp. and CODE Magazine, Ryan Cromwell of Strategic Data Systems, Todd Neal of McKesson Corp, Dipesh Patel of Fidelity Investments, and David Poll of Microsoft.

1: Introduction

Prism provides guidance designed to help you more easily design and build rich, flexible, and easy-to-maintain Windows Presentation Foundation (WPF) desktop applications. Using design patterns such as Model-View-ViewModel (MVVM), Composite View, and Event Aggregator that embody important architectural design principles helps you create a modular application—using loosely coupled components that can evolve independently. These types of applications are known as composite applications.

Composite applications typically feature multiple screens, rich user interaction and data visualization, and that embody significant presentation and business logic. These applications typically interact with multiple back-end systems and services and, using a layered architecture, may be physically deployed across multiple tiers. It is expected that the application will evolve significantly over its lifetime in response to new requirements and business opportunities. In short, these applications are "built to last" and "built for change." Applications that do not demand these characteristics may not benefit from using Prism.

Prism includes reference implementations, QuickStarts, reusable library code (the Prism Library), and extensive documentation. This version of Prism targets the Microsoft .NET Framework 4.5 and includes new guidance around the Model-View-ViewModel (MVVM) pattern, navigation, and the Managed Extensibility Framework (MEF). Because Prism is built on the .NET Framework 4.5 (which includes WPF), familiarity with these technologies is useful for evaluating and adopting Prism.

It should be noted that while Prism is not difficult to learn, developers must be ready and willing to embrace patterns and practices that may be new to them. Management understanding and commitment is crucial, and the project deadline must accommodate an investment of time up front for learning these patterns and practices.

Why Use Prism?

Designing and building rich WPF client applications that are flexible and easy to maintain can be challenging. This section describes some of the common challenges you might encounter when building WPF client applications, and describes how Prism helps you to address those challenges.

Client Application Development Challenges

Typically, developers of client applications face quite a few challenges. Application requirements can change over time. New business opportunities and challenges may present themselves, new technologies may become available, or even ongoing customer feedback during the development cycle may significantly affect the requirements of the application. Therefore, it is important to build the application so that it is flexible and can be easily modified or extended over time. Designing for this type of flexibility can be hard to accomplish. It requires an architecture that allows individual parts of the application to be independently developed and tested and that can be modified or updated later, in isolation, without affecting the rest of the application.

Most enterprise applications are sufficiently complex that they require more than one developer, maybe even a large team of developers that includes user interface (UI) designers and localizers in addition to developers. It can be a significant challenge to decide how to design the application so that multiple developers or subteams can work effectively on different pieces of the application independently, yet ensuring that the pieces come together seamlessly when integrated into the application.

Designing and building applications in a *monolithic* style can lead to an application that is very difficult and inefficient to maintain. In this case, "monolithic" refers to an application in which the components are very tightly coupled and there is no clear separation between them. Typically, applications designed and built this way suffer from problems that make the developer's life hard. It is difficult to add new features to the system or replace existing features, it is difficult to resolve bugs without breaking other portions of the system, and it is difficult to test and deploy. Also, it impacts the ability of developers and designers to work efficiently together.

The Composite Approach

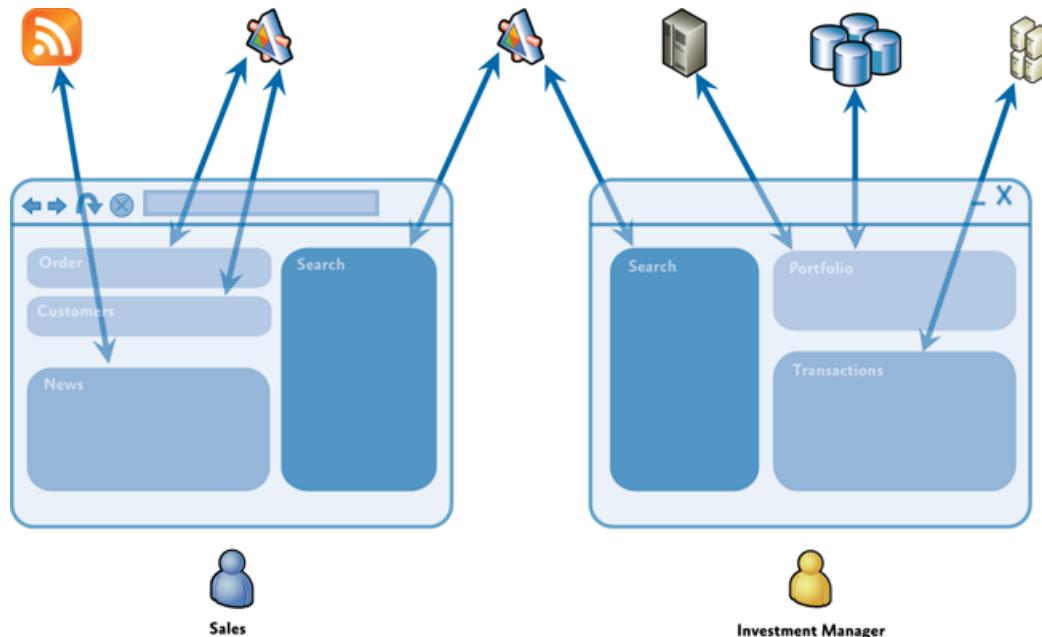
An effective remedy for these challenges is to partition the application into a number of discrete, loosely coupled, semi-independent components that can then be easily integrated together into an application "shell" to form a coherent solution. Applications designed and built this way are often known as composite applications.

Composite applications provide many benefits, including the following:

- They allow modules to be individually developed, tested, and deployed by different individuals or subteams; they also allow them to be modified or extended with new functionality more easily, thereby allowing the application to be more easily extended and maintained. Note that even single-person projects experience benefits in creating more testable and maintainable applications using the composite approach.
- They provide a common shell composed of UI components contributed from various modules that interact in a loosely coupled way. This reduces the contention that arises from multiple developers adding new functionality to the UI, and it promotes a common appearance.
- They promote reuse and a clean separation of concerns between the application's horizontal capabilities, such as logging and authentication, and the vertical capabilities, such as business functionality that is specific to your application. This also allows you to more easily manage the dependencies and interactions between application components.
- They help maintain a separation of roles by allowing different individuals or subteams to focus on a specific task or piece of functionality according to their focus or expertise. In particular, it provides a cleaner separation between the UI and the business logic of the application—this means the UI designer can focus on creating a richer user experience.

Composite applications are highly suited to a range of client application scenarios. For example, a composite application is ideal for creating a rich end-user experience over disparate back-end systems. The following illustration shows an example of this type of a composite application.

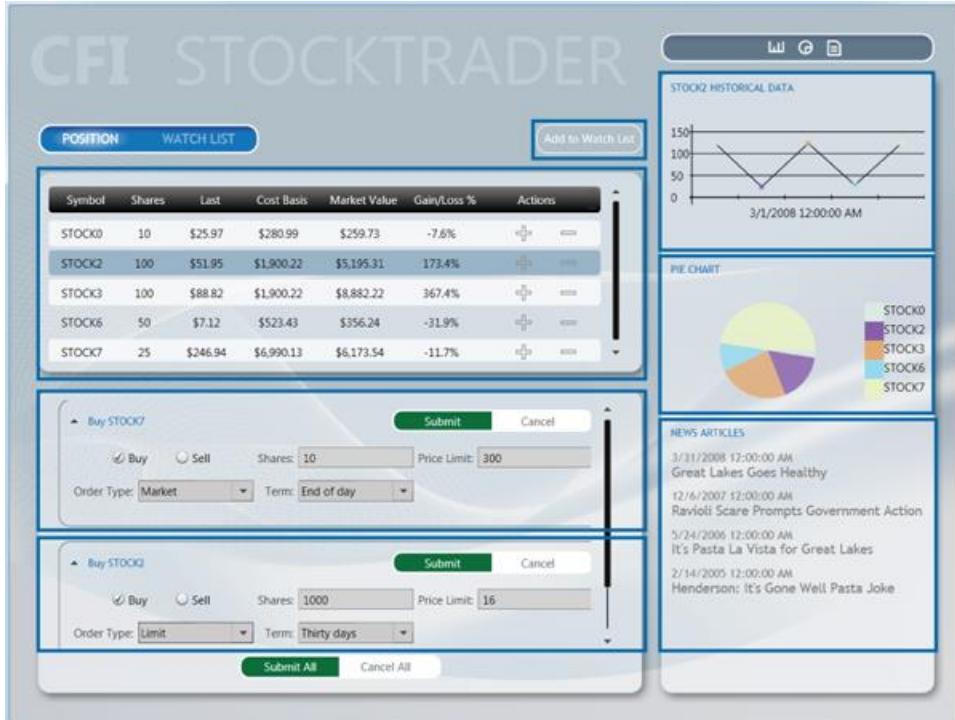
Composite application with multiple back-end systems



In this type of application, the user can be presented with a rich and flexible user experience that provides a task-oriented focus over functionality that spans multiple back-end systems, services, and data stores, where each is represented by one or more dedicated modules. The clean separation between the application logic and the UI allows the application to provide a consistent and differentiated appearance across all constituent modules.

Additionally, a composite application can be useful when there are independently evolving components in the UI that heavily integrate with each other and that are often maintained by separate teams. The following illustration shows a screen shot of this type of application. Each of the areas highlighted represent independent components that are composed into the UI.

Stock Trader Reference Implementation composite application



In this case, the composite application allows the UI to be dynamic composed. This delivers a flexible user experience. For example, it can allow new functionality to be dynamically added to the application at run time, which enables rich end-user customization and extensibility.

Challenges Not Addressed by Prism

Although Prism helps you to address many of the challenges you might face when building WPF applications, there are many other challenges that you might face, depending on your application scenario and requirements. For example, Prism does not directly address the following topics:

- Occasional connectivity and data synchronization
- Service and messaging infrastructure design
- Authentication and authorization
- Application performance
- Application versioning
- Error handling and fault tolerance

Prerequisites

Prism assumes you have hands-on experience with WPF. There are a few important concepts that Prism uses heavily, and you should become familiar with them. They include the following:

- **XAML (Extensible Application Markup Language).** The language to declaratively define and initialize the user interface in WPF applications.

- **Data binding.** This is how UI elements are connected to components and data in WPF.
 - **Resources.** These are how styles, data templates, and control templates are created and managed in WPF.
 - **Commands.** These are how user gestures and input are connected to controls.
 - **User controls.** These are components that provide custom behavior or custom appearance.
 - **Dependency properties.** These are extensions to the common language runtime (CLR) property system to enable property setting and monitoring in support of data binding, routed commands, and events.
 - **Behaviors.** Behaviors are objects that encapsulate interactive functionality that can be easily applied to controls in the user interface.
-

An Overview of Prism

Architectural Goals

The guidance is designed to help architects and developers achieve the following objectives:

- Create an application from modules that can be built, assembled and, optionally, deployed by independent teams using WPF.
 - Minimize cross-team dependencies and allow teams to specialize in different areas, such as user interface (UI) design, business logic implementation, and infrastructure code development.
 - Use an architecture that promotes reusability across independent teams.
 - Increase the quality of applications by abstracting common services that are available to all the teams.
 - Incrementally integrate new capabilities.
-

Prism Design Goals

Prism was designed to help you design and build rich, flexible, and easy-to-maintain WPF applications. The Prism Library implements design patterns that embody important architectural design principles, such as separation of concerns and loose coupling. Using the design patterns and capabilities provided by the Prism Library, you can design and build applications using loosely coupled components that can evolve independently but that can be easily and seamlessly integrated into the overall application.

Prism is designed around the core architectural design principles of separation of concerns and loose coupling. This allows Prism to provide many benefits, including the following:

- **Reuse.** Prism promotes reuse by allowing components and services to be easily developed, tested and integrated into one or more applications. Reuse can be achieved at the component level through the reuse of unit-tested components that can be easily discovered and integrated

at run time through dependency injection, and at the application level through the use of modules that encapsulate application-level capabilities that can be reused across applications.

- **Extensibility.** Prism helps to create applications that are easy to extend by managing component dependencies, allowing components to be more easily integrated or replaced with alternative implementations at run time, and by providing the ability to decompose an application into modules that can be independently updated and deployed. Many of the components in the Prism Library itself can also be extended or replaced.
- **Flexibility.** Prism helps to create flexible applications by allowing them to be more easily updated as new capabilities are developed and integrated. Prism also allows WPF applications to be developed using common services and components, allowing the application to be deployed and consumed in the most appropriate way. It also allows applications to provide different experiences based on role or configuration.
- **Team Development.** Prism promotes team development by allowing separate teams to develop and even deploy different parts of the application independently. Prism helps to minimize cross-team dependencies and allows teams to focus on different functional areas (such as UI design, business logic implementation, and infrastructure code development), or on different business-level functional areas (such as profile, sales, inventory, or logistics).
- **Quality.** Prism can help to increase the quality of applications by allowing common services and components to be fully tested and made available to the development teams. In addition, by providing fully tested implementations of common design patterns, and the guidance needed to fully leverage them, Prism allows development teams to focus on their application requirements instead of implementing and testing infrastructure code.

It is important to note that Prism was designed so that you can use any of Prism's capabilities and design patterns individually, or all together, depending on your requirements and your application scenario. Prism was designed so that it could be incrementally adopted, allowing you to use the capabilities and design patterns that make sense for your particular application without requiring major structural changes.

Finally, because software testing should be considered a first-class development activity and tightly integrated into the development process, Prism provides extensive support for various types of software testing, thereby allowing you to design and build applications that are easy to test. Prism itself was developed with testing in mind. It was developed to meet multiple strict quality gates to ensure that it meets Microsoft security standards and that it will function correctly on multiple operating systems, with multiple versions of Visual Studio, and with multiple programming languages. Unit tests were run after each check-in. In addition, the Prism library was tested against several additional quality gates, as listed in the following table.

Test	Description
Acceptance Testing	Validates the application functionality using user scenarios to drive the test requirements. Tests can be executed manually or automated.
Application Building Exercises	Team members build applications consuming the deliverable software.
Black Box Testing	Manual acceptance tests perform from the user point of view.
Cross Platform Testing	All automated tests are run on multiple platforms.
Globalization Testing	All automated tests are run on multiple languages.
Performance Testing	Measures how fast a particular aspect of a system performs under-load.
Security Review	Internal Microsoft security audit standards that cover thread models, identifying attack factors and running the code through security analysis tools.
Stress Testing	Measures stability of the system under extreme loads; specifically looking to drive out issues like memory leaks and threading issues.
White Box Testing	In-depth source code analysis validating the coding standards, structure and how it maps to the overall architecture.

The Prism Library source code includes unit and UI automation tests, as shown in the following table. You can use these as an educational resource, or you can run the tests against the Prism Library itself. This allows you to customize, re-compile, test and deploy a modified version of the Prism Library using similar quality gates as the Prism team.

Test	Description
UI Automation Tests	Limited range of acceptance testing; driving the application from the user perspective
Unit Tests	Validates the implementation of a class

Prism Key Concepts

Prism provides capabilities and design patterns that may be unfamiliar to you, especially if you're new to design patterns and composite application development. This section provides a brief overview of the main concepts behind Prism and defines some of the terminology that you will see used throughout the documentation and code.

- **Modules.** Modules are packages of functionality that can be independently developed, tested, and (optionally) deployed. In many situations, modules are developed and maintained by separate teams. A typical Prism application is built from multiple modules. Modules can be used to represent specific business-related functionality (for example, profile management) and

encapsulate all the views, services, and data models required to implement that functionality. Modules can also be used to encapsulate common application infrastructure or services (for example, logging and exception management services) that can be reused across multiple applications.

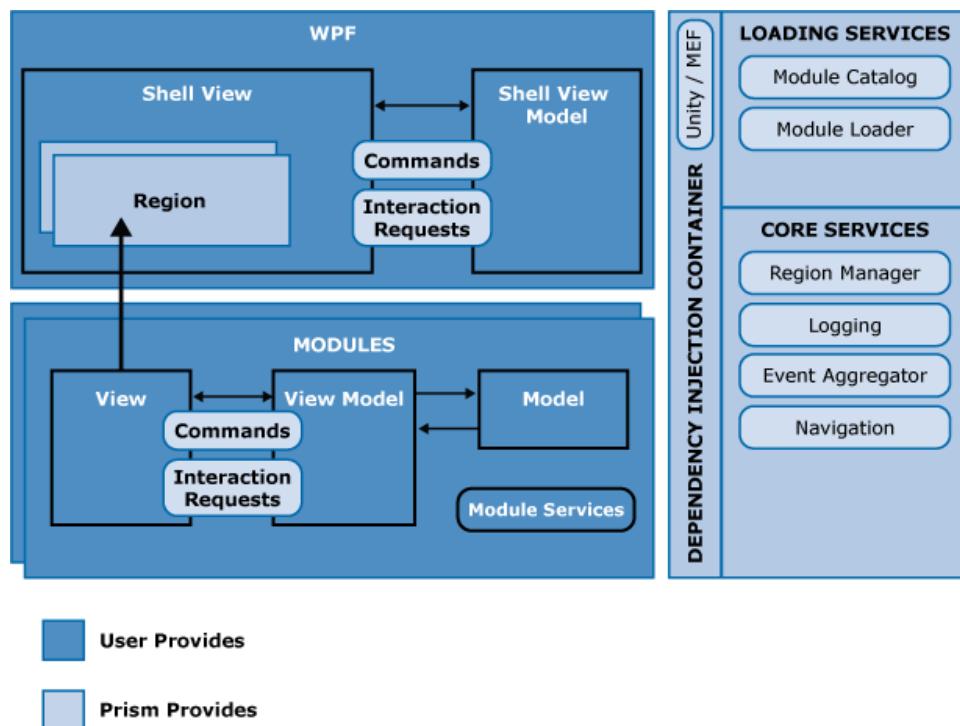
- **Module catalog.** In a composite application, modules must be discovered and loaded at run time by the host application. In Prism, a module catalog is used to specify which modules to load, when they are loaded, and in what order. The module catalog is used by the **ModuleManager** and **ModuleLoader** components, which are responsible for downloading the modules if they are remote, loading the module's assemblies into the application domain, and for initializing the module. Prism allows the module catalog to be specified in different ways, including programmatically using code, declaratively using XAML, or using a configuration file. You can also implement a custom module catalog if you need to.
- **Shell.** The shell is the host application into which modules are loaded. The shell defines the overall layout and structure of the application, but it is typically unaware of the exact modules that it will host. It usually implements common application services and infrastructure, but most of the application's functionality and content is implemented within the modules. The shell also provides the top-level window or visual element that will then host the different UI components provided by the loaded modules.
- **Views.** Views are UI controls that encapsulate the UI for a particular feature or functional area of the application. Views are used in conjunction with the MVVM pattern, which is used to provide a clean separation of concerns between the UI and the application's presentation logic and data. Views are used to encapsulate the UI and define user interaction behavior, thereby allowing the view to be updated or replaced independently of the underlying application functionality. Views use data binding to interact with view model classes.
- **View models.** View models are classes that encapsulate the application's presentation logic and state. They are part of the MVVM pattern. View models encapsulate much of the application's functionality. View models define properties, commands, and events, to which controls in the view can data-bind.
- **Models.** Model classes encapsulate the application data and business logic. They are used as part of the MVVM pattern. Models encapsulate data and any associated validation and business rules to ensure data consistency and integrity.
- **Commands.** Commands are used to encapsulate application functionality in a way that allows them to be defined and tested independently of the application's UI. They can be defined as command objects or as command methods in the view model. Prism provides the **DelegateCommand** class and the **CompositeCommand** class. The latter is used to represent a collection of commands which are all invoked together.
- **Regions.** Regions are logical placeholders defined within the application's UI (in the shell or within views) into which views are displayed. Regions allow the layout of the application's UI to

be updated without requiring changes to the application logic. Many common controls can be used as a region, allowing views to be automatically displayed within controls, such as a **ContentControl**, **ItemsControl**, **ListBox**, or **TabControl**. Views can be displayed within a region programmatically or automatically. Prism also provides support for implementing navigation with regions. Regions can be located by other components through the **RegionManager** component, which uses **RegionAdapter** and **RegionBehavior** components to coordinate the display of views within specific regions.

- **Navigation.** Navigation is defined as the process by which the application coordinates changes to its UI as a result of the user's interaction with the application or internal application state changes. Prism supports two styles of navigation: state-based navigation, where the state of an existing view is updated to implement simple navigation scenarios, and view-switching navigation, where new views are created and old views replaced within the application's UI. View-switching navigation uses a Uniform Resource Identifier (URI)-based navigation mechanism in conjunction with Prism regions to allow flexible navigation schemes to be implemented.
- **EventAggregator.** Components in a composite application often need to communicate with other components and services in the application in a loosely coupled way. To support this, Prism provides the **EventAggregator** component, which implements a pub-sub event mechanism, thereby allowing components to publish events and other components to subscribe to those events without either of them requiring a reference to the other. The **EventAggregator** is often used to allow components defined in different modules to communicate with each other.
- **Dependency injection container.** The Dependency Injection (DI) pattern is used throughout Prism to allow the dependencies between components to be managed. Dependency injection allows component dependencies to be fulfilled at run time, and it supports extensibility and testability. Prism is designed to work with Unity or MEF, or with any other dependency injection containers via the **ServiceLocator**.
- **Services.** Services are components that encapsulate non-UI related functionality, such as logging, exception management, and data access. Services can be defined by the application or within a module. Services are often registered with the dependency injection container so that they can be located or constructed as required and used by other components that depend on them.
- **Controllers.** Controllers are classes that are used to coordinate the construction and initialization of views that are to be displayed in a region within the application's UI. Controllers encapsulate the presentation logic that determines which views are to be displayed. The controller will use Prism's view-switching navigation mechanism, which provides an extensible URI-based navigation mechanism to coordinate the construction and placement of views within regions. The Application Controller pattern defines an abstraction that maps to this responsibility.

- **Bootstrapper.** The **Bootstrapper** component is used by the application to initialize the various Prism components and services. It is used to initialize the dependency injection container to register any application-level components and services with it. It is also used to configure and initialize the module catalog and the shell's view and view model or presenter.

Prism is designed so that you can use any of the preceding capabilities and design patterns individually, or all together, depending on your requirements and your application scenario. You can use the MVVM pattern, modularity, regions, commands, or events in any combination without having to adopt all of them. Of course, if you want to take full advantage of the benefits that separation of concerns and loose coupling offers, you will typically use many of Prism's capabilities and design patterns in conjunction with each other. The following illustration shows a typical Prism application architecture and shows how all the various capabilities of Prism can work together within a multi-module composite application.



Typical composite application architecture with the Prism Library

Most Prism applications consist of a shell application that defines regions for displaying top-level views and shared services that can be accessed by the loaded modules. The shell defines a suitable catalog to specify which modules are to be loaded at startup time , as appropriate. A dependency injection container is also defined, which allows component dependencies to be fulfilled at run time. Shared services and components are registered with the container by the **Bootstrapper** when the application starts.

Individual modules encapsulate a portion of the overall application's functionality and, using a separated presentation pattern such as MVVM, define views, view models, models, and service components. When the modules are loaded, views defined within the modules are displayed within the regions defined by the shell. After initialization completes, the user then navigates within the application using

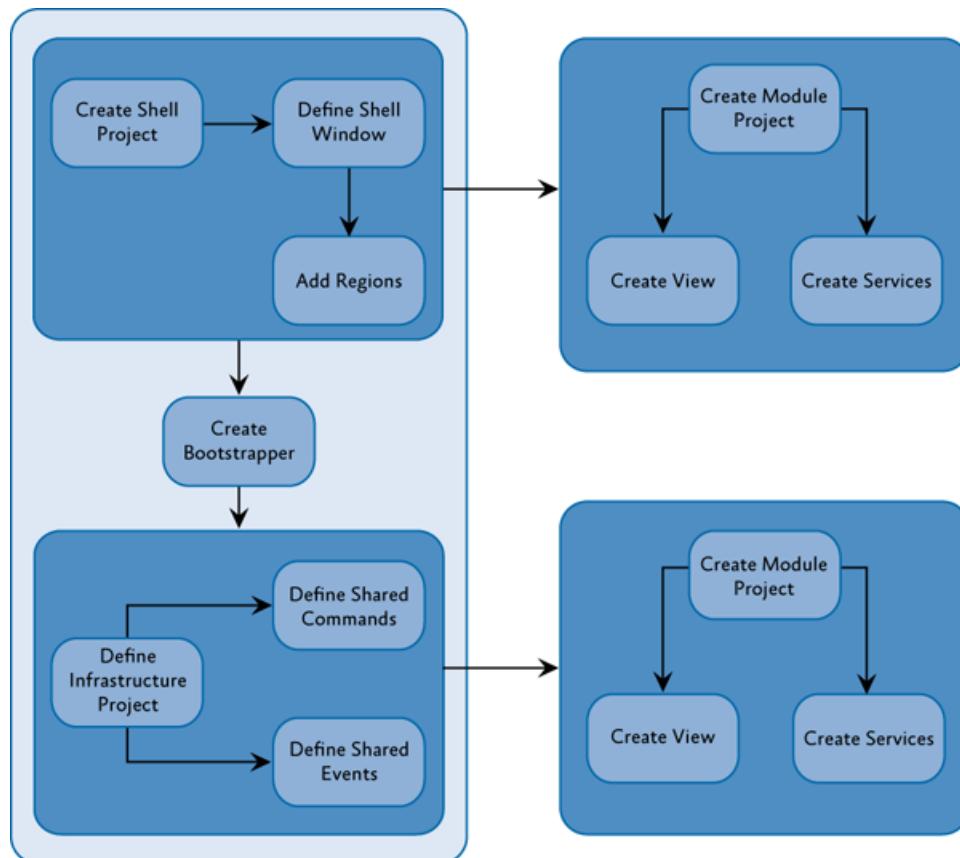
state-based or view-switching navigation to coordinate the visual update or display of new views within the application's regions.

Using Prism

Now that you've seen the major capabilities and design patterns that Prism supports, it's time to see how easily you can start to use Prism when developing a new application. This section provides an overview of the first few steps required to create a basic Prism application. You can extend this basic application to leverage the additional capabilities and design patterns provided by Prism, as required by your scenario.

Note: Although the Prism Library can be easily used to build new composite WPF applications, you can also use Prism with existing applications that want to take advantage of one or more Prism capabilities or design patterns.

A Prism application typically consists of a shell project and multiple module projects. The following illustration shows common activities needed when developing a composite application using the Prism Library.



Activities for creating a composite application

A typical Prism application leverages most or all of the Prism capabilities and design patterns described earlier to be able to fully realize the benefits of the loose coupling and separation of concerns

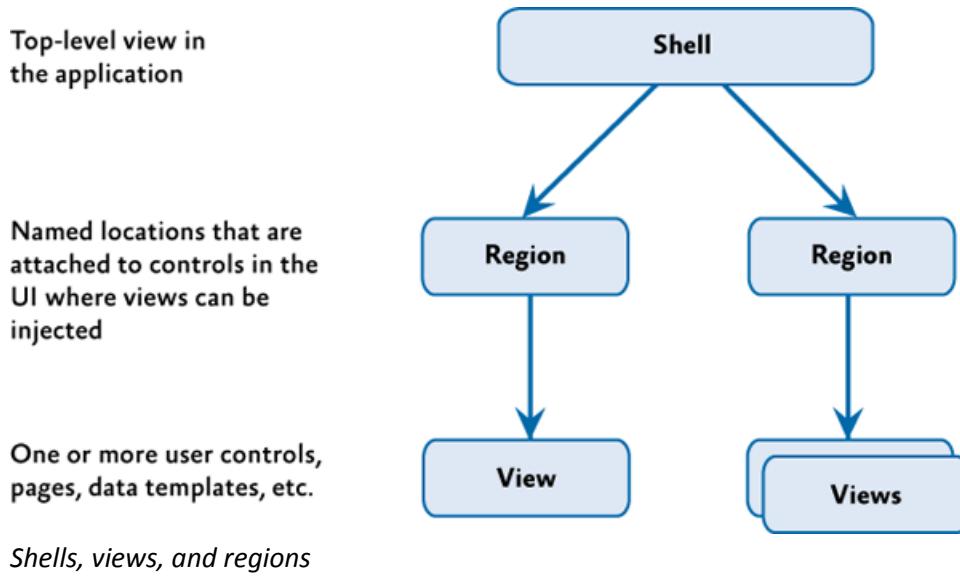
architectural design principles. However, for this example, the steps required to create a basic Prism application that consists of a single module that defines a single view are described.

Prism Library References

Most of your projects will need to reference the Prism Library assemblies. Prism provides signed binaries through NuGet packages so that you can use the Visual Studio **Manage NuGet Packages** dialog box to add references to them. You can also include the Prism Library projects in your solution and then use project references to them. The latter has the advantage of being able to use features like Go To Definition to step down into the Prism types, as well as being able to build and sign the Prism Library assemblies with your own strong name or certificate as part of your build process.

Define the Shell

The application shell provides the basic layout for the application. This layout is defined using regions that modules can use to place views. Views, like shells, can use regions to define discoverable areas that content can be added to, as shown in the following illustration. Shells typically set the appearance for the entire application and contain the styles that are used throughout the application.



Create the Bootstrapper

The bootstrapper is the glue that connects the application with the Prism Library services and the Unity or MEF containers. Each application creates an application-specific bootstrapper, which typically inherits from either **UnityBootstrapper** or **MefBootstrapper**, as shown in the following illustration. You will need to decide the approach you want to use to populate the module catalog. Minimally, each application will provide a module catalog and a shell.

By default, the bootstrapper logs events using the .NET Framework **Trace** class. Most applications will want to supply their own logging services, such as Enterprise Library logging. Applications can supply their logging service in their bootstrapper.

By default, the **UnityBootstrapper** and **MefBootstrapper** enable the Prism Library services. These can be disabled or replaced in your application-specific bootstrapper.

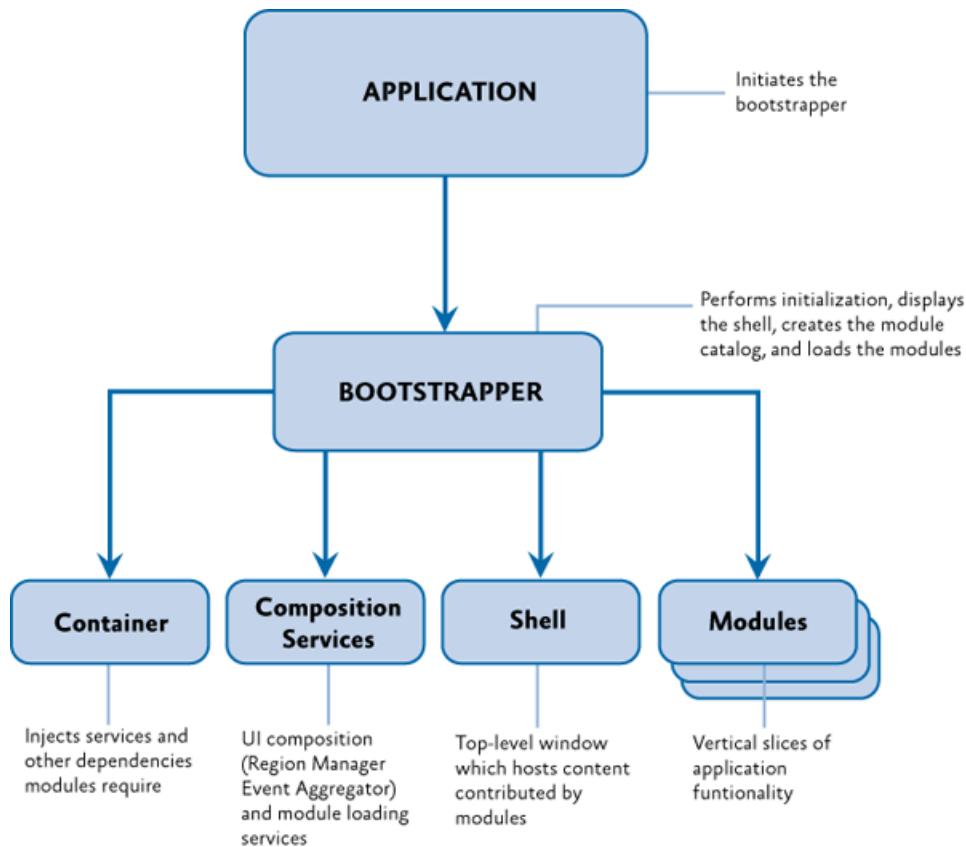


Diagram demonstrating connecting to the Prism Library

Create the Module

The module contains the views and services specific to a piece of the application's functionality. Frequently, these are contained in separate assemblies and developed by separate teams. A module is denoted by a class that implements the **IModule** interface. These modules, during initialization, register their views and services and may add one or more views to the shell. Depending on your module discovery approach, you may need to apply attributes to your module classes or define dependencies between your modules.

Add a Module View to the Shell

Modules take advantage of the shell's regions for placing content. During initialization, modules use the **RegionManager** to locate regions in the shell and add one or more views to those regions or register one or more view types to be created within those regions. The **RegionManager** is responsible for keeping track of regions throughout the application and is a core service initialized from the bootstrapper.

The remaining topics in this guide provide details about Prism key concepts.

Exploring Prism

Prism consists of the following:

- [Prism Library source code](#). The source code for the Prism Library assemblies, including the core Prism functionality, plus Unity and MEF extensions, which provide additional components for using Prism with the [Unity Application Block](#) (Unity) and the [Managed Extensibility Framework](#). The source code also includes Prism.PubSubEvents and Prism.Mvvm assemblies.
- [Prism binary assemblies](#). Signed binary versions of the Prism Library assemblies. These assemblies can be downloaded from NuGet by searching for Prism, Prism.Composition, Prism.PubSubEvents, and Prism.Mvvm, Prism.Interactivity, Prism.UnityExtensions, and Prism.MefExtensions. These NuGet packages will load dependencies such as the [Unity Application Block](#) and the [Service Locator](#).

The Prism NuGet package will download the Prism.Composition, Prism.PubSubEvents, Prism.Mvvm, Prism.Interactivity, Prism.PubSubEvents, and Prism.Mvvm NuGet packages.

- [Code samples](#). Prism includes a reference implementation sample and QuickStart samples. The Stock Trader Reference Implementation is a comprehensive sample application that illustrates how Prism can be used to implement real-world application scenarios. The reference implementation is intentionally incomplete, but they illustrate how many of the patterns in Prism can work together within a single application. The QuickStart samples include several small, focused sample applications that illustrate the MVVM pattern, navigation, UI composition, modularity, commanding, event aggregation, and interactivity.
- [Documentation](#). The Prism 5.0 documentation provides an overview of the goals and concepts behind Prism and detailed guidance on using each of the capabilities and design patterns provided by Prism. The next section provides an overview of the topics covered.

Exploring the Documentation

The Prism documentation spans a wide range of topics, including an overview of common development challenges and the composite application approach, an overview of the Prism Library and the design patterns that it implements, as well as step-by-step instructions for using the Prism Library during development. The documentation is intended to appeal to a broad technical audience to help the reader to understand and use Prism within their own applications. The documentation includes the following:

- [Initializing Applications](#). This topic discusses what needs to happen to get a modular Prism application up and running.
- [Managing Dependencies Between Components](#). Applications based on the Prism Library rely on a dependency injection container. Although Prism has the ability to work with nearly any dependency injection container, the Prism Library provides two default options for

dependency injection containers: Unity or MEF. This topic discusses the different capabilities and what you need to think about when working with a dependency injection container.

- [Modular Application Development](#). This topic discusses the core concepts, key decisions, and core scenarios when you create a modular client application with Prism.
- [Implementing the MVVM Pattern](#). Using the MVVM pattern, you separate the UI of your application and the underlying presentation and business logic into three separate classes: the view, model, and view model. This topic discusses the core concepts behind the MVVM pattern and describes how to implement it in your application using Prism.
- [Advanced MVVM Scenarios](#). This topic provides guidance on implementing more advanced scenarios using the MVVM pattern, including how to implement composite commands (commands that represent a group of commands), and how to handle asynchronous web service and user interactions. This topic also provides guidance on using a dependency injection container, such as Unity or MEF, to handle the construction and wire-up of the MVVM classes.
- [Composing the User Interface](#). Regions are placeholders that allow a developer to specify where views will be displayed in the application's UI. In Prism, there are two approaches to displaying views in a region: view discovery and view injection. This topic describes how to work with regions and the UI. It also includes information for UI designers to understand composite applications.
- [Navigation](#). Navigation is the process by which the application coordinates changes to its UI as a result of the user's interaction with the application or internal application state changes. This topic provides guidance on implementing state-based navigation, where the state of the UI in a view is updated to reflect navigation, and view-switching navigation, where a new view is created and displayed in a region.
- [Communicating Between Loosely Coupled Components](#). This topic discusses the various options for communicating between components in different modules, using commanding, the EventAggregator, region context, and shared services.
- [Deploying Applications](#). This topic addresses deployment considerations for Prism WPF applications.
- [Glossary](#). This appendix provides a concise summary of the terms, concepts, design patterns and capabilities provided by Prism.
- [Patterns in the Prism Library](#). This appendix describes the software design patterns applied in the Prism Library and the Stock Trader RI. This topic primarily targets architects and developers wanting to familiarize themselves with the patterns used to address the challenges in building composite applications.
- [Prism Library](#). This appendix provides an overview of the Prism Library for WPF.

- [Upgrading from Prism Library 4.1](#). This appendix discusses what you need to know if you are upgrading from previous versions of Prism.
 - [Extending the Prism Library](#). This appendix discusses how you can extend Prism modularity, behaviors, and navigation.
 - [Code Samples](#). Prism includes the source code for several samples that demonstrate key concepts. For more information, see the next section, "Code Samples Using the Prism Library for WPF."
 - [Getting Started Using the Prism Library Hands-On Lab](#). This hands-on labs demonstrates building a simple composite application, step-by-step, in WPF. It primarily targets developers who want to understand the basic concepts of the Prism Library.
 - [Publishing and Updating Applications Using the Prism Library Hands-On Lab](#). This hands-on lab walks you through the process of publishing and updating a Prism WPF application with ClickOnce.
-

Exploring the Code Samples

The code samples illustrate specific Prism-related concepts. The samples are an ideal starting point if you want to gain an understanding of a key concept and you are comfortable learning new techniques by examining source code. Prism includes the following:

- [Stock Trader Reference Implementation](#). The Stock Trader RI is a composite application that demonstrates an implementation of the baseline architecture using the Prism Library.
- [Modularity QuickStarts](#). These QuickStarts demonstrate how to build WPF applications composed of modules. The modules can be statically loaded, when the shell contains a reference to the module's assembly, or dynamically loaded, when modules are dynamically discovered and loaded at run time. The QuickStarts also demonstrate using the Unity container and MEF.
- [Interactivity QuickStart](#). This QuickStart demonstrates how to create a view and view model that work together when the view model needs to interact with the user or a user gesture needs to raise an event that invokes a command. In each of these scenarios the view model should not need to know about the view. The first scenario is handled by using **InteractionRequests** and **InteractionRequestTriggers**. The second scenario is handled by **InvokeCommandAction**.
- [MVVM QuickStart](#). This QuickStart demonstrates how to build an application that implements the MVVM presentation pattern, showing some of the more common challenges that developers can face, such as validation, UI interactions, and data templates.
- [Commanding QuickStart](#). This QuickStart demonstrates how to build a WPF UI that uses commands provided by the Prism.Mvvm Library to handle UI actions in a decoupled way.

- [UI Composition QuickStart](#). This QuickStart demonstrates how to build WPF UIs composed of different views that are dynamically loaded into regions and that interact with each other in a decoupled way. It illustrates how to use both the view discovery and view injection approaches for UI composition.
 - [State-Based Navigation QuickStart](#). This QuickStart demonstrates an approach to define the navigation of a simple application. The approach used in this QuickStart uses the WPF Visual State Manager (VSM) to define the different states that the application has and defines animations for both the states and the transitions between states.
 - [View-Switching Navigation QuickStart](#). This QuickStart demonstrates how to use the Prism Region Navigation API. The QuickStart shows multiple navigation scenarios, including navigating to a view in a region, navigating to a view in a region contained in another view (nested navigation), navigation journal support, just-in-time view creation, passing contextual information when navigating to a view, views and view models participating in navigation, and using navigation as part of an application built through modularity and UI composition.
 - [Event Aggregation QuickStart](#). This QuickStart demonstrates how to build a WPF application that uses the Event Aggregator service. This service enables you to establish loosely coupled communications between components in your application.
-

More Information

Prism assumes you have hands-on experience with WPF. If you need general information about WPF , see the following resources:

- [Windows Presentation Foundation](#) on MSDN.
 - MacDonald, Matthew. *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4*, Apress, 2010.
 - Nathan, Adam. *WPF 4 Unleashed*. Sams Publishing, 2010.
-

Community

Prism's community sites are:

- Prism: <http://www.codeplex.com/Prism>
 - PubSubEvents (Event Aggregator): <http://www.codeplex.com/pnpPubSub>
 - MVVM (Model-View-ViewModel): <http://www.codeplex.com/pnpMvvm>
-

On these community sites, you can post questions, provide feedback, or connect with other users for sharing ideas. Community members can also help Microsoft plan and test future offerings and download additional content, such as extensions and training material.

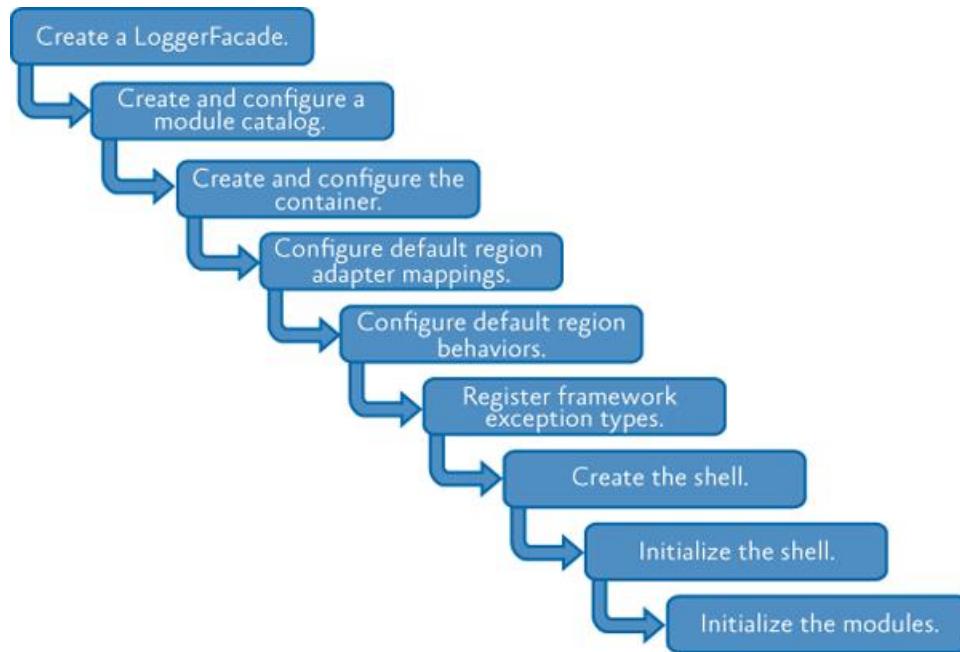
2: Initializing Applications

This topic addresses what needs to happen to get a Prism for WPF application up and running. A Prism application requires registration and configuration during the application startup process—this is known as bootstrapping the application. The Prism bootstrapping process includes creating and configuring a module catalog, creating a dependency injection container such as Unity, configuring default region adapter for UI composition, creating and initializing the shell view, and initializing modules.

What Is a Bootstrapper?

A bootstrapper is a class that is responsible for the initialization of an application built using the Prism Library. By using a bootstrapper, you have more control of how the Prism Library components are wired up to your application.

The Prism Library includes a default abstract **Bootstrapper** base class that can be specialized for use with any container. Many of the methods on the bootstrapper classes are virtual methods. You can override these methods as appropriate in your own custom bootstrapper implementation.



Basic stages of the bootstrapping process

The Prism Library provides some additional base classes, derived from **Bootstrapper**, that have default implementations that are appropriate for most applications. The only stages left for your application bootstrapper to implement are creating and initializing the shell.

Dependency Injection

Applications built with the Prism Library rely on dependency injection provided by a container. The library provides assemblies that work with the Unity Application Block (Unity) or Managed Extensibility

Framework (MEF), and it allows you to use other dependency injection containers. Part of the bootstrapping process is to configure this container and register types with the container.

The Prism Library includes the **UnityBootstrapper** and **MefBootstrapper** classes, which implement most of the functionality necessary to use either Unity or MEF as the dependency injection container in your application. In addition to the stages shown in the previous illustration, each bootstrapper adds some steps specific to its container.

Creating the Shell

In a traditional Windows Presentation Foundation (WPF) application, a startup Uniform Resource Identifier (URI) is specified in the App.xaml file that launches the main window.

In an application created with the Prism Library, it is the bootstrapper's responsibility to create the shell or the main window. This is because the shell relies on services, such as the Region Manager, that need to be registered before the shell can be displayed.

Key Decisions

After you decide to use the Prism Library in your application, there are a number of additional decisions that need to be made:

- You will need to decide whether you are using MEF, Unity, or another container for your dependency injection container. This will determine which provided bootstrapper class you should use and whether you need to create a bootstrapper for another container.
- You should think about the application-specific services you want in your application. These will need to be registered with the container.
- Determine whether the built-in logging service is adequate for your needs or if you need to create another logging service.
- Determine how modules will be discovered by the application: via explicit code declarations, code attributes on the modules discovered via directory scanning, configuration, or XAML.

The rest of this topic provides more details.

Core Scenarios

Creating a startup sequence is an important part of building your Prism application. This section describes how to create a bootstrapper and customize it to create the shell, configure the dependency injection container, register application level services, and how to load and initialize the modules.

Creating a Bootstrapper for Your Application

If you choose to use either Unity or MEF as your dependency injection container, creating a simple bootstrapper for your application is easy. You will need to create a new class that derives from either **MefBootstrapper** or **UnityBootstrapper**. Then, implement the **CreateShell** method. Optionally, you may override the **InitializeShell** method for shell specific initialization.

Implementing the CreateShell Method

The **CreateShell** method allows a developer to specify the top-level window for a Prism application. The shell is usually the **MainWindow** or **MainPage**. Implement this method by returning an instance of your application's shell class. In a Prism application, you can create the shell object, or resolve it from the container, depending on your application's requirements.

An example of using the **ServiceLocator** to resolve the shell object is shown in the following code example.

C#

```
protected override DependencyObject CreateShell()
{
    return ServiceLocator.Current.GetInstance<Shell>();
}
```

Note: You will often see the **ServiceLocator** being used to resolve instances of types instead of the specific dependency injection container. The **ServiceLocator** is implemented by calling the container, so it makes a good choice for container agnostic code. You can also directly reference and use the container instead of the **ServiceLocator**.

Implementing the InitializeShell Method

After you create a shell, you may need to run initialization steps to ensure that the shell is ready to be displayed. For WPF applications, you will create the shell application object and set it as the application's main window, as shown here (from the Modularity QuickStarts for WPF).

C#

```
protected override void InitializeShell()
{
    Application.Current.MainWindow = Shell;
    Application.Current.MainWindow.Show();
}
```

The base implementation of **InitializeShell** does nothing. It is safe to not call the base class implementation.

Creating and Configuring the Module Catalog

If you are building a module application, you will need to create and configure a module catalog. Prism uses a concrete **IModuleCatalog** instance to keep track of what modules are available to the application, which modules may need to be downloaded, and where the modules reside.

The **Bootstrapper** provides a protected **ModuleCatalog** property to reference the catalog as well as a base implementation of the virtual **CreateModuleCatalog** method. The base implementation returns a new **ModuleCatalog**; however, this method can be overridden to provide a different **IModuleCatalog** instance instead, as shown in the following code from the **QuickStartBootstrapper** in the Modularity with MEF for WPF QuickStart.

C#

```
protected override IModuleCatalog CreateModuleCatalog()
{
    // When using MEF, the existing Prism ModuleCatalog is still
    // the place to configure modules via configuration files.
    return new ConfigurationModuleCatalog()
}
```

In both the **UnityBootstrapper** and **MefBootstrapper** classes, the **Run** method calls the **CreateModuleCatalog** method and then sets the class's **ModuleCatalog** property using the returned value. If you override this method, it is not necessary to call the base class's implementation because you will replace the provided functionality. For more information about modularity, see "[Modular Application Development](#)."

Creating and Configuring the Container

Containers play a key role in an application created with the Prism Library. Both the Prism Library and the applications built on top of it depend on a container for injecting required dependencies and services. During the container configuration phase, several core services are registered. In addition to these core services, you may have application-specific services that provide additional functionality as it relates to composition.

Core Services

The following table lists the core non-application specific services in the Prism Library.

Service interface	Description
IModuleManager	Defines the interface for the service that will retrieve and initialize the application's modules.
IModuleCatalog	Contains the metadata about the modules in the application. The Prism Library provides several different catalogs.
IModuleInitializer	Initializes the modules.
IRegionManager	Registers and retrieves regions, which are visual containers for layout.
IEventAggregator	A collection of events that is loosely coupled between the publisher and the subscriber.
ILoggerFacade	A wrapper for a logging mechanism, so you can choose your own logging mechanism. The Stock Trader Reference Implementation (Stock Trader RI) uses the Enterprise Library Logging Application Block, via the EnterpriseLibraryLoggerAdapter class, as an example of how you can use your own logger. The logging service is registered with the container by the bootstrapper's Run method, using the value returned by the CreateLogger method. Registering another logger with the container will not work; instead override the CreateLogger method on the bootstrapper.
IServiceLocator	Allows the Prism Library to access the container. If you want to customize or extend the library, this may be useful.

Application-Specific Services

The following table lists the application-specific services used in the Stock Trader RI. This can be used as an example to understand the types of services your application may provide.

Services in the Stock Trader RI	Description
IMarketFeedService	Provides real-time (mocked) market data. The PositionSummaryViewModel updates the position screen based on notifications it receives from this service.
IMarketHistoryService	Provides historical market data used for displaying the trend line for the selected fund.
IAccountPositionService	Provides the list of funds in the portfolio.
IOrdersService	Persists submitted buy/sell orders.
INewsFeedService	Provides a list of news items for the selected fund.
IWatchListService	Handles when new watch items are added to the watch list.

There are two **Bootstrapper**-derived classes available in Prism, the **UnityBootstrapper** and the **MefBootstrapper**. Creating and configuring the different containers involve similar concepts that are implemented differently.

Creating and Configuring the Container in the UnityBootstrapper

The **UnityBootstrapper** class's **CreateContainer** method simply creates and returns a new instance of a **UnityContainer**. In most cases, you will not need to change this functionality; however, the method is virtual, thereby allowing that flexibility.

After the container is created, it probably needs to be configured for your application. The **ConfigureContainer** implementation in the **UnityBootstrapper** registers a number of core Prism services by default, as shown here.

Note: An example of this is when a module registers module-level services in its **Initialize** method.

C#

```
// UnityBootstrapper.cs
protected virtual void ConfigureContainer()
{
    ...
    if (useDefaultConfiguration)
    {
        RegisterTypeIfMissing(typeof(IServiceLocator),
        typeof(UnityServiceLocatorAdapter), true);
        RegisterTypeIfMissing(typeof(IModuleInitializer), typeof(ModuleInitializer),
        true);
        RegisterTypeIfMissing(typeof(IModuleManager), typeof(ModuleManager), true);
        RegisterTypeIfMissing(typeof(RegionAdapterMappings),
        typeof(RegionAdapterMappings), true);
        RegisterTypeIfMissing(typeof(IRegionManager), typeof(RegionManager), true);
    }
}
```

```

        RegisterTypeIfMissing(typeof(IEventAggregator), typeof(EventAggregator), true);
        RegisterTypeIfMissing(typeof(IRegionViewRegistry), typeof(RegionViewRegistry),
true);
        RegisterTypeIfMissing(typeof(IRegionBehaviorFactory),
typeof(RegionBehaviorFactory), true);
        RegisterTypeIfMissing(typeof(IRegionNavigationJournalEntry),
typeof(RegionNavigationJournalEntry), false);
        RegisterTypeIfMissing(typeof(IRegionNavigationJournal),
typeof(RegionNavigationJournal), false);
        RegisterTypeIfMissing(typeof(IRegionNavigationService),
typeof(RegionNavigationService), false);
        RegisterTypeIfMissing(typeof(IRegionNavigationContentLoader),
typeof(UnityRegionNavigationContentLoader), true);

    }
}

```

The bootstrapper's **RegisterTypeIfMissing** method determines whether a service has already been registered—it will not register it twice. This allows you to override the default registration through configuration. You can also turn off registering any services by default; to do this, use the overloaded **Bootstrapper.Run** method passing in **false**. You can also override the **ConfigureContainer** method and disable services that you do not want to use, such as the event aggregator.

Note: If you turn off the default registration, you will need to manually register required services.

To extend the default behavior of **ConfigureContainer**, simply add an override to your application's bootstrapper and optionally call the base implementation, as shown in the following code from the **QuickStartBootstrapper** from the Modularity for WPF (with Unity) QuickStart. This implementation calls the base class's implementation, registers the **ModuleTracker** type as the concrete implementation of **IModuleTracker**, and registers the **callbackLogger** as a singleton instance of **CallbackLogger** with Unity.

C#

```

protected override void ConfigureContainer()
{
    base.ConfigureContainer();

    this.RegisterTypeIfMissing(typeof(IModuleTracker), typeof(ModuleTracker), true);
    this.Container.RegisterInstance<CallbackLogger>(this.callbackLogger);
}

```

Creating and Configuring the Container in the MefBootstrapper

The **MefBootstrapper** class's **CreateContainer** method does several things. First, it creates an **AssemblyCatalog** and a **CatalogExportProvider**. The **CatalogExportProvider** allows the **MefExtensions** assembly to provide default exports for a number of Prism types and still allows you to override the default type registration. Then **CreateContainer** creates and returns a new instance of a **CompositionContainer** using the **CatalogExportProvider**. In most cases, you will not need to change this functionality; however, the method is virtual, thereby allowing that flexibility.

After the container is created, it needs to be configured for your application. The **ConfigureContainer** implementation in the **MefBootstrapper** registers a number of core Prism services by default, as shown in the following code example. If you override this method, consider carefully whether you should invoke the base class's implementation to register the core Prism services, or if you will provide these services in your implementation.

C#

```
protected virtual void ConfigureContainer()
{
    this.RegisterBootstrapperProvidedTypes();
}

protected virtual void RegisterBootstrapperProvidedTypes()
{
    this.Container.ComposeExportedValue<ILoggerFacade>(this.Logger);
    this.Container.ComposeExportedValue<IModuleCatalog>(this.ModuleCatalog);
    this.Container.ComposeExportedValue<IServiceLocator>(new
MefServiceLocatorAdapter(this.Container));
    this.Container.ComposeExportedValue<AggregateCatalog>(this.AggregateCatalog);
}
```

Note: In the **MefBootstrapper**, the core services of Prism are added to the container as singletons so they can be located through the container throughout the application.

In addition to providing the **CreateContainer** and **ConfigureContainer** methods, the **MefBootstrapper** also provides two methods to create and configure the **AggregateCatalog** used by MEF. The **CreateAggregateCatalog** method simply creates and returns an **AggregateCatalog** object. Like the other methods in the **MefBootstrapper**, **CreateAggregateCatalog** is virtual and can be overridden if necessary.

The **ConfigureAggregateCatalog** method allows you to add type registrations to the **AggregateCatalog** imperatively. For example, the **QuickStartBootstrapper** from the Modularity with MEF QuickStart explicitly adds ModuleA and ModuleC to the **AggregateCatalog**, as shown here.

C#

```
protected override void ConfigureAggregateCatalog()
{
    base.ConfigureAggregateCatalog();
    // Add this assembly to export ModuleTracker
    thisAggregateCatalog.Catalogs.Add(
        new AssemblyCatalog(typeof(QuickStartBootstrapper).Assembly));
    // Module A is referenced in in the project and directly in code.
    thisAggregateCatalog.Catalogs.Add(
        new AssemblyCatalog(typeof(ModuleA.ModuleA).Assembly));
    thisAggregateCatalog.Catalogs.Add(
        new AssemblyCatalog(typeof(ModuleC.ModuleC).Assembly));

    // Module B and Module D are copied to a directory as part of a post-build step.
}
```

```
// These modules are not referenced in the project and are discovered by
// inspecting a directory.
// Both projects have a post-build step to copy themselves into that directory.
DirectoryCatalog catalog = new DirectoryCatalog("DirectoryModules");
thisAggregateCatalog.Catalogs.Add(catalog);
}
```

More Information

For more information about MEF, **AggregateCatalog**, and **AssemblyCatalog**, see [Managed Extensibility Framework Overview](#) on MSDN.

3: Managing Dependencies Between Components

Applications based on the Prism Library are composite applications that potentially consist of many loosely coupled types and services. They need to interact to contribute content and receive notifications based on user actions. Because they are loosely coupled, they need a way to interact and communicate with one another to deliver the required business functionality. To tie together these various pieces, applications based on the Prism Library rely on a dependency injection container.

Dependency injection containers reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first. In some cases, the container itself is resolved as a dependency. For example, when using the Unity Application Block (Unity) as the container, modules have the container injected, so they can register their views and services with that container.

There are several advantages of using a container:

- A container removes the need for a component to locate its dependencies or manage their lifetimes.
- A container allows swapping of implemented dependencies without affecting the component.
- A container facilitates testability by allowing dependencies to be mocked.
- A container increases maintainability by allowing new components to be easily added to the system.

In the context of an application based on the Prism Library, there are specific advantages to a container:

- A container injects module dependencies into the module when it is loaded.
- A container is used for registering and resolving view models and views.
- A container can create the view models and injects the view.
- A container injects the composition services, such as the region manager and the event aggregator.
- A container is used for registering module-specific services, which are services that have module-specific functionality.

Note: Some samples in the Prism guidance rely on the Unity Application Block (Unity) as the container. Other code samples, for example the Modularity QuickStarts, use Managed Extensibility Framework

(MEF). The Prism Library itself is not container-specific, and you can use its services and patterns with other containers, such as Castle Windsor, StructureMap, and Spring.NET.

Key Decision: Choosing a Dependency Injection Container

The Prism Library provides two options for dependency injection containers: Unity or MEF. Prism is extensible, thereby allowing other containers to be used instead with a little bit of work. Both Unity and MEF provide the same basic functionality for dependency injection, even though they work very differently. Some of the capabilities provided by both containers include the following:

- They both register types with the container.
- They both register instances with the container.
- They both imperatively create instances of registered types.
- They both inject instances of registered types into constructors.
- They both inject instances of registered types into properties.
- They both have declarative attributes for marking types and dependencies that need to be managed.
- They both resolve dependencies in an object graph.

Unity provides several capabilities that MEF does not:

- It resolves concrete types without registration.
- It resolves open generics.
- It uses interception to capture calls to objects and add additional functionality to the target object.

MEF provides several capabilities that Unity does not:

- It discovers assemblies in a directory.
- It uses XAP file download and assembly discovery.
- It recomposes properties and collections as new types are discovered.
- It automatically exports derived types.
- It is deployed with the .NET Framework.

The containers have differences in capabilities and work differently, but the Prism Library will work with either container and provide similar functionality. When considering which container to use, keep in mind the preceding capabilities and determine which fits your scenario better.

Considerations for Using the Container

You should consider the following before using containers:

- Consider whether it is appropriate to register and resolve components using the container:
 - Consider whether the performance impact of registering with the container and resolving instances from it is acceptable in your scenario. For example, if you need to create 10,000 polygons to draw a surface within the local scope of a rendering method, the cost of resolving all of those polygon instances through the container might have a significant performance cost because of the container's use of reflection for creating each entity.
 - If there are many or deep dependencies, the cost of creation can increase significantly.
 - If the component does not have any dependencies or is not a dependency for other types, it may not make sense to put it in the container.
 - If the component has a single set of dependencies that are integral to the type and will never change, it may not make sense to put it in the container.
- Consider whether a component's lifetime should be registered as a singleton or instance:
 - If the component is a global service that acts as a resource manager for a single resource, such as a logging service, you may want to register it as a singleton.
 - If the component provides shared state to multiple consumers, you may want to register it as a singleton.
 - If the object that is being injected needs to have a new instance of it injected each time a dependent object needs one, register it as a non-singleton. For example, each view probably needs a new instance of a view model.
- Consider whether you want to configure the container through code or configuration:
 - If you want to centrally manage all the different services, configure the container through configuration.
 - If you want to conditionally register specific services, configure the container through code.
 - If you have module-level services, consider configuring the container through code so that those services are registered only if the module is loaded.

Note: Some containers, such as MEF, cannot be configured via a configuration file and must be configured via code.

Core Scenarios

Containers are used for two primary purposes, namely registering and resolving.

Registering

Before you can inject dependencies into an object, the types of the dependencies need to be registered with the container. Registering a type typically involves passing the container an interface and a concrete type that implements that interface. There are primarily two means for registering types and objects: through code or through configuration. The specific means vary from container to container.

Typically, there are two ways of registering types and objects in the container through code:

- You can register a type or a mapping with the container. At the appropriate time, the container will build an instance of the type you specify.
- You can register an existing object instance in the container as a singleton. The container will return a reference to the existing object.

Registering Types with the Unity Container

During initialization, a type can register other types, such as views and services. Registration allows their dependencies to be provided through the container and allows them to be accessed from other types. To do this, the type will need to have the container injected into the module constructor. The following code shows how the **OrderModule** type in the Commanding QuickStart registers a type.

C#

```
// OrderModule.cs
public class OrderModule : IModule
{
    public void Initialize()
    {
        this.container.RegisterType<IOrdersRepository, OrdersRepository>(new
ContainerControlledLifetimeManager());
        ...
    }
    ...
}
```

Depending on which container you use, registration can also be performed outside the code through configuration. For an example of this, see [Registering Modules using a Configuration File](#) in [Modular Application Development](#).

Note: The advantage of registering in code, compared to configuration, is that the registration happens only if the module loads.

Registering Types with MEF

MEF uses an attribute-based system for registering types with the container. As a result, adding type registration to the container is simple: it requires the addition of the **[Export]** attribute to a type as shown in the following code example.

C#

```
[Export(typeof(ILoggerFacade))]
public class CallbackLogger: ILoggerFacade
{}
```

Another option when using MEF is to create an instance of a class and register that particular instance with the container. The **QuickStartBootstrapper** in the Modularity with MEF QuickStart shows an example of this in the **ConfigureContainer** method, as shown here.

C#

```
protected override void ConfigureContainer()
{
    base.ConfigureContainer();

    // Because we created the CallbackLogger and it needs to
    // be used immediately, we compose it to satisfy any imports it has.
    this.Container.ComposeExportedValue<CallbackLogger>(this.callbackLogger);
}
```

Note: When using MEF as your container, it is recommended that you use attributes to register types.

Resolving

After a type is registered, it can be resolved or injected as a dependency. When a type is being resolved, and the container needs to create a new instance, it injects the dependencies into these instances.

In general, when a type is resolved, one of three things happens:

- If the type has not been registered, the container throws an exception.

Note: Some containers, including Unity, allow you to resolve a concrete type that has not been registered.

- If the type has been registered as a singleton, the container returns the singleton instance. If this is the first time the type was called for, the container creates it and holds on to it for future calls.
- If the type has not been registered as a singleton, the container returns a new instance.

Note: By default, types registered with MEF are singletons and the container holds a reference to the object. In Unity, new instances of objects are returned by default, and the container does not maintain a reference to the object.

Resolving Instances with Unity

The following code example from the Commanding QuickStart shows where the **OrdersEditorView** and **OrdersToolBar** views are resolved from the container to associate them to the corresponding regions.

```
C#
// OrderModule.cs
public class OrderModule : IModule
{
    public void Initialize()
    {
        this.container.RegisterType<IOrdersRepository, OrdersRepository>(new
ContainerControlledLifetimeManager());

        // Show the Orders Editor view in the shell's main region.
        this.regionManager.RegisterViewWithRegion("MainRegion",
            () => this.container.Resolve<OrdersEditorView>());

        // Show the Orders Toolbar view in the shell's toolbar region.
        this.regionManager.RegisterViewWithRegion("GlobalCommandsRegion",
            () => this.container.Resolve<OrdersToolBar>());
    }
    ...
}
```

The **OrdersEditorViewModel** constructor contains the following dependencies (the orders repository and the orders command proxy), which are injected when it is resolved.

```
C#
// OrdersEditorViewModel.cs
public OrdersEditorViewModel(IOrdersRepository ordersRepository, OrdersCommandProxy
commandProxy)
{
    this.ordersRepository = ordersRepository;
    this.commandProxy     = commandProxy;

    // Create dummy order data.
    this.PopulateOrders();

    // Initialize a CollectionView for the underlying Orders collection.
    this.Orders = new ListCollectionView( _orders );
    // Track the current selection.
    this.Orders.CurrentChanged += SelectedOrderChanged;
    this.Orders.MoveCurrentTo(null);
}
```

In addition to the constructor injection shown in the preceding code, Unity also allows for property injection. Any properties that have a **[Dependency]** attribute applied are automatically resolved and injected when the object is resolved.

Resolving Instances with MEF

The following code example shows how the **Bootstrapper** in the Modularity with MEF QuickStart obtains an instance of the shell. Instead of requesting a concrete type, the code could request an instance of an interface.

C#

```
protected override DependencyObject CreateShell()
{
    return this.Container.GetExportedValue<Shell>();
```

In any class that is resolved by MEF, you can also use constructor injection, as shown in the following code example from ModuleA in the Modularity with MEF QuickStart, which has an **ILoggerFacade** and an **IModuleTracker** injected.

C#

```
[ImportingConstructor]
public ModuleA(ILoggerFacade logger, IModuleTracker moduleTracker)
{
    if (logger == null)
    {
        throw new ArgumentNullException("logger");
    }
    if (moduleTracker == null)
    {
        throw new ArgumentNullException("moduleTracker");
    }
    this.logger = logger;
    this.moduleTracker = moduleTracker;
    this.moduleTracker.RecordModuleConstructed(WellKnownModuleNames.ModuleA);
}
```

Another option is to use property injection, as shown in the **ModuleTracker** class from the Modularity with MEF QuickStart, which has an instance of the **ILoggerFacade** injected.

C#

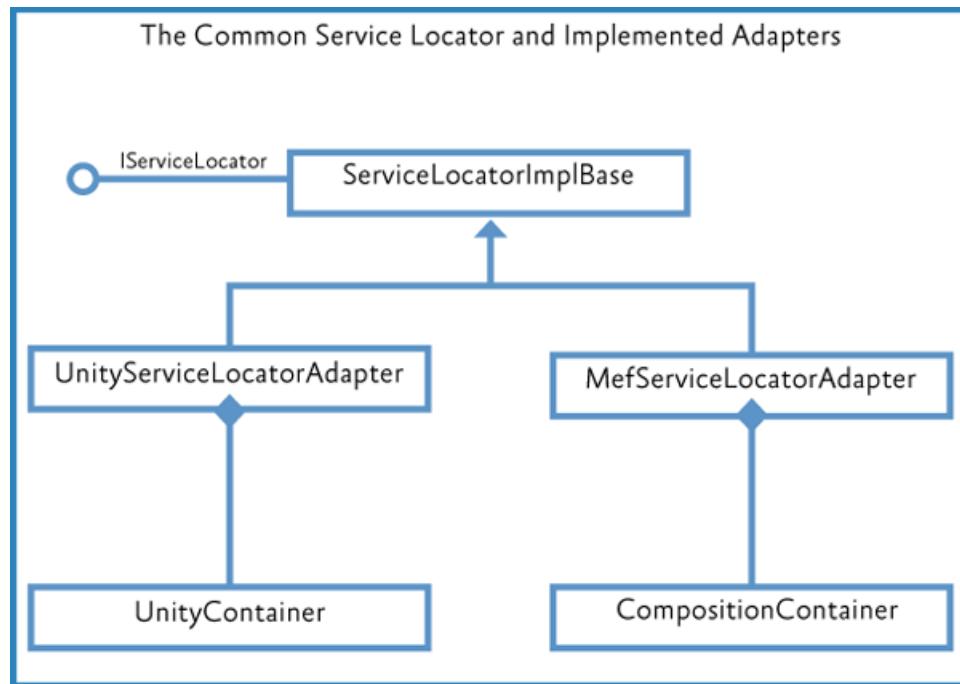
```
[Export(typeof(IModuleTracker))]
public class ModuleTracker : IModuleTracker
{
    [Import] private ILoggerFacade Logger;
```

Using Dependency Injection Containers and Services in Prism

Dependency injection containers, often referred to as just "containers," are used to satisfy dependencies between components; satisfying these dependencies typically involves registration and resolution. The Prism Library provides support for the Unity container and for MEF, but it is not container-specific. Because the library accesses the container through the **IServiceLocator** interface, the

container can be replaced. To do this, your container must implement the **IServiceLocator** interface. Usually, if you are replacing the container, you will also need to provide your own container-specific bootstrapper. The **IServiceLocator** interface is defined in the Common Service Locator Library. This is an open source effort to provide an abstraction over IoC (Inversion of Control) containers, such as dependency injection containers, and service locators. The objective of using this library is to leverage IoC and Service Location without tying to a specific implementation.

The Prism Library provides the **UnityServiceLocatorAdapter** and the **MefServiceLocatorAdapter**. Both adapters implement the **IServiceLocator** interface by extending the **ServiceLocatorImplBase** type. The following illustration shows the class hierarchy.



The Common Service Locator implementations in Prism

Although the Prism Library does not reference or rely on a specific container, it is typical for an application to rely on a specific container. This means that it is reasonable for a specific application to refer to the container, but the Prism Library does not refer to the container directly. For example, the Stock Trader RI and several of the QuickStarts included with Prism rely on Unity as the container. Other samples and QuickStarts rely on MEF.

IServiceLocator

The following code shows the **IServiceLocator** interface.

C#

```

public interface IServiceLocator : IServiceProvider
{
    object GetInstance(Type serviceType);
    object GetInstance(Type serviceType, string key);
  
```

```

IEnumerable<object> GetAllInstances(Type serviceType);
TService GetInstance<TService>();
TService GetInstance<TService>(string key);
IEnumerable<TService> GetAllInstances<TService>();
}

```

The Service Locator is extended in the Prism Library with the extension methods shown in the following code. You can see that **IServiceLocator** is used only for resolving, meaning it is used to obtain an instance; it is not used for registration.

C#

```

// ServiceLocatorExtensions
public static class ServiceLocatorExtensions
{
    public static object TryResolve(this IServiceLocator locator, Type type)
    {
        try
        {
            return locator.GetInstance(type);
        }
        catch (ActivationException)
        {
            return null;
        }
    }

    public static T TryResolve<T>(this IServiceLocator locator) where T: class
    {
        return locator.TryResolve(typeof(T)) as T;
    }
}

```

The **TryResolve** extension method—which the Unity container does not support—returns an instance of the type to be resolved if it has been registered; otherwise, it returns **null**.

The **ModuleInitializer** uses **IServiceLocator** for resolving the module during module loading, as shown in the following code examples.

C#

```

// ModuleInitializer.cs - Initialize()
IModule moduleInstance = null;
try
{
    moduleInstance = this.CreateModule(moduleInfo);
    moduleInstance.Initialize();
}
...

```

C#

```
// ModuleInitializer.cs - CreateModule()
protected virtual IModule CreateModule(string typeName)
{
    Type moduleType = Type.GetType(typeName);
    if (moduleType == null)
    {
        throw new ModuleInitializeException(string.Format(CultureInfo.CurrentCulture,
Properties.Resources.FailedToGetType, typeName));
    }

    return (IModule)this.serviceLocator.GetInstance(moduleType);
}
```

Considerations for Using **IServiceLocator**

IServiceLocator is not meant to be the general-purpose container. Containers have different semantics of usage, which often drives the decision for why that container is chosen. Bearing this in mind, the Stock Trader RI uses the dependency injection container directly instead of using the **IServiceLocator**. This is the recommend approach for your application development.

In the following situations, it may be appropriate for you to use the **IServiceLocator**:

- You are an independent software vendor (ISV) designing a third-party service that needs to support multiple containers.
- You are designing a service to be used in an organization where they use multiple containers.

More Information

For information related to containers, see the following:

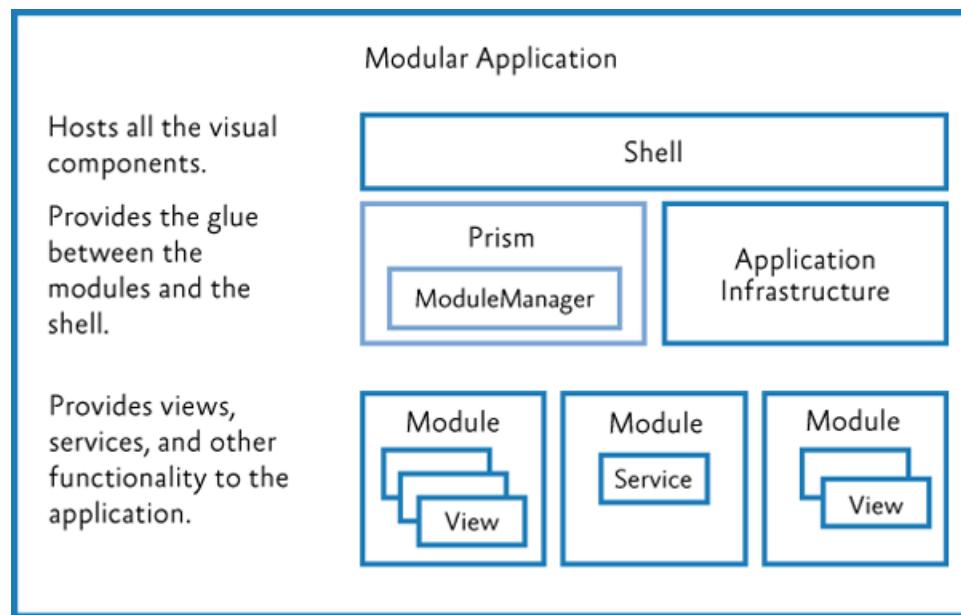
- [Unity Application Block](#) on MSDN.
- [Unity community site](#) on CodePlex.
- [Managed Extensibility Framework Overview](#) on MSDN.
- [MEF community site](#) on CodePlex.
- [Inversion of Control containers and the Dependency Injection pattern](#) on Martin Fowler's website.
- [Design Patterns: Dependency Injection](#) in *MSDN Magazine*.
- [Loosen Up: Tame Your Software Dependencies for More Flexible Apps](#) in *MSDN Magazine*.
- [Castle Project](#)
- [StructureMap](#)
- [Spring.NET](#)

4: Modular Application Development

A modular application is an application that is divided into a set of loosely coupled functional units (named modules) that can be integrated into a larger application. A client module encapsulates a portion of the application's overall functionality and typically represents a set of related concerns. It can include a collection of related components, such as application features, including user interface and business logic, or pieces of application infrastructure, such as application-level services for logging or authenticating users. Modules are independent of one another but can communicate with each other in a loosely coupled fashion. Using a modular application design makes it easier for you to develop, test, deploy, and maintain your application.

For example, consider a personal banking application. The user can access a variety of functions, such as transferring money between accounts, paying bills, and updating personal information from a single user interface (UI). However, behind the scenes, each of these functions is encapsulated within a discrete module. These modules communicate with each other and with back-end systems such as database servers and web services. Application services integrate the various components within each of the different modules and handle the communication with the user. The user sees an integrated view that looks like a single application.

The following illustration shows a design of a modular application with multiple modules.



Module composition

Benefits of Building Modular Applications

You are probably already building a well-architected application using assemblies, interfaces, and classes, and employing good object-oriented design principles. Even so, unless great care is taken, your

application design may still be "monolithic" (where all the functionality is implemented in a tightly coupled way within the application), which can make the application difficult to develop, test, extend, and maintain.

The modular application approach, on the other hand, can help you to identify the large scale functional areas of your application and allow you to develop and test that functionality independently. This can make development and testing easier, but it can also make your application more flexible and easier to extend in the future. The benefit of the modular approach is that it can make your overall application architecture more flexible and maintainable because it allows you to break your application into manageable pieces. Each piece encapsulates specific functionality, and each piece is integrated through clear but loosely coupled communication channels.

Prism's Support for Modular Application Development

Prism provides support for modular application development and for run-time module management within your application. Using Prism's modular development functionality can save you time because you don't have to implement and test your own modularity framework. Prism supports the following modular application development features:

- A module catalog for registering named modules and each module's location; you can create the module catalog in the following ways:
 - By defining modules in code or Extensible Application Markup Language (XAML)
 - By discovering modules in a directory so you can load all your modules without explicitly defining in a centralized catalog
 - By defining modules in a configuration file
- Declarative metadata attributes for modules to support initialization mode and dependencies
- Integration with dependency injection containers to support loose coupling between modules
- For module loading:
 - Dependency management, including duplicate and cycle detection to ensure modules are loaded in the correct order and only loaded and initialized once
 - On-demand and background downloading of modules to minimize application start-up time; the rest of the modules can be loaded and initialized in the background or when they are required

Core Concepts

This section introduces the core concepts related to modularity in Prism, including the **IModule** interface, the module loading process, the module catalog, communicating between modules, and dependency injection containers.

IModule: The Building Block of Modular Applications

A module is a logical collection of functionality and resources that is packaged in a way that can be separately developed, tested, deployed, and integrated into an application. A package can be one or more assemblies. Each module has a central class that is responsible for initializing the module and integrating its functionality into the application. That class implements the **IModule** interface.

Note: The presence of a class that implements the **IModule** interface is enough to identify the package as a module.

The **IModule** interface has a single method, named **Initialize**, within which you can implement whatever logic is required to initialize and integrate the module's functionality into the application. Depending on the purpose of the module, it can register views into composite user interfaces, make additional services available to the application, or extend the application's functionality. The following code shows the minimum implementation for a module.

C#

```
public class MyModule : IModule
{
    public void Initialize()
    {
        // Do something here.
    }
}
```

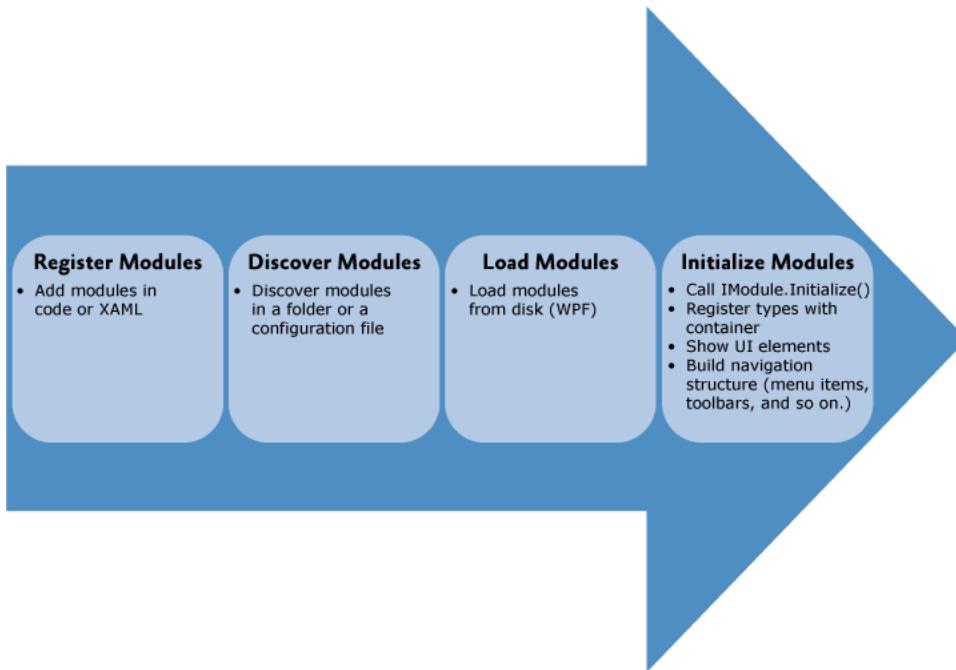
Note: Instead of using the initialization mechanism provided by the **IModule** interface, the Stock Trader RI uses a declarative, attribute-based approach for registering views, services, and types.

Module Lifetime

The module loading process in Prism includes the following:

1. **Registering/discovering modules.** The modules to be loaded at run-time for a particular application are defined in a Module catalog. The catalog contains information about the modules to be loaded, their location, and the order in which they are to be loaded.
2. **Loading modules.** The assemblies that contain the modules are loaded into memory. This phase may require the module to be retrieved from some remote location or local directory.
3. **Initializing modules.** The modules are then initialized. This means creating instances of the module class and calling the **Initialize** method on them via the **IModule** interface.

The following figure shows the module loading process.



Module loading process

Module Catalog

The **ModuleCatalog** holds information about the modules that can be used by the application. The catalog is essentially a collection of **ModuleInfo** classes. Each module is described in a **ModuleInfo** class that records the name, type, and location, among other attributes of the module. There are several typical approaches to filling the **ModuleCatalog** with **ModuleInfo** instances:

- Registering modules in code
- Registering modules in XAML
- Registering modules in a configuration file
- Discovering modules in a local directory on disk

The registration and discovery mechanism you should use depends on what your application needs. Using a configuration file or XAML file allows your application to not require references to the modules. Using a directory can allow an application to discover modules without having to specify them in a file.

Controlling When to Load a Module

Prism applications can initialize modules as soon as possible, known as "when available," or when the application needs them, known as "on-demand." Consider the following guidelines for loading modules:

- Modules required for the application to run must be loaded with the application and initialized when the application runs.

- Modules containing features that are almost always used in typical usage of the application can be loaded in the background and initialized when they become available.
 - Modules containing features that are rarely used (or are support modules that other modules optionally depend upon) can be loaded and initialized on-demand.
-

Consider how you are partitioning your application, common usage scenarios, application start-up time, and the number and size of downloads to determine how to configure your module for downloading and initialization.

Integrate Modules with the Application

Prism provides the following classes to bootstrap your application: the **UnityBootstrapper** or the **MefBootstrapper**. These classes can be used to create and configure the module manager to discover and load modules. You can override a configuration method to register modules specified in a XAML file, a configuration file, or a directory location in a few lines of code.

Use the module **Initialize** method to integrate the module with the rest of the application. The way you do this varies, depending on the structure of your application and the content of the module. The following are common things to do to integrate your module into your application:

- Add the module's views to the application's navigation structure. This is common when building composite UI applications using view discovery or view injection.
 - Subscribe to application level events or services.
 - Register shared services with the application's dependency injection container.
-

Communicate Between Modules

Even though modules should have low coupling between each other, it is common for modules to communicate with each other. There are several loosely coupled communication patterns, each with their own strengths. Typically, combinations of these patterns are used to create the resulting solution. The following are some of these patterns:

- **Loosely coupled events.** A module can broadcast that a certain event has occurred. Other modules can subscribe to those events so they will be notified when the event occurs. Loosely coupled events are a lightweight manner of setting up communication between two modules; therefore, they are easily implemented. However, a design that relies too heavily on events can become hard to maintain, especially if many events have to be orchestrated together to fulfill a single task. In that case, it might be better to consider a shared service.
- **Shared services.** A shared service is a class that can be accessed through a common interface. Typically, shared services are found in a shared assembly and provide system-wide services, such as authentication, logging, or configuration.

- **Shared resources.** If you do not want modules to directly communicate with each other, you can also have them communicate indirectly through a shared resource, such as a database or a set of web services.

Dependency Injection and Modular Applications

Containers like the Unity Application Block (Unity) and Managed Extensibility Framework (MEF) allow you to easily use Inversion of Control (IoC) and Dependency Injection, which are powerful design patterns that help to compose components in a loosely-coupled fashion. It allows components to obtain references to the other components that they depend on without having to hard code those references, thereby promoting better code re-use and improved flexibility. Dependency injection is very useful when building a loosely coupled, modular application. Prism is designed to be agnostic about the dependency injection container used to compose components within an application. The choice of container is up to you and will largely depend on your application requirements and preferences. However, there are two principal dependency injection frameworks from Microsoft to consider – Unity and MEF.

The patterns & practices Unity Application Block provides a fully-featured dependency injection container. It supports property-based and constructor-based injection and policy injection, which allows you to transparently inject behavior and policy between components; it also supports a host of other features that are typical of dependency injection containers.

MEF (which is part of .NET Framework 4.5) provides support for building extensible .NET applications by supporting dependency injection-based component composition and provides other features that support modular application development. It allows an application to discover components at run time and then to integrate those components into the application in a loosely-coupled way. MEF is a great extensibility and composition framework. It includes assembly and type discovery, type dependency resolution, dependency injection, and some nice assembly download capabilities. Prism supports taking advantage of MEF features, as well as the following:

- Module registration through XAML and code attributes
- Module registration through configuration files and directory scans
- State tracking as the module is loaded
- Custom declarative metadata for modules when using MEF

Both the Unity and MEF dependency injection containers work seamlessly with Prism.

Key Decisions

The first decision you will make is whether you want to develop a modular solution. There are numerous benefits of building modular applications as discussed in the previous section, but there is a commitment in terms of time and effort that you need to make to reap these benefits. If you decide to develop a modular solution, there are several more things to consider:

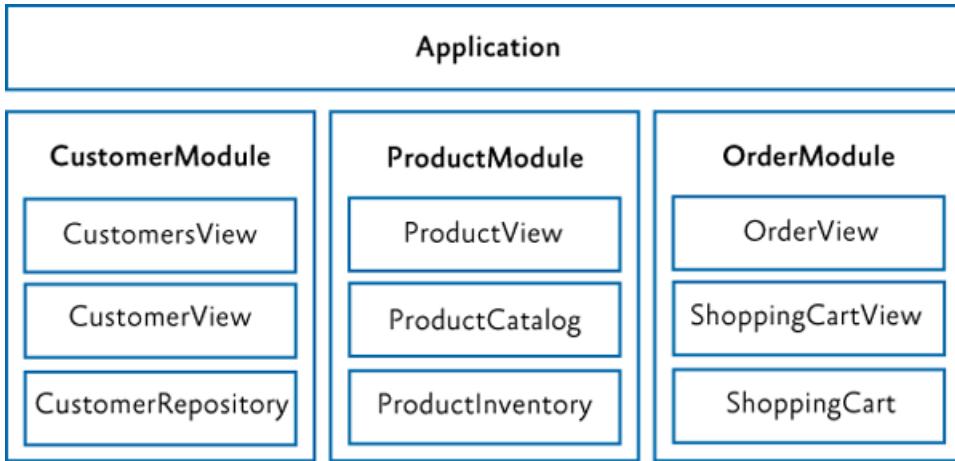
- **Determine the framework you will use.** You can create your own modularity framework, use Prism, MEF, or another framework.
- **Determine how to organize your solution.** Approach a modular architecture by defining the boundaries of each module, including what assemblies are part of each module. You can decide to use modularity to ease the development, as well as to have control over how the application will be deployed or if it will support a plug-in or extensible architecture.
- **Determine how to partition your modules.** Modules can be partitioned differently based on requirements, for example, by functional areas, provider modules, development teams and deployment requirements.
- **Determine the core services that the application will provide to all modules.** An example is that core services could be an error reporting service or an authentication and authorization service.
- **If you are using Prism, determine what approach you are using to register modules in the module catalog.** For WPF, you can register modules in code, XAML, in a configuration file, or discovering modules in a local directory on disk. **Determine your module communication and dependency strategy.** Modules will need to communicate with each other, and you will need to deal with dependencies between modules.
- **Determine your dependency injection container.** Typically, modular systems require dependency injection, inversion of control, or service locator to allow the loose coupling and dynamic loading and creating of modules. Prism allows a choice between using the Unity, MEF, or another container and provides libraries for Unity or MEF-based applications.
- **Minimize application startup time.** Think about on-demand and background downloading of modules to minimize application startup time.
- **Determine deployment requirements.** You will need to think about how you intend to deploy your application.

The next sections provide details about some of these decisions.

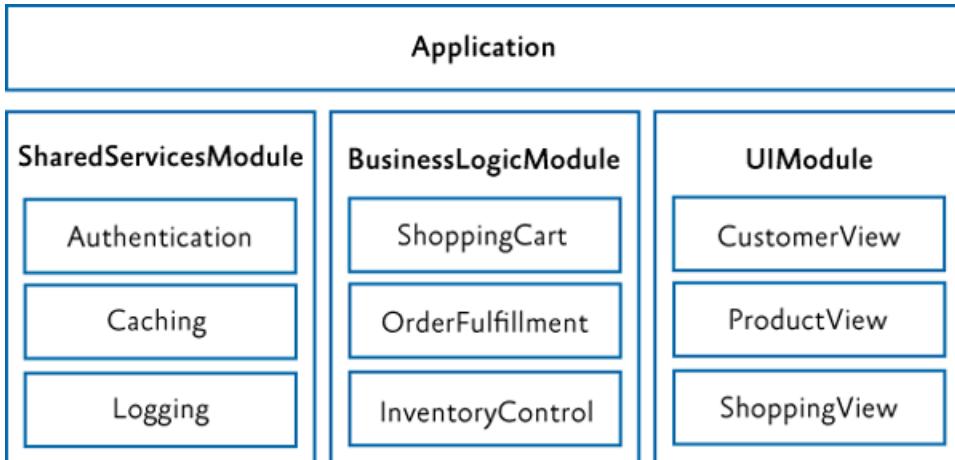
Partition Your Application into Modules

When you develop your application in a modularized fashion, you structure the application into separate client modules that can be individually developed, tested, and deployed. Each module will encapsulate a portion of your application's overall functionality. One of the first design decisions you will have to make is to decide how to partition your application's functionality into discrete modules.

A module should encapsulate a set of related concerns and have a distinct set of responsibilities. A module can represent a vertical slice of the application or a horizontal service layer. Large applications will likely have both types of modules.



An application with modules organized around vertical slices



An application with modules organized around horizontal layers

A larger application may have modules organized with vertical slices and horizontal layers. Some examples of modules include the following:

- A module that contains a specific application feature, such as the News module in the Stock Trader Reference Implementation (Stock Trader RI)
- A module that contains a specific sub-system or functionality for a set of related use cases, such as purchasing, invoicing, or general ledger
- A module that contains infrastructure services, such as logging, caching, and authorization services, or web services
- A module that contains services that invoke line-of-business (LOB) systems, such as Siebel CRM and SAP, in addition to other internal systems

A module should have a minimal set of dependencies on other modules. When a module has a dependency on another module, it should be loosely coupled by using interfaces defined in a shared library instead of concrete types, or by using the **EventAggregator** to communicate with other modules via **EventAggregator** event types.

The goal of modularity is to partition the application in such a way that it remains flexible, maintainable, and stable even as features and technologies are added and removed. The best way to accomplish this is to design your application so that modules are as independent as possible, have well defined interfaces, and are as isolated as possible.

Determine Ratio of Projects to Modules

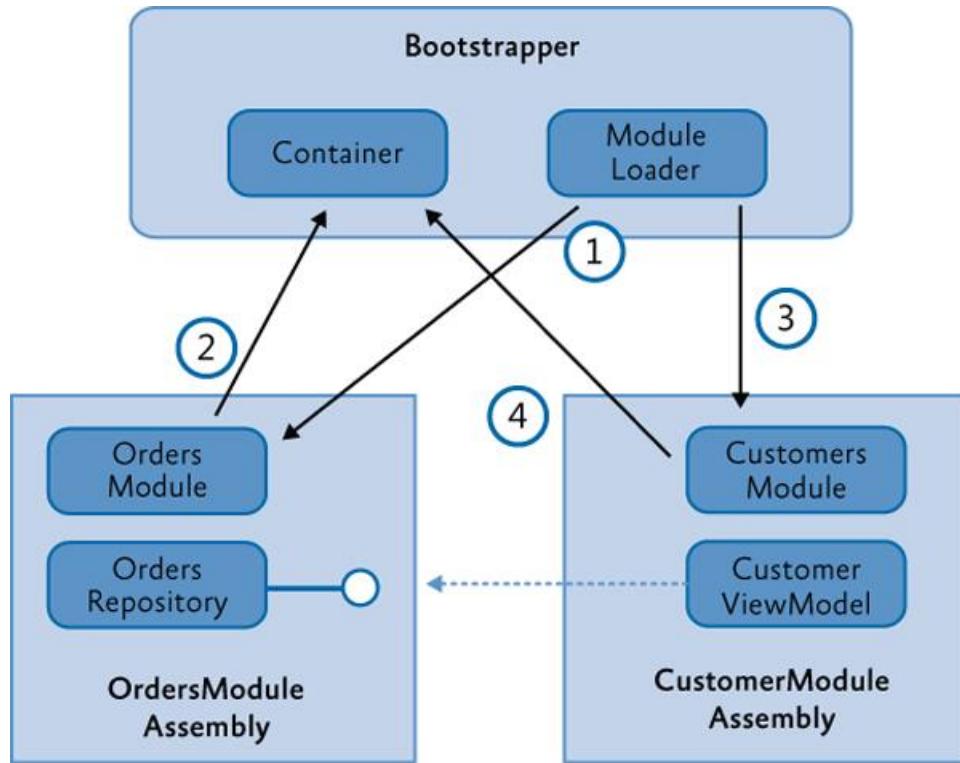
There are several ways to create and package modules. The recommended and most common way is to create a single assembly per module. This helps keep logical modules separate and promotes proper encapsulation. It also makes it easier to talk about the assembly as the module boundary as well as the packaging of how you deploy the module. However, nothing prevents a single assembly from containing multiple modules, and in some cases this may be preferred to minimize the number of projects in your solution. For a large application, it is not uncommon to have 10–50 modules. Separating each module into its own project adds a lot of complexity to the solution and can slow down Visual Studio performance. Sometimes it makes sense to break a module or set of modules into their own solution to manage this if you choose to stick to one module per assembly/Visual Studio project.

Use Dependency Injection for Loose Coupling

A module may depend on components and services provided by the host application or by other modules. Prism supports the ability to register dependencies between modules so that they are loaded and initialized in the right order. Prism also supports the initialization of modules when they are loaded into the application. During module initialization, the module can retrieve references to the additional components and services it requires, and/or register any components and services that it contains in order to make them available to other modules.

A module should use an independent mechanism to get instances of external interfaces instead of directly instantiating a concrete type, for example by using a dependency injection container or factory service. Dependency injection containers such as Unity or MEF allow a type to automatically acquire instances of the interfaces and types it needs through dependency injection. Prism integrates with both Unity and MEF to allow a module to easily use dependency injection.

The following diagram shows the typical sequence of operations when modules are loaded that need to acquire or register references to the components and services.



Example of dependency injection

In this example, the **OrdersModule** assembly defines an **OrdersRepository** class (along with other views and classes that implement order functionality). The **CustomerModule** assembly defines a **CustomersViewModel** class which depends on the **OrdersRepository**, typically based on an interface exposed by the service. The application startup and bootstrapping process contains the following steps:

1. The bootstrapper starts the module initialization process, and the module loader loads and initializes the **OrdersModule**.
2. In the initialization of the **OrdersModule**, it registers the **OrdersRepository** with the container.
3. The module loader then loads the **CustomerModule**. The order of module loading can be specified by the dependencies in the module metadata.
4. The **CustomerModule** constructs an instance of the **CustomerViewModel** by resolving it through the container. The **CustomerViewModel** has a dependency on the **OrdersRepository** (typically based on its interface) and indicates it through constructor or property injection. The container injects that dependency in the construction of the view model based on the type registered by the **OrdersModule**. The net result is an interface reference from the **CustomerViewModel** to the **OrderRepository** without tight coupling between those classes.

Note: The interface used to expose the **OrderRepository (IOrderRepository)** could reside in a separate "shared services" assembly or an "orders services" assembly that only contains the service interfaces and types required to expose those services. This way, there is no hard dependency between the **CustomersModule** and the **OrdersModule**.

Note that both modules have an implicit dependency on the dependency injection container. This dependency is injected during module construction in the module loader.

Core Scenarios

This section describes the common scenarios you will encounter when working with modules in your application. These scenarios include defining a module, registering and discovering modules, loading modules, initializing modules, specifying module dependencies, loading modules on demand, downloading remote modules in the background, and detecting when a module has already been loaded. You can register and discover modules in code, in a XAML or application configuration file, or by scanning a local directory.

Defining a Module

A module is a logical collection of functionality and resources that is packaged in a way that can be separately developed, tested, deployed, and integrated into an application. Each module has a central class that is responsible for initializing the module and integrating its functionality into the application. That class implements the **IModule** interface, as shown here.

C#

```
public class MyModule : IModule
{
    public void Initialize()
    {
        // Initialize module
    }
}
```

The way you implement the **Initialize** method will depend on the requirements of your application. The module class type, initialization mode, and any module dependencies are defined in the module catalog. For each module in the catalog, the module loader creates an instance of the module class, and then it calls the **Initialize** method. Modules are processed in the order specified in the module catalog. The runtime initialization order is based on when the modules are downloaded, available, and the dependencies are satisfied.

Depending on the type of module catalog that your application is using, module dependencies can be set either by declarative attributes on the module class itself or within the module catalog file. The following sections provide more details.

Registering and Discovering Modules

The modules that an application can load are defined in a module catalog. The Prism Module Loader uses the module catalog to determine which modules are available to be loaded into the application, when to load them, and in which order they are to be loaded.

The module catalog is represented by a class that implements the **IModuleCatalog** interface. The module catalog class is created by the application bootstrapper class during application initialization. Prism provides different implementations of module catalog for you to choose from. You can also populate a module catalog from another data source by calling the **AddModule** method or by deriving from **ModuleCatalog** to create a module catalog with customized behavior.

Note: Typically, modules in Prism use a dependency injection container and the Common Service Locator to retrieve instances of types that are required for module initialization. Both the Unity and the MEF containers are supported by Prism. Although the overall process of registering, discovering, downloading, and initializing modules is the same, the details can vary based on whether Unity or MEF is being used. The container-specific differences between approaches are explained throughout this topic.

Registering Modules in Code

The most basic module catalog is provided by the **ModuleCatalog** class. You can use this module catalog to programmatically register modules by specifying the module class type. You can also programmatically specify the module name and initialization mode. To register the module directly with the **ModuleCatalog** class, call the **AddModule** method in your application's **Bootstrapper** class. An example is shown in the following code.

C#

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

In the preceding example, the modules are directly referenced by the shell, so the module class types are defined and can be used in the call to **AddModule**. That is why this example uses **typeof(Module)** to add modules to the catalog.

Note: If your application has a direct reference to the module type, you can add it by type as shown above; otherwise you need to provide the fully qualified type name and the location of the assembly.

To see another example of defining the module catalog in code, see `StockTraderRIBootstrapper.cs` in the Stock Trader Reference Implementation (Stock Trader RI).

Note: The **Bootstrapper** base class provides the **CreateModuleCatalog** method to assist in the creation of the **ModuleCatalog**. By default, this method creates a **ModuleCatalog** instance, but this method can be overridden in a derived class in order to create different types of module catalog.

Registering Modules Using a XAML File

You can define a module catalog declaratively by specifying it in a XAML file. The XAML file specifies what kind of module catalog class to create and which modules to add to it. Usually, the .xaml file is added as a resource to your shell project. The module catalog is created by the bootstrapper with a call to the **CreateFromXaml** method. From a technical perspective, this approach is very similar to defining the **ModuleCatalog** in code because the XAML file simply defines a hierarchy of objects to be instantiated.

The following code example shows a XAML file specifying a module catalog.

XAML

```
<!-- ModulesCatalog.xaml -->
<Modularity:ModuleCatalog
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        xmlns:Modularity="clr-
namespace:Microsoft.Practices.Prism.Modularity;assembly=Microsoft.Practices.Prism">
    <Modularity:ModuleInfoGroup
Ref="file:///DirectoryModules/ModularityWithMef.Desktop.ModuleB.dll"
InitializationMode="WhenAvailable">
    <Modularity:ModuleInfo ModuleName="ModuleB"
ModuleType="ModularityWithMef.Desktop.ModuleB, ModularityWithMef.Desktop.ModuleB,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    </Modularity:ModuleInfoGroup>
    <Modularity:ModuleInfoGroup InitializationMode="OnDemand">
        <Modularity:ModuleInfo Ref="file:///ModularityWithMef.Desktop.ModuleE.dll"
ModuleName="ModuleE" ModuleType="ModularityWithMef.Desktop.ModuleE,
ModularityWithMef.Desktop.ModuleE, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null" />
        <Modularity:ModuleInfo Ref="file:///ModularityWithMef.Desktop.ModuleF.dll"
ModuleName="ModuleF" ModuleType="ModularityWithMef.Desktop.ModuleF,
ModularityWithMef.Desktop.ModuleF, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null">
            <Modularity:ModuleInfo.DependsOn>
                <sys:String>ModuleE</sys:String>
            </Modularity:ModuleInfo.DependsOn>
        </Modularity:ModuleInfo>
    </Modularity:ModuleInfoGroup>

    <!-- Module info without a group -->
    <Modularity:ModuleInfo
Ref="file:///DirectoryModules/ModularityWithMef.Desktop.ModuleD.dll"
```

```



```

Note: **ModuleInfoGroups** provide a convenient way to group modules that are in the same assembly, are initialized in the same way, or only have dependencies on modules in the same group.

Dependencies between modules can be defined within modules in the same **ModuleInfoGroup**; however, you cannot define dependencies between modules in different **ModuleInfoGroups**.

Putting modules inside module groups is optional. The properties that are set for a group will be applied to all its contained modules. Note that modules can also be registered without being inside a group.

In your application's **Bootstrapper** class, you need to specify that the XAML file is the source for your **ModuleCatalog**, as shown in the following code.

C#

```

protected override IModuleCatalog CreateModuleCatalog()
{
    return ModuleCatalog.CreateFromXaml(new
Uri("/MyProject;component/ModulesCatalog.xaml",
UriKind.Relative));
}
```

Registering Modules Using a Configuration File

In WPF, it is possible to specify the module information in the App.config file. The advantage of this approach is that this file is not compiled into the application. This makes it very easy to add or remove modules at run time without recompiling the application.

The following code example shows a configuration file specifying a module catalog. If you want the module to automatically load, set **startupLoaded="true"**.

XML

```

<!-- ModularityWithUnity.Desktop\app.config -->
<?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="ModularityWithUnity.Desktop.ModuleE.dll"
moduleType="ModularityWithUnity.Desktop.ModuleE, ModularityWithUnity.Desktop.ModuleE,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleE"
startupLoaded="false" />
```

```

<module assemblyFile="ModularityWithUnity.Desktop.ModuleF.dll"
moduleType="ModularityWithUnity.Desktop.ModuleF, ModularityWithUnity.Desktop.ModuleF,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleF"
startupLoaded="false">
    <dependencies>
        <dependency moduleName="ModuleE"/>
    </dependencies>
</module>
</modules>
</configuration>

```

Note: Even if your assemblies are in the global assembly cache or in the same folder as the application, the **assemblyFile** attribute is required. The attribute is used to map the **moduleType** to the correct **IModuleTypeLoader** to use.

In your application's **Bootstrapper** class, you need to specify that the configuration file is the source for your **ModuleCatalog**. To do this, you use the **ConfigurationModuleCatalog** class, as shown in the following code.

C#

```

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

```

Note: You can still add modules to a **ConfigurationModuleCatalog** in code. You can use this, for example, to make sure that the modules that your application absolutely needs to function are defined in the catalog.

Discovering Modules in a Directory

The Prism **DirectoryModuleCatalog** class allows you to specify a local directory as a module catalog in WPF. This module catalog will scan the specified folder and search for assemblies that define the modules for your application. To use this approach, you will need to use declarative attributes on your module classes to specify the module name and any dependencies that they have. The following code example shows a module catalog that is populated by discovering assemblies in a directory.

C#

```

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

```

Loading Modules

After the **ModuleCatalog** is populated, the modules are ready to be loaded and initialized. Module loading means that the module assembly is transferred from disk into memory. The **ModuleManager** is responsible for coordinating the loading and initialization process.

Initializing Modules

After the modules load, they are initialized. This means an instance of the module class is created and its **Initialize** method is called. Initialization is the place to integrate the module with the application. Consider the following possibilities for module initialization:

- **Register the module's views with the application.** If your module is participating in user interface (UI) composition using view discovery or view injection, your module will need to associate its views or view models with the appropriate region name. This allows views to show up dynamically on menus, toolbars, or other visual regions within the application.
- **Subscribe to application level events or services.** Often, applications expose application-specific services and/or events that your module is interested in. Use the **Initialize** method to add the module's functionality to those application-level events and services.

For example, the application might raise an event when it is shutting down and your module wants to react to that event. It is also possible that your module must provide some data to an application level service. For example, if you have created a **MenuService** (it is responsible for adding and removing menu items), the module's **Initialize** method is where you would add the correct menu items.

Note: Module instance lifetime is short-lived by default. After the **Initialize** method is called during the loading process, the reference to the module instance is released. If you do not establish a strong reference chain to the module instance, it will be garbage collected.

This behavior may be problematic to debug if you subscribe to events that hold a weak reference to your module, because your module just "disappears" when the garbage collector runs.

- **Register types with a dependency injection container.** If you are using a dependency injection pattern such as Unity or MEF, the module may register types for the application or other modules to use. It may also ask the container to resolve an instance of a type it needs.

Specifying Module Dependencies

Modules may depend on other modules. If Module A depends on Module B, Module B must be initialized before Module A. The **ModuleManager** keeps track of these dependencies and initializes the modules accordingly. Depending on how you defined your module catalog, you can define your module dependencies in code, configuration, or XAML.

Specifying Dependencies in Code

For WPF applications that register modules in code or discover modules by directory, Prism provides declarative attributes to use when creating a module as shown in the following code example.

C#

```
// (when using Unity)
[Module(ModuleName = "ModuleA")]
[ModuleDependency("ModuleD")]
public class ModuleA : IModule
{
    ...
}
```

Specify Dependencies in XAML

The following XAML shows where Module F depends on Module E.

XAML

```
<!-- ModulesCatalog.xaml -->
<Modularity:ModuleInfo Ref="file:///ModularityWithMef.Desktop.ModuleE.dll"
moduleName="ModuleE" moduleType="ModularityWithMef.Desktop.ModuleE,
ModularityWithMef.Desktop.ModuleE, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null">

<Modularity:ModuleInfo Ref="file:///ModularityWithMef.Desktop.ModuleF.dll"
moduleName="ModuleF" moduleType="ModularityWithMef.Desktop.ModuleF,
ModularityWithMef.Desktop.ModuleF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
>

<Modularity:ModuleInfo.DependsOn>
    <sys:String>ModuleE</sys:String>
</Modularity:ModuleInfo.DependsOn>
</Modularity:ModuleInfo>
...
```

Specify Dependencies in Configuration

The following example App.config file shows where Module F depends on Module E.

XML

```
<!-- App.config -->
<modules>
    <module assemblyFile="ModularityWithUnity.Desktop.ModuleE.dll"
moduleType="ModularityWithUnity.Desktop.ModuleE, ModularityWithUnity.Desktop.ModuleE,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleE"
startupLoaded="false" />
    <module assemblyFile="ModularityWithUnity.Desktop.ModuleF.dll"
moduleType="ModularityWithUnity.Desktop.ModuleF, ModularityWithUnity.Desktop.ModuleF,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleF"
startupLoaded="false">
```

```

<dependencies>
    <dependency moduleName="ModuleE" />
</dependencies>
</module>
</modules>

```

Loading Modules on Demand

To load modules on demand, you need to specify that they should be loaded into the module catalog with the **InitializationMode** set to **OnDemand**. After you do that, you need to write the code in your application that requests the module be loaded.

Specifying On-Demand Loading in Code

A module is specified as on-demand using attributes, as shown in the following code example.

C#

```

// Bootstrapper.cs
protected override void ConfigureModuleCatalog()
{
    . .
    Type moduleCType = typeof(ModuleC);
    this.ModuleCatalog.AddModule(new ModuleInfo()
    {
        ModuleName = moduleCType.Name,
        ModuleType = moduleCType.AssemblyQualifiedName,
        InitializationMode = InitializationMode.OnDemand
    });
    . .
}

```

Specifying On-Demand Loading in XAML

You can specify the **InitializationMode.OnDemand** when you define your module catalog in XAML, as shown in the following code example.

XAML

```

<!-- ModulesCatalog.xaml -->
...
<module assemblyFile="ModularityWithUnity.Desktop.ModuleE.dll"
moduleType="ModularityWithUnity.Desktop.ModuleE, ModularityWithUnity.Desktop.ModuleE,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleE"
startupLoaded="false" />
...

```

Specifying On-Demand Loading in Configuration

You can specify the **InitializationMode.OnDemand** when you define your module catalog in the App.config file, as shown in the following code example.

XML

```
<!-- App.config -->
<module assemblyFile="ModularityWithUnity.Desktop.ModuleC.dll"
moduleType="ModularityWithUnity.Desktop.ModuleC, ModularityWithUnity.Desktop.ModuleC,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleC"
startupLoaded="false" />
```

Requesting On-Demand Loading of a Module

After a module is specified as on demand, the application can then ask the module to be loaded. The code that wants to initiate the loading needs to obtain a reference to the **IModuleManager** service registered with the container by the bootstrapper.

C#

```
private void OnLoadModuleCClick(object sender, RoutedEventArgs e)
{
    moduleManager.LoadModule("ModuleC");
}
```

Detecting When a Module Has Been Loaded

The **ModuleManager** service provides an event for applications to track when a module loads or fails to load. You can get a reference to this service through dependency injection of the **IModuleManager** interface.

C#

```
this.moduleManager.LoadModuleCompleted += this.ModuleManager_LoadModuleCompleted;
```

C#

```
void ModuleManager_LoadModuleCompleted(object sender, LoadModuleCompletedEventArgs e)
{
    ...
}
```

To keep the application and modules loosely coupled, the application should avoid using this event to integrate the module with the application. Instead, the module's **Initialize** method should handle integrating with the application.

The **LoadModuleCompletedEventArgs** contains an **IsErrorHandler** property. If a module fails to load and the application wants to prevent the **ModuleManager** from logging the error and throwing an exception, it can set this property to **true**.

Note: After a module is loaded and initialized, the module assembly cannot be unloaded. The module instance reference will not be held by the Prism libraries, so the module class instance may be garbage collected after initialization is complete.

Modules in MEF

This section only highlights the differences if you choose to use MEF as your dependency injection container.

Note: When using MEF, the **MefModuleManager** is used by the **MefBootstrapper**. It extends the **ModuleManager** and implements the **IPartImportsSatisfiedNotification** interface to ensure that the **ModuleCatalog** is updated when new types are imported by MEF.

Registering Modules in Code Using MEF

When using MEF, you can apply the **ModuleExport** attribute to module classes to have MEF automatically discover the types. The following is an example.

C#

```
[ModuleExport(typeof(ModuleB), InitializationMode = InitializationMode.OnDemand)]
public class ModuleB : IModule
{
    ...
}
```

You can also use MEF to discover and load modules using the **AssemblyCatalog** class, which can be used to discover all the exported module classes in an assembly, and the **AggregateCatalog** class, that allows multiple catalogs to be combined into one logical catalog. By default, the Prism **MefBootstrapper** class creates an **AggregateCatalog** instance. You can then override the **ConfigureAggregateCatalog** method to register assemblies, as shown in the following code example.

C#

```
protected override void ConfigureAggregateCatalog()
{
    base.ConfigureAggregateCatalog();
    //Module A is referenced in in the project and directly in code.
    this.AggregateCatalog.Catalogs.Add(
        new AssemblyCatalog(typeof(ModuleA).Assembly));

    this.AggregateCatalog.Catalogs.Add(
        new AssemblyCatalog(typeof(ModuleC).Assembly));
    ...
}
```

The Prism **MefModuleManager** implementation keeps the MEF **AggregateCatalog** and the Prism **ModuleCatalog** synchronized, thereby allowing Prism to discover modules added via the **ModuleCatalog** or the **AggregateCatalog**.

Note: MEF uses **Lazy<T>** extensively to prevent instantiation of exported and imported types until the **Value** property is used.

Discovering Modules in a Directory Using MEF

MEF provides a **DirectoryCatalog** that can be used to inspect a directory for assemblies containing modules (and other MEF exported types). In this case, you override the **ConfigureAggregateCatalog** method to register the directory. This approach is only available in WPF.

To use this approach, you first need to apply the module names and dependencies to your modules using the **ModuleExport** attribute, as shown in the following code example. This allows MEF to import the modules and allows Prism to keep the **ModuleCatalog** updated.

C#

```
protected override void ConfigureAggregateCatalog()
{
    base.ConfigureAggregateCatalog();
    . . .

    DirectoryCatalog catalog = new DirectoryCatalog("DirectoryModules");
    this.AggregateCatalog.Catalogs.Add(catalog);
}
```

Specifying Dependencies in Code Using MEF

For WPF applications using MEF, use the **ModuleExport** attribute, as shown here.

C#

```
// (when using MEF)
[ModuleExport(typeof(ModuleA), DependsOnModuleNames = new string[] { "ModuleD" })]
public class ModuleA : IModule
{
    ...
}
```

Because MEF allows you to discover modules at run time, you may also discover new dependencies between modules at run time. Although you can use MEF alongside the **ModuleCatalog**, it is important to remember that the **ModuleCatalog** validates the dependency chain when it is loaded from XAML or configuration (before any modules are loaded). If a module is listed in the **ModuleCatalog** and then loaded using MEF, the **ModuleCatalog** dependencies will be used, and the **DependsOnModuleNames** attribute will be ignored.

Specifying On-Demand Loading Using MEF

If you are using MEF and the **ModuleExport** attribute for specifying modules and module dependencies, you can use the **InitializationMode** property to specify that a module should be loaded on demand, as shown here.

C#

```
[ModuleExport(typeof(ModuleC), InitializationMode = InitializationMode.OnDemand)]
public class ModuleC : IModule
{
```

}

More Information

For more information about assembly caching, see "[How to: Use Assembly Library Caching](#)" on MSDN.

To learn more about modularity in Prism, see the Modularity with MEF for WPF QuickStart or the Modularity with Unity for WPF QuickStart. For more information about the QuickStarts, see [Modularity QuickStarts](#).

For information about the modularity features that can be extended in the Prism Library, see [Modules](#) in [Extending the Prism Library](#).

5: Implementing the MVVM Pattern

The Model-View-ViewModel (MVVM) pattern helps you to cleanly separate the business and presentation logic of your application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make your application much easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of the application.

Using the MVVM pattern, the UI of the application and the underlying presentation and business logic is separated into three separate classes: the view, which encapsulates the UI and UI logic; the view model, which encapsulates presentation logic and state; and the model, which encapsulates the application's business logic and data.

Prism includes samples and reference implementations that show how to implement the MVVM pattern in a Windows Presentation Foundation (WPF) application. The Prism Library also provides features that can help you implement the pattern in your own applications. These features embody the most common practices for implementing the MVVM pattern and are designed to support testability and to work well with Expression Blend and Visual Studio.

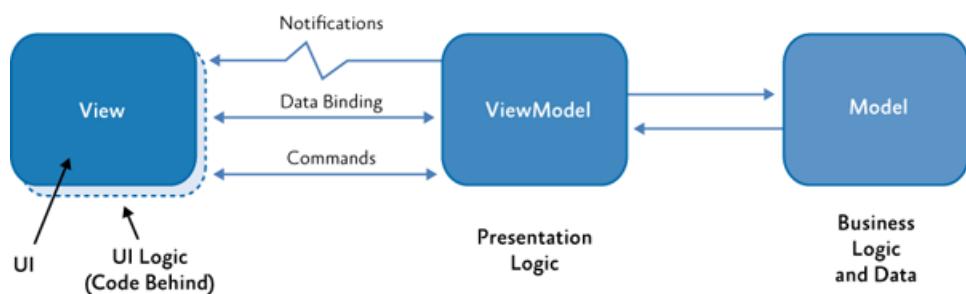
This topic provides an overview of the MVVM pattern and describes how to implement its fundamental characteristics. The topic [Advanced MVVM Scenarios](#) describes how to implement more advanced MVVM scenarios using the Prism Library.

Class Responsibilities and Characteristics

The MVVM pattern is a close variant of the Presentation Model pattern, optimized to leverage some of the core capabilities of WPF , such as data binding, data templates, commands, and behaviors.

In the MVVM pattern, the view encapsulates the UI and any UI logic, the view model encapsulates presentation logic and state, and the model encapsulates business logic and data. The view interacts with the view model through data binding, commands, and change notification events. The view model queries, observes, and coordinates updates to the model, converting, validating, and aggregating data as necessary for display in the view.

The following illustration shows the three MVVM classes and their interaction.



The MVVM classes and their interactions

Like with all separated presentation patterns, the key to using the MVVM pattern effectively lies in understanding the appropriate way to factor your application's code into the correct classes, and in understanding the ways in which these classes interact in various scenarios. The following sections describe the responsibilities and characteristics of each of the classes in the MVVM pattern.

The View Class

The view's responsibility is to define the structure and appearance of what the user sees on the screen. Ideally, the code-behind of a view contains only a constructor that calls the **InitializeComponent** method. In some cases, the code-behind may contain UI logic code that implements visual behavior that is difficult or inefficient to express in Extensible Application Markup Language (XAML), such as complex animations, or when the code needs to directly manipulate visual elements that are part of the view. You should not put any logic code in the view that you need to unit test. Typically, logic code in the view's code-behind will be tested via a UI automation testing approach.

In WPF, data binding expressions in the view are evaluated against its data context. In MVVM, the view's data context is set to the view model. The view model implements properties and commands to which the view can bind and notifies the view of any changes in state through change notification events. There is typically a one-to-one relationship between a view and its view model.

Typically, views are **Control**-derived or **UserControl**-derived classes. However, in some cases, the view may be represented by a data template, which specifies the UI elements to be used to visually represent an object when it is displayed. Using data templates, a visual designer can easily define how a view model will be rendered or can modify its default visual representation without changing the underlying object itself or the behavior of the control that is used to display it.

Data templates can be thought of as views that do not have any code-behind. They are designed to bind to a specific view model type whenever one is required to be displayed in the UI. At run time, the view, as defined by the data template, will be automatically instantiated and its data context set to the corresponding view model.

In WPF, you can associate a data template with a view model type at the application level. WPF will then automatically apply the data template to any view model objects of the specified type whenever they are displayed in the UI. This is known as implicit data templating. The data template can be defined in-line with the control that uses it or in a resource dictionary outside the parent view and declaratively merged into the view's resource dictionary.

To summarize, the view has the following key characteristics:

- The view is a visual element, such as a window, page, user control, or data template. The view defines the controls contained in the view and their visual layout and styling.
- The view references the view model through its **DataContext** property. The controls in the view are data bound to the properties and commands exposed by the view model.

- The view may customize the data binding behavior between the view and the view model. For example, the view may use value converters to format the data to be displayed in the UI, or it may use validation rules to provide additional input data validation to the user.
 - The view defines and handles UI visual behavior, such as animations or transitions that may be triggered from a state change in the view model or via the user's interaction with the UI.
 - The view's code-behind may define UI logic to implement visual behavior that is difficult to express in XAML or that requires direct references to the specific UI controls defined in the view.
-

The View Model Class

The view model in the MVVM pattern encapsulates the presentation logic and data for the view. It has no direct reference to the view or any knowledge about the view's specific implementation or type. The view model implements properties and commands to which the view can data bind and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be rendered.

The view model is responsible for coordinating the view's interaction with any model classes that are required. Typically, there is a one-to many-relationship between the view model and the model classes. The view model may choose to expose model classes directly to the view so that controls in the view can data bind directly to them. In this case, the model classes will need to be designed to support data binding and the relevant change notification events. For more information about this scenario, see the section, [Data Binding](#), later in this topic.

The view model may convert or manipulate model data so that it can be easily consumed by the view. The view model may define additional properties to specifically support the view; these properties would not normally be part of (or cannot be added to) the model. For example, the view model may combine the value of two fields to make it easier for the view to present, or it may calculate the number of characters remaining for input for fields with a maximum length. The view model may also implement data validation logic to ensure data consistency.

The view model may also define logical states the view can use to provide visual changes in the UI. The view may define layout or styling changes that reflect the state of the view model. For example, the view model may define a state that indicates that data is being submitted asynchronously to a web service. The view can display an animation during this state to provide visual feedback to the user.

Typically, the view model will define commands or actions that can be represented in the UI and that the user can invoke. A common example is when the view model provides a **Submit** command that allows the user submit data to a web service or to a data repository. The view may choose to represent that command with a button so that the user can click the button to submit the data. Typically, when the command becomes unavailable, its associated UI representation becomes disabled. Commands

provide a way to encapsulate user actions and to cleanly separate them from their visual representation in the UI.

To summarize, the view model has the following key characteristics:

- The view model is a non-visual class and does not derive from any WPF base class. It encapsulates the presentation logic required to support a use case or user task in the application. The view model is testable independently of the view and the model.
- The view model typically does not directly reference the view. It implements properties and commands to which the view can data bind. It notifies the view of any state changes via change notification events via the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces.
- The view model coordinates the view's interaction with the model. It may convert or manipulate data so that it can be easily consumed by the view and may implement additional properties that may not be present on the model. It may also implement data validation via the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
- The view model may define logical states that the view can represent visually to the user.

View or View Model?

Many times, determining where certain functionality should be implemented is not obvious. The general rule of thumb is: Anything concerned with the specific visual appearance of the UI on the screen and that could be re-styled later (even if you are not currently planning to re-style it) should go into the view; anything that is important to the logical behavior of the application should go into the view model. In addition, because the view model should have no explicit knowledge of the specific visual elements in the view, code to programmatically manipulate visual elements within the view should reside in the view's code-behind or be encapsulated in a behavior. Similarly, code to retrieve or manipulate data items that are to be displayed in the view through data binding should reside in the view model.

For example, the highlight color of the selected item in a list box should be defined in the view, but the list of items to display, and the reference to the selected item itself, should be defined by the view model.

The Model Class

The model in the MVVM pattern encapsulates business logic and data. Business logic is defined as any application logic that is concerned with the retrieval and management of application data and for making sure that any business rules that ensure data consistency and validity are imposed. To maximize re-use opportunities, models should not contain any use case-specific or user task-specific behavior or application logic.

Typically, the model represents the client-side domain model for the application. It can define data structures based on the application's data model and any supporting business and validation logic. The model may also include the code to support data access and caching, though typically a separate data

repository or service is employed for this. Often, the model and data access layer are generated as part of a data access or service strategy, such as the ADO.NET Entity Framework, WCF Data Services, or WCF RIA Services.

Typically, the model implements the facilities that make it easy to bind to the view. This usually means it supports property and collection changed notification through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces. Models classes that represent collections of objects typically derive from the **ObservableCollection<T>** class, which provides an implementation of the **INotifyCollectionChanged** interface.

The model may also support data validation and error reporting through the **IDataErrorInfo** (or **INotifyDataErrorInfo**) interfaces. The **IDataErrorInfo** and **INotifyDataErrorInfo** interfaces allow WPF data binding to be notified when values change so that the UI can be updated. They also enable support for data validation and error reporting in the UI layer.

What if your model classes do not implement the required interfaces?

Sometimes you will need to work with model objects that do not implement the **INotifyPropertyChanged**, **INotifyCollectionChanged**, **IDataErrorInfo**, or **INotifyDataErrorInfo** interfaces. In those cases, the view model may need to wrap the model objects and expose the required properties to the view. The values for these properties will be provided directly by the model objects. The view model will implement the required interfaces for the properties it exposes so that the view can easily data bind to them.

The model has the following key characteristics:

- Model classes are non-visual classes that encapsulate the application's data and business logic. They are responsible for managing the application's data and for ensuring its consistency and validity by encapsulating the required business rules and data validation logic.
- The model classes do not directly reference the view or view model classes and have no dependency on how they are implemented.
- The model classes typically provide property and collection change notification events through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces. This allows them to be easily data bound in the view. Model classes that represent collections of objects typically derive from the **ObservableCollection<T>** class.
- The model classes typically provide data validation and error reporting through either the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
- The model classes are typically used in conjunction with a service or repository that encapsulates data access and caching.

Class Interactions

The MVVM pattern provides a clean separation between your application's user interface, its presentation logic, and its business logic and data by separating each into separate classes. Therefore, when you implement MVVM, it is important to factor in your application's code to the correct classes, as described in the previous section.

Well-designed view, view model, and model classes will not only encapsulate the correct type of code and behavior; they will also be designed so that they can easily interact with each other via data binding, commands, and data validation interfaces.

The interactions between the view and its view model are perhaps the most important to consider, but the interactions between the model classes and the view model are also important. The following sections describe the various patterns for these interactions and describe how to design for them when implementing the MVVM pattern in your applications.

Data Binding

Data binding plays a very important role in the MVVM pattern. WPF provides powerful data binding capabilities. Your view model and (ideally) your model classes should be designed to support data binding so that they can take advantage of these capabilities. Typically, this means that they must implement the correct interfaces.

WPF data binding supports multiple data binding modes. With one-way data binding, UI controls can be bound to a view model so that they reflect the value of the underlying data when the display is rendered. Two-way data binding will also automatically update the underlying data when the user modifies it in the UI.

To ensure that the UI is kept up to date when the data changes in the view model, it should implement the appropriate change notification interface. If it defines properties that can be data bound, it should implement the **INotifyPropertyChanged** interface. If the view model represents a collection, it should implement the **INotifyCollectionChanged** interface or derive from the **ObservableCollection<T>** class that provides an implementation of this interface. Both of these interfaces define an event that is raised whenever the underlying data is changed. Any data bound controls will be automatically updated when these events are raised.

In many cases, a view model will define properties that return objects (and which, in turn, may define properties that return additional objects). WPF data binding supports binding to nested properties via the **Path** property. Therefore, it is very common for a view's view model to return references to other view model or model classes. All view model and model classes accessible to the view should implement the **INotifyPropertyChanged** or **INotifyCollectionChanged** interfaces, as appropriate.

The following sections describe how to implement the required interfaces in order to support data binding within the MVVM pattern.

Implementing **INotifyPropertyChanged**

Implementing the **INotifyPropertyChanged** interface in your view model or model classes allows them to provide change notifications to any data-bound controls in the view when the underlying property value changes. Implementing this interface is straightforward, as shown in the following code example.

C#

```
public class Questionnaire : INotifyPropertyChanged
{
    private string favoriteColor;
    public event PropertyChangedEventHandler PropertyChanged;
    ...
    public string FavoriteColor
    {
        get { return this.favoriteColor; }
        set
        {
            if (value != this.favoriteColor)
            {
                this.favoriteColor = value;

                var handler = this.PropertyChanged;
                if (handler != null)
                {
                    handler(this,
                            new PropertyChangedEventArgs("FavoriteColor"));
                }
            }
        }
    }
}
```

Implementing the **INotifyPropertyChanged** interface on many view model classes can be repetitive and error-prone because of the need to specify the property name in the event argument. The Prism Library provides the **BindableBase** base class from which you can derive your view model classes that implements the **INotifyPropertyChanged** interface in a type-safe manner, as shown here.

C#

```
public abstract class BindableBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    ...
    protected virtual bool SetProperty<T>(ref T storage, T value,
                                              [CallerMemberName] string propertyName = null)
    {...}
    protected void OnPropertyChanged<T>(
        Expression<Func<T>> propertyExpression)
    {...}
}
```

```

protected void OnPropertyChanged(string propertyName)
{...}
}

```

A derived view model class can raise the property change event in the setter by calling the **SetProperty** method. The **SetProperty** method checks whether the backing field is different from the value being set. If different, the backing field is updated and the **PropertyChanged** event is raised.

The following code example shows how to set the property and simultaneously signal the change of another property by using a lambda expression in the **OnPropertyChanged** method. This example comes from the Stock Trader RI. The **TransactionInfo** and **TickerSymbol** properties are related. If the **TransactionInfo** property changes, the **TickerSymbol** will also likely be updated. By calling **OnPropertyChanged** for the **TickerSymbol** property in the setter of the **TransactionInfo** property, two **PropertyChanged** events will be raised, one for **TransactionInfo** and one for **TickerSymbol**.

C#

```

public TransactionInfo TransactionInfo
{
    get { return this.transactionInfo; }
    set
    {
        SetProperty(ref this.transactionInfo, value);
        this.OnPropertyChanged(() => this.TickerSymbol);
    }
}

```

Note: Using a lambda expression in this way involves a small performance cost because the lambda expression has to be evaluated for each call. The benefit is that this approach provides compile-time type safety and refactoring support if you rename a property. Although the performance cost is small and would not normally impact your application, the costs can accrue if you have many change notifications. In this case, you should consider using the non-lambda method overload.

Often, your model or view model will include properties whose values are calculated from other properties in the model or view model. When handling changes to properties, be sure to also raise notification events for any calculated properties.

Implementing INotifyCollectionChanged

Your view model or model class may represent a collection of items, or it may define one or more properties that return a collection of items. In either case, it is likely that you will want to display the collection in an **ItemsControl**, such as a **ListBox**, or in a **DataGrid** control in the view. These controls can be data bound to a view model that represents a collection or to a property that returns a collection via the **ItemSource** property.

XAML

```
<DataGrid ItemsSource="{Binding Path=LineItems}" />
```

To properly support change notification requests, the view model or model class, if it represents a collection, should implement the **INotifyCollectionChanged** interface (in addition to the **INotifyPropertyChanged** interface). If the view model or model class defines a property that returns a reference to a collection, the collection class returned should implement the **INotifyCollectionChanged** interface.

However, implementing the **INotifyCollectionChanged** interface can be challenging because it has to provide notifications when items are added, removed, or changed within the collection. Instead of directly implementing the interface, it is often easier to use or derive from a collection class that already implements it. The **ObservableCollection<T>** class provides an implementation of this interface and is commonly used as either a base class or to implement properties that represent a collection of items.

If you need to provide a collection to the view for data binding, and you do not need to track the user's selection or to support filtering, sorting, or grouping of the items in the collection, you can simply define a property on your view model that returns a reference to the **ObservableCollection<T>** instance.

C#

```
public class OrderViewModel : BindableBase
{
    public OrderViewModel( IOrderService orderService )
    {
        this.LineItems = new ObservableCollection<OrderLineItem>(
            orderService.GetLineItemList() );
    }

    public ObservableCollection<OrderLineItem> LineItems { get; private set; }
}
```

If you obtain a reference to a collection class (for example, from another component or service that does not implement **INotifyCollectionChanged**), you can often wrap that collection in an **ObservableCollection<T>** instance using one of the constructors that take an **IEnumerable<T>** or **List<T>** parameter.

Note: **BindableBase** can be found in the `Microsoft.Practices.Prism.Mvvm` namespace which is located in the `Prism.Mvvm` NuGet package.

Implementing **ICollectionView**

The preceding code example shows how to implement a simple view model property that returns a collection of items that can be displayed via data bound controls in the view. Because the **ObservableCollection<T>** class implements the **INotifyCollectionChanged** interface, the controls in the view will be automatically updated to reflect the current list of items in the collection as items are added or removed.

However, you will often need to more finely control how the collection of items is displayed in the view, or track the user's interaction with the displayed collection of items, from within the view model itself. For example, you may need to allow the collection of items to be filtered or sorted according to

presentation logic implemented in the view model, or you may need to keep track of the currently selected item in the view so that commands implemented in the view model can act on the currently selected item.

WPF supports these scenarios by providing various classes that implement the **ICollectionView** interface. This interface provides properties and methods to allow a collection to be filtered, sorted, or grouped, and allow the currently selected item to be tracked or changed. WPF provides an implementation of this interface using the **ListCollectionView** class.

Collection view classes work by wrapping an underlying collection of items so that they can provide automatic selection tracking and sorting, filtering, and paging for them. An instance of these classes can be created programmatically or declaratively in XAML using the **CollectionViewSource** class.

Note: In WPF, a default collection view will actually be automatically created whenever a control is bound to a collection.

Collection view classes can be used by the view model to keep track of important state information for the underlying collection, while maintaining a clean separation of concerns between the UI in the view and the underlying data in the model. In effect, **CollectionViews** are view models that are designed specifically to support collections.

Therefore, if you need to implement filtering, sorting, grouping, or selection tracking of items in the collection from within your view model, your view model should create an instance of a collection view class for each collection to be exposed to the view. You can then subscribe to selection changed events, such as the **CurrentChanged** event, or control filtering, sorting, or grouping using the methods provided by the collection view class from within your view model.

The view model should implement a read-only property that returns an **ICollectionView** reference so that controls in the view can data bind to the collection view object and interact with it. All WPF controls that derive from the **ItemsControl** base class can automatically interact with **ICollectionView** classes.

The following code example shows the use of the **ListCollectionView** in WPF to keep track of the currently selected customer.

C#

```
public class MyViewModel : BindableBase
{
    public ICollectionView Customers { get; private set; }

    public MyViewModel( ObservableCollection<Customer> customers )
    {
        // Initialize the CollectionView for the underlying model
        // and track the current selection.
        Customers = new ListCollectionView( customers );

        Customers.CurrentChanged += SelectedItemChanged;
    }
}
```

```

private void SelectedItemChanged( object sender, EventArgs e )
{
    Customer current = Customers.SelectedItem as Customer;
    ...
}
...
}

```

In the view, you can then bind an **ItemsControl**, such as a **ListBox**, to the **Customers** property on the view model via its **ItemsSource** property, as shown here.

XAML

```

<ListBox ItemsSource="{Binding Path=Customers}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding Path=Name}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

```

When the user selects a customer in the UI, the view model will be informed so that it can apply the commands that relate to the currently selected customer. The view model can also programmatically change the current selection in the UI by calling methods on the collection view object, as shown in the following code example.

C#

```
Customers.MoveCurrentToNext();
```

When the selection changes in the collection view, the UI automatically updates to visually represent the selected state of the item.

Commands

In addition to providing access to the data to be displayed or edited in the view, the view model will likely define one or more actions or operations that can be performed by the user. In WPF, actions or operations that the user can perform through the UI are typically defined as commands. Commands provide a convenient way to represent actions or operations that can be easily bound to controls in the UI. They encapsulate the actual code that implements the action or operation and help to keep it decoupled from its actual visual representation in the view.

Commands can be visually represented and invoked in many different ways by the user as they interact with the view. In most cases, they are invoked as a result of a mouse click, but they can also be invoked as a result of shortcut key presses, touch gestures, or any other input events. Controls in the view are data bound to the view model's commands so that the user can invoke them using whatever input event or gesture the control defines. Interaction between the UI controls in the view and the command can be

two-way. In this case, the command can be invoked as the user interacts with the UI, and the UI can be automatically enabled or disabled as the underlying command becomes enabled or disabled.

The view model can implement commands as either a **Command Method** or as a **Command Object** (an object that implements the **ICommand** interface). In either case, the view's interaction with the command can be defined declaratively without requiring complex event handling code in the view's code-behind file. For example, certain controls in WPF inherently support commands and provide a **Command** property that can be data bound to an **ICommand** object provided by the view model. In other cases, a command behavior can be used to associate a control with a command method or command object provided by the view model.

Note: Behaviors are a powerful and flexible extensibility mechanism that can be used to encapsulate interaction logic and behavior that can then be declaratively associated with controls in the view. Command behaviors can be used to associate command objects or methods with controls that were not specifically designed to interact with commands.

The following sections describe how to implement commands in your view, as command methods or as command objects, and how to associate them with controls in the view.

Implementing a Task-Based Delegate Command

There are a number of scenarios where the command calls code with long running transactions that cannot block the UI thread. For these scenarios you should use the **FromAsyncHandler** method of the **DelegateCommand** class, which creates a new instance of the **DelegateCommand** from an async handler method.

C#

```
// DelegateCommand.cs
public static DelegateCommand FromAsyncHandler(Func<Task> executeMethod, Func<bool>
canExecuteMethod)
{
    return new DelegateCommand(executeMethod, canExecuteMethod);
}
```

For example, the following code shows how a **DelegateCommand** instance, which represents a sign in command, is constructed by specifying delegates to the **SignInAsync** and **CanSignIn** view model methods. The command is then exposed to the view through a read-only property that returns a reference to an [**ICommand**](#).

C#

```
// SignInFlyoutViewModel.cs
public DelegateCommand SignInCommand { get; private set; }

...
SignInCommand = DelegateCommand.FromAsyncHandler(SignInAsync, CanSignIn);
```

Implementing Command Objects

A command object is an object that implements the **ICommand** interface. This interface defines an **Execute** method, which encapsulates the operation itself, and a **CanExecute** method, which indicates whether the command can be invoked at a particular time. Both of these methods take a single argument as the parameter for the command. The encapsulation of the implementation logic for an operation in a command object means it can be more easily unit tested and maintained.

Implementing the **ICommand** interface is straightforward. However, there are a number of implementations of this interface that you can readily use in your application. For example, you can use the **ActionCommand** class from the Blend for Visual Studio SDK or the **DelegateCommand** class provided by Prism.

Note: **DelegateCommand** can be found in the Microsoft.Practices.Prism.Mvvm namespace which is located in the Prism.Mvvm NuGet package.

The Prism **DelegateCommand** class encapsulates two delegates that each reference a method implemented within your view model class. It inherits from the **DelegateCommandBase** class, which implements the **ICommand** interface's **Execute** and **CanExecute** methods by invoking these delegates. You specify the delegates to your view model methods in the **DelegateCommand** class constructor, which is defined as follows.

C#

```
// DelegateCommand.cs
public class DelegateCommand<T> : DelegateCommandBase
{
    public DelegateCommand(Action<T> executeMethod, Func<T, bool> canExecuteMethod) :
        base((o) => executeMethod((T)o), (o) => canExecuteMethod((T)o))
    {
        ...
    }
}
```

For example, the following code example shows how a **DelegateCommand** instance, which represents a **Submit** command, is constructed by specifying delegates to the **OnSubmit** and **CanSubmit** view model methods. The command is then exposed to the view via a read-only property that returns a reference to an **ICommand**.

C#

```
public class QuestionnaireViewModel
{
    public QuestionnaireViewModel()
    {
        this.SubmitCommand = new DelegateCommand<object>(
            this.OnSubmit, this.CanSubmit );
    }
}
```

```

public ICommand SubmitCommand { get; private set; }

private void OnSubmit(object arg) { ... }
private bool CanSubmit(object arg) { return true; }
}

```

When the **Execute** method is called on the **DelegateCommand** object, it simply forwards the call to the method in your view model class via the delegate that you specified in the constructor. Similarly, when the **CanExecute** method is called, the corresponding method in your view model class is called. The delegate to the **CanExecute** method in the constructor is optional. If a delegate is not specified, **DelegateCommand** will always return **true** for **CanExecute**.

The **DelegateCommand** class is a generic type. The type argument specifies the type of the command parameter passed to the **Execute** and **CanExecute** methods. In the preceding example, the command parameter is of type **object**. A non-generic version of the **DelegateCommand** class is also provided by Prism for use when a command parameter is not required.

The view model can indicate a change in the command's **CanExecute** status by calling the **RaiseCanExecuteChanged** method on the **DelegateCommand** object. This causes the **CanExecuteChanged** event to be raised. Any controls in the UI that are bound to the command will update their enabled status to reflect the availability of the bound command.

Other implementations of the **ICommand** interface are available. The **ActionCommand** class provided by the Expression Blend SDK is similar to Prism's **DelegateCommand** class described earlier, but it supports only a single **Execute** method delegate. Prism also provides the **CompositeCommand** class, which allows **DelegateCommands** to be grouped together for execution. For more information about using the **CompositeCommand** class, see "[Composite Commands](#)" in "[Advanced MVVM Scenarios](#)".

[Invoking Command Objects from the View](#)

There are a number of ways in which a control in the view can be associated with a command object proffered by the view model. Certain WPF controls, notably **ButtonBase** derived controls, such as **Button** or **RadioButton**, and **Hyperlink**, or **MenuItem** derived controls, can be easily data bound to a command object through the **Command** property. WPF also supports binding view model **ICommand** to a **KeyGesture**.

XAML

```
<Button Command="{Binding Path=SubmitCommand}" CommandParameter="SubmitOrder"/>
```

A command parameter can also be optionally defined using the **CommandParameter** property. The type of the expected argument is specified in the **Execute** and **CanExecute** target methods. The control will automatically invoke the target command when the user interacts with that control, and the command parameter, if provided, will be passed as the argument to the command's **Execute** method. In the preceding example, the button will automatically invoke the **SubmitCommand** when it is clicked. Additionally, if a **CanExecute** handler is specified, the button will be automatically disabled if **CanExecute** returns **false**, and it will be enabled if it returns **true**.

An alternative approach is to use Blend for Visual Studio 2013 interaction triggers and **InvokeCommandAction** behavior. For more information on **InvokeCommandAction** behavior and associating commands to events see "[Interaction Triggers and Commands](#)" in "[Advanced MVVM Scenarios](#)."

Data Validation and Error Reporting

Your view model or model will often be required to perform data validation and to signal any data validation errors to the view so that the user can act to correct them.

WPF provides support for managing data validation errors that occur when changing individual properties that are bound to controls in the view. For single properties that are data-bound to a control, the view model or model can signal a data validation error within the property setter by rejecting an incoming bad value and throwing an exception. If the **ValidatesOnExceptions** property on the data binding is **true**, the data binding engine in WPF will handle the exception and display a visual cue to the user that there is a data validation error.

However, throwing exceptions with properties in this way should be avoided where possible. An alternative approach is to implement the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces on your view model or model classes. These interfaces allow your view model or model to perform data validation for one or more property values and to return an error message to the view so that the user can be notified of the error.

Implementing IDataErrorInfo

The **IDataErrorInfo** interface provides basic support for property data validation and error reporting. It defines two read-only properties: an indexer property, with the property name as the indexer argument, and an **Error** property. Both properties return a string value.

The indexer property allows the view model or model class to provide an error message specific to the named property. An empty string or null return value indicates to the view that the changed property value is valid. The **Error** property allows the view model or model class to provide an error message for the entire object. Note, however, that this property is not currently called by the WPF data binding engine.

The **IDataErrorInfo** indexer property is accessed when a data-bound property is first displayed, and whenever it is subsequently changed. Because the indexer property is called for all properties that change, you should be careful to ensure that data validation is as fast and as efficient as possible.

When binding controls in the view to properties you want to validate through the **IDataErrorInfo** interface, set the **ValidatesOnDataErrors** property on the data binding to **true**. This will ensure that the data binding engine will request error information for the data-bound property.

XAML

```
<TextBox
Text="{Binding Path=CurrentEmployee.Name, Mode=TwoWay, ValidatesOnDataErrors=True,
NotifyOnValidationError=True }"
```

/>

Implementing INotifyDataErrorInfo

The **INotifyDataErrorInfo** interface is more flexible than the **IDataErrorInfo** interface. It supports multiple errors for a property, asynchronous data validation, and the ability to notify the view if the error state changes for an object.

The **INotifyDataErrorInfo** interface defines a **HasErrors** property, which allows the view model to indicate whether an error (or multiple errors) for any properties exist, and a **GetErrors** method, which allows the view model to return a list of error messages for a particular property.

The **INotifyDataErrorInfo** interface also defines an **ErrorsChanged** event. This supports asynchronous validation scenarios by allowing the view or view model to signal a change in error state for a particular property through the **ErrorsChanged** event. Property values can be changed in a number of ways, and not just via data binding—for example, as a result of a web service call or background calculation. The **ErrorsChanged** event allows the view model to inform the view of an error once a data validation error has been identified.

To support **INotifyDataErrorInfo**, you will need to maintain a list of errors for each property. The Model-View-ViewModel Reference Implementation (MVVM RI) demonstrates one way to do this using an **ErrorsContainer** collection class that tracks all the validation errors in the object. It also raises notification events if the error list changes. The following code example shows a **DomainObject** (a root model object) and shows an example implementation of **INotifyDataErrorInfo** using the **ErrorsContainer** class.

C#

```
public abstract class DomainObject : INotifyPropertyChanged,
                                      INotifyDataErrorInfo
{
    private ErrorsContainer<ValidationResult> errorsContainer =
        new ErrorsContainer<ValidationResult>(
            pn => this.RaiseErrorsChanged( pn ) );

    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    public bool HasErrors
    {
        get { return this.errorsContainer.HasErrors; }
    }

    public IEnumerable GetErrors( string propertyName )
    {
        return this.errorsContainer.GetErrors( propertyName );
    }

    protected void RaiseErrorsChanged( string propertyName )
    {
```

```

    var handler = this.ErrorsChanged;
    if (handler != null)
    {
        handler(this, new DataErrorsChangedEventArgs(propertyName));
    }
}
...
}

```

Construction and Wire-Up

The MVVM pattern helps you to cleanly separate your UI from your presentation and business logic and data, so implementing the right code in the right class is an important first step in using the MVVM pattern effectively. Managing the interactions between the view and view model classes through data binding and commands are also important aspects to consider. The next step is to consider how the view, view model, and model classes are instantiated and associated with each other at run time.

Note: Choosing an appropriate strategy to manage this step is especially important if you are using a dependency injection container in your application. The Managed Extensibility Framework (MEF) and the Unity Application Block (Unity) both provide the ability to specify dependencies between the view, view model, and model classes and to have them fulfilled by the container. For more advanced scenarios, see [Advanced MVVM Scenarios](#).

Typically, there is a one-to-one relationship between a view and its view model. The view and view model are loosely coupled via the view's data context property; this allows visual elements and behaviors in the view to be data bound to properties, commands, and methods on the view model. You will need to decide how to manage the instantiation of the view and view model classes and their association via the **DataContext** property at run time.

Care must also be taken when constructing and connecting the view and view model to ensure that loose coupling is maintained. As noted in the previous section, the view model should ideally not depend on any specific implementation of a view. Similarly, the view should ideally not depend on any specific implementation of a view model.

Note: However, it should be noted that the view will *implicitly* depend on specific properties, commands, and methods on the view model because of the data bindings it defines. If the view model does not implement the required property, command, or method, a run-time exception will be generated by the data binding engine, which will be displayed in the Visual Studio output window during debugging.

There are multiple ways the view and the view model can be constructed and associated at run time. The most appropriate approach for your application will largely depend on whether you create the view or the view model first, and whether you do this programmatically or declaratively. The following sections describe common ways in which the view and view model classes can be created and associated with each other at run time.

Creating the View Model Using XAML

Perhaps the simplest approach is for the view to declaratively instantiate its corresponding view model in XAML. When the view is constructed, the corresponding view model object will also be constructed. You can also specify in XAML that the view model be set as the view's data context.

XAML

```
<UserControl.DataContext>
    <my:MyViewModel/>
</UserControl.DataContext>
```

When this view is created, an instance of the **MyViewModel** is automatically constructed and set as the view's data context. This approach requires your view model to have a default (parameter-less) constructor.

The declarative construction and assignment of the view model by the view has the advantage that it is simple and works well in design-time tools such as Microsoft Expression Blend or Microsoft Visual Studio. The disadvantage of this approach is that the view has knowledge of the corresponding view model type and that the view model type must have a default constructor.

Creating the View Model Programmatically

Another approach is for the view to instantiate its corresponding view model instance programmatically in its constructor. It can then set it as its data context, as shown in the following code example.

C#

```
public MyView()
{
    InitializeComponent();
    this.DataContext = new MyViewModel();
}
```

The programmatic construction and assignment of the view model within the view's code-behind has the advantage that it is simple and works well in design-time tools like Expression Blend or Visual Studio. The disadvantage of this approach is that the view needs to have knowledge of the corresponding view model type and that it requires code in the view's code-behind. Using a dependency injection container, such as Unity or MEF, can help to maintain loose coupling between the view and view model. For more information, see [Managing Dependencies Between Components](#).

Creating the View Model Using a View Model Locator

Another way to create a view model instance and associate it with its view is by using a view model locator.

The Prism view model locator has a **AutoWireViewModel** attached property that when set calls **AutoWireViewModelChanged** method in the **ViewModelLocationProvider** class to resolve the view model for the view. By default it uses a convention based approach.

In the Basic MVVM QuickStart, the `MainWindow.xaml` uses the view model locator to resolve the view model.

XAML

```
...
    prism:ViewModelLocator.AutoWireViewModel="True">
```

Prism's **ViewModelLocator** class has an attached property, **AutoWireViewModel** that when set to true will try to locate the view model of the view, and then set the view's data context to an instance of the view model. To locate the corresponding view model, the **ViewModelLocationProvider** first attempts to resolve the view model from any mappings that may have been registered by the **Register** method of the **ViewModelLocationProvider** class. If the view model cannot be resolved using this approach, for instance if the mapping wasn't created, the **ViewModelLocationProvider** falls back to a convention-based approach to resolve the correct view model type. This convention assumes that view models are in the same assembly as the view types, that view models are in a **.ViewModels** child namespace, that views are in a **.Views** child namespace, and that view model names correspond with view names and end with "ViewModel.". For instructions on how to change Prism's View Model Locator convention, see [Extending the Prism Library](#).

Note: **ViewModelLocationProvider** can be found in the **Microsoft.Practices.Prism.Mvvm** assembly and **ViewModelLocator** can be found in the **Microsoft.Practices.Prism.Mvvm.Desktop** assembly which is located in the **Prism.Mvvm** NuGet package.

Creating a View Defined as a Data Template

A view can be defined as a data template and associated with a view model type. Data templates can be defined as resources, or they can be defined inline within the control that will display the view model. The "content" of the control is the view model instance, and the data template is used to visually represent it. WPF will automatically instantiate the data template and set its data context to the view model instance at run time. This technique is an example of a situation in which the view model is instantiated first, followed by the creation of the view.

Data templates are flexible and lightweight. The UI designer can use them to easily define the visual representation of a view model without requiring any complex code. Data templates are restricted to views that do not require any UI logic (code-behind). Microsoft Blend for Visual Studio 2013 can be used to visually design and edit data templates.

The following example shows an **ItemsControl** that is bound to a list of customers. Each customer object in the underlying collection is a view model instance. The view for the customer is defined by an inline data template. In the following example, the view for each customer view model consists of a **StackPanel** with a label and text box control bound to the **Name** property on the view model.

XAML

```
<ItemsControl ItemsSource="{Binding Customers}">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock VerticalAlignment="Center" Text="Customer Name: " />
                <TextBox Text="{Binding Name}" />
            
```

```

        </StackPanel>
    </DataTemplate>
</ItemsControl.ItemTemplate>
</ItemsControl>

```

You can also define a data template as a resource. The following example shows the data template defined a resource and applied to a content control via the **StaticResource** markup extension.

XAML

```

<UserControl ...>
    <UserControl.Resources>
        <DataTemplate x:Key="CustomerViewTemplate">
            <local:CustomerContactView />
        </DataTemplate>
    </UserControl.Resources>

    <Grid>
        <ContentControl Content="{Binding Customer}"
                       ContentTemplate="{StaticResource CustomerViewTemplate}" />
    </Grid>
</UserControl>

```

Here, the data template wraps a concrete view type. This allows the view to define code-behind behavior. In this way, the data template mechanism can be used to externally provide the association between the view and the view model. Although the preceding example shows the template in the **UserControl** resources, it would often be placed in application's resources for reuse.

Key Decisions

When you choose to use the MVVM pattern to construct your application, you will have to make certain design decisions that will be difficult to change later on. Generally, these decisions are application-wide and their consistent use throughout the application will improve developer and designer productivity. The following summarizes the most important decisions when implementing the MVVM pattern:

- Decide on the approach to view and view model construction you will use. You need to decide if your application constructs the views or the view models first and whether to use a dependency injection container, such as Unity or MEF. You will usually want this to be consistent application-wide. For more information, see the section, [Construction and Wire-Up](#), in this topic and the section [Advanced Construction and Wire-Up](#), in [Advanced MVVM Scenarios](#).
- Decide if you will expose commands from your view models as command methods or command objects. Command methods are simple to expose and can be invoked through behaviors in the view. Command objects can neatly encapsulate the command and enabled/disabled logic and can be invoked through behaviors or via the **Command** property on **ButtonBase**-derived controls. To make it easier on your developers and designers, it is a good idea to make this an application-wide choice. For more information, see the section, [Commands](#), in this topic.

- Decide how your view models and models will report errors to the view. Your models can either support **IDataErrorInfo** or **INotifyDataErrorInfo**. Not all models may need to report error information, but for those that do, it is preferable to have a consistent approach for your developers. For more information, see the section, [Data Validation and Error Reporting](#), in this topic.
- Decide whether Microsoft Blend for Visual Studio 2013 design-time data support is important to your team. If you will use Blend to design and maintain your UI and want to see design time data, make sure that your views and view models offer constructors that do not have parameters and that your views provide a design-time data context. Alternatively, consider using the design-time features provided by Microsoft Blend for Visual Studio 2013 using design-time attributes such as **d:DataContext** and **d:DesignSource**. For more information, see [Guidelines for Creating Designer Friendly Views](#) in [Composing the User Interface](#).

More Information

For more information about data binding in WPF, see [Data Binding](#) on MSDN.

For more information about binding to collections in WPF, see [Binding to Collections](#) in [Data Binding Overview](#) on MSDN.

For more information about the Presentation Model pattern, see [Presentation Model](#) on Martin Fowler's website.

For more information about data templates, see [Data Templating Overview](#) on MSDN.

For more information about MEF, see [Managed Extensibility Framework Overview](#) on MSDN.

For more information about Unity, see [Unity Application Block](#) on MSDN.

For more information about **DelegateCommand** and **CompositeCommand**, see [Communicating Between Loosely Coupled Components](#).

For more information about using MVVM in Windows Store Apps see [Using the Model-View-ViewModel \(MVVM\) pattern in a Windows Store business app using C#, XAML, and Prism](#).

6: Advanced MVVM Scenarios

The previous topic described how to implement the basic elements of the Model-View-ViewModel (MVVM) pattern by separating your application's user interface (UI), presentation logic, and business logic into three separate classes (the view, view model, and model), implementing the interactions between those classes (through data binding, commands, and data validation interfaces), and by implementing a strategy to handle construction and wire-up. This topic describes some sophisticated scenarios and describes how the MVVM pattern can support them. The next section describes how commands can be chained together or associated with child views and how they can be extended to support custom requirements. The following sections then describe how to handle asynchronous data requests and subsequent UI interactions and how to handle interaction requests between the view and the view model.

The section, [Advanced Construction and Wire-Up](#), provides guidance on handling construction and wire-up when using a dependency injection container, such as the Unity Application Block (Unity), or when using the Managed Extensibility Framework (MEF). The final section describes how you can test MVVM applications by providing guidance on unit testing your application's view model and model classes, and on testing behaviors.

Commands

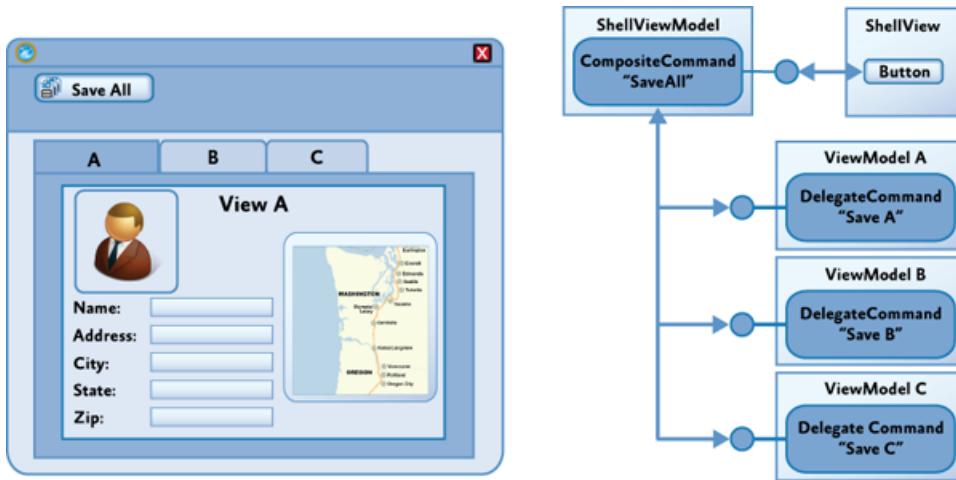
Commands provide a way to separate the command's implementation logic from its UI representation. Data binding or behaviors provide a way to declaratively associate elements in the view with commands proffered by the view model. The section, [Commands](#) in [Implementing the MVVM Pattern](#), described how commands can be implemented as command objects or command methods on the view model, and how they can be invoked from controls in the view by using the built-in **Command** property provided by certain controls.

WPF Routed Commands: It should be noted that commands implemented as command objects or command methods in the MVVM pattern differ somewhat from WPF's built-in implementation of commands named routed commands. WPF routed commands deliver command messages by routing them through elements in the UI tree (specifically the [logical tree](#)). Therefore, command messages are routed up or down the UI tree from the focused element or to an explicitly specified target element; by default, they are not routed to components outside of the UI tree, such as the view model associated with the view. However, WPF-routed commands can use a command handler defined in the view's code-behind to forward the command call to the view model class.

Composite Commands

In many cases, a command defined by a view model will be bound to controls in the associated view so that the user can directly invoke the command from within the view. However, in some cases, you may want to be able to invoke commands on one or more view models from a control in a parent view in the application's UI.

For example, if your application allows the user to edit multiple items at the same time, you may want to allow the user to save all the items using a single command represented by a button in the application's toolbar or ribbon. In this case, the **Save All** command will invoke each of the **Save** commands implemented by the view model instance for each item as shown in the following illustration.



Implementing the SaveAll composite command

Prism supports this scenario through the **CompositeCommand** class.

The **CompositeCommand** class represents a command that is composed from multiple child commands. When the composite command is invoked, each of its child commands is invoked in turn. It is useful in situations where you need to represent a group of commands as a single command in the UI or where you want to invoke multiple commands to implement a logical command.

For example, the **CompositeCommand** class is used in the Stock Trader Reference Implementation (Stock Trader RI) in order to implement the **SubmitAllOrders** command represented by the **Submit All** button in the buy/sell view. When the user clicks the **Submit All** button, each **SubmitCommand** defined by the individual buy/sell transactions is executed.

The **CompositeCommand** class maintains a list of child commands (**DelegateCommand** instances). The **Execute** method of the **CompositeCommand** class simply calls the **Execute** method on each of the child commands in turn. The **CanExecute** method similarly calls the **CanExecute** method of each child command, but if any of the child commands cannot be executed, the **CanExecute** method will return **false**. In other words, by default, a **CompositeCommand** can only be executed when all the child commands can be executed.

Registering and Unregistering Child Commands

Child commands are registered or unregistered using the **RegisterCommand** and **UnregisterCommand** methods. In the Stock Trader RI, for example, the **Submit** and **Cancel** commands for each buy/sell order are registered with the **SubmitAllOrders** and **CancelAllOrders** composite commands, as shown in the following code example (see the **OrdersController** class).

C#

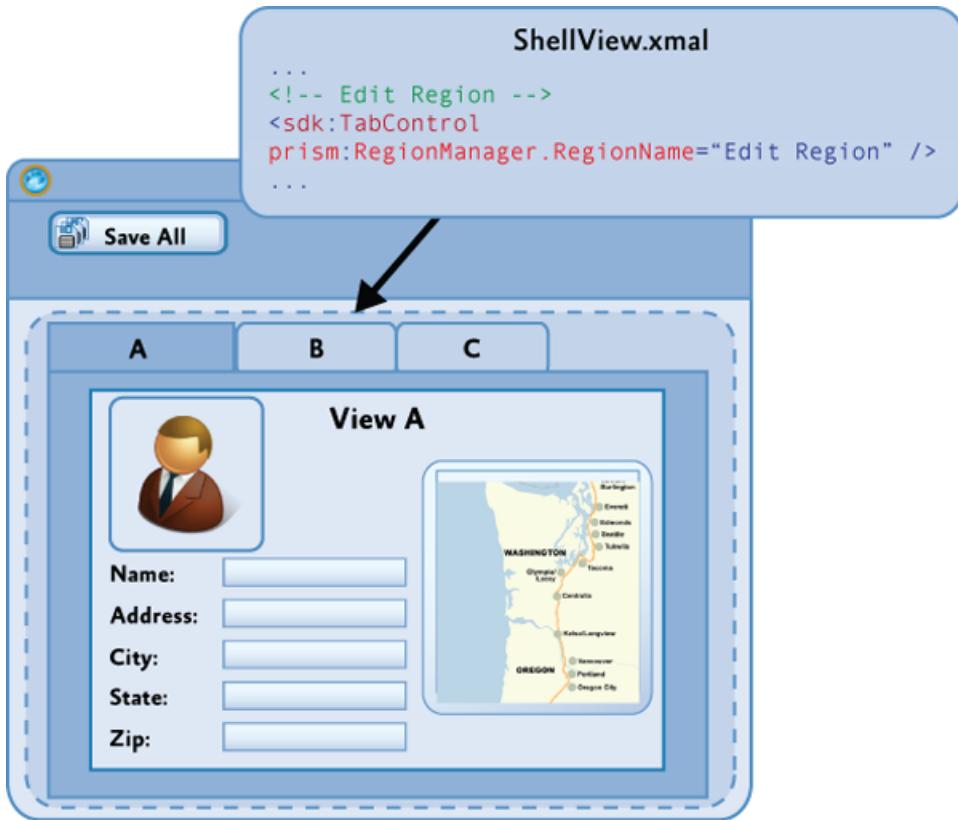
```
// OrdersController.cs
commandProxy.SubmitAllOrdersCommand.RegisterCommand(
    orderCompositeViewModel.SubmitCommand );
commandProxy.CancelAllOrdersCommand.RegisterCommand(
    orderCompositeViewModel.CancelCommand );
```

Note: The preceding **commandProxy** object provides instance access to the **Submit** and **Cancel** composite commands, which are defined statically. For more information, see the class file StockTraderRICommands.cs.

Executing Commands on Active Child Views

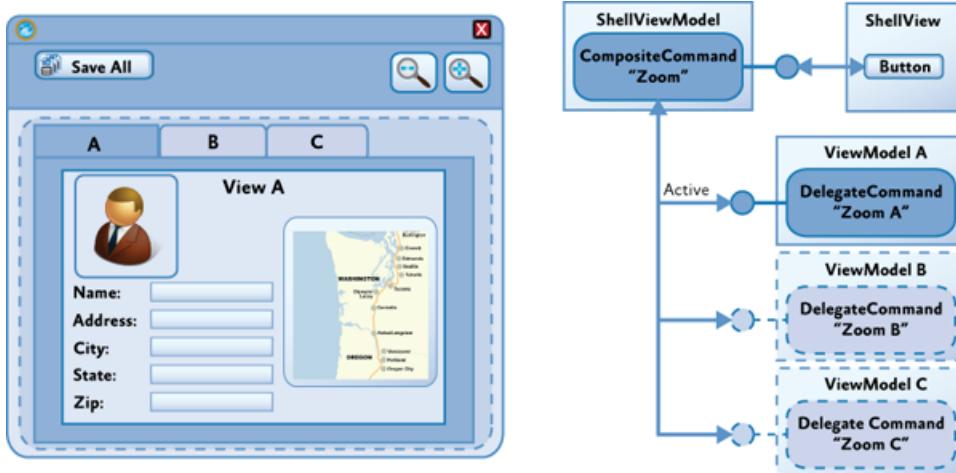
Often, your application will need to display a collection of child views within the application's UI, where each child view will have a corresponding view model that, in turn, may implement one or more commands. Composite commands can be used to represent the commands implemented by child views within the application's UI and help to coordinate how they are invoked from within the parent view. To support these scenarios, the Prism **CompositeCommand** and **DelegateCommand** classes have been designed to work with Prism regions.

Prism regions (described in section, [Regions](#) in [Composing the User Interface](#)) provide a way for child views to be associated with logical placeholders in the application's UI. They are often used to decouple the specific layout of child views from their logical placeholder and its position in the UI. Regions are based on named placeholders that are attached to specific layout controls. The following illustration shows an example where each child view has been added to the region named **EditRegion**, and the UI designer has chosen to use a **Tab** control to lay out the views within that region.



Defining the EditRegion using a Tab control

Composite commands at the parent view level will often be used to coordinate how commands at the child view level are invoked. In some cases, you will want the commands for all shown views to be executed, as in the **Save All** command example described earlier. In other cases, you will want the command to be executed only on the active view. In this case, the composite command will execute the child commands only on views that are deemed to be active; it will not execute the child commands on views that are not active. For example, you may want to implement a **Zoom** command on the application's toolbar or ribbon that causes only the currently active item to be zoomed, as shown in the following diagram.



Defining the EditRegion using a Tab control

To support this scenario, Prism provides the **IActiveAware** interface. The **IActiveAware** interface defines an **IsActive** property that returns **true** when the implementer is active, and an **IsActiveChanged** event that is raised whenever the active state is changed.

You can implement the **IActiveAware** interface on child views or view models. It is primarily used to track the active state of a child view within a region. Whether or not a view is active is determined by the region adapter that coordinates the views within the specific region control. For the **TabControl** shown earlier, there is a region adapter that sets the view in the currently selected tab as **active**, for example.

The **DelegateCommand** class also implements the **IActiveAware** interface. The **CompositeCommand** can be configured to evaluate the active status of child **DelegateCommands** (in addition to the **CanExecute** status) by specifying **true** for the **monitorCommandActivity** parameter in the constructor. When this parameter is set to **true**, the **CompositeCommand** class will consider each child **DelegateCommand**'s active status when determining the return value for the **CanExecute** method and when executing child commands within the **Execute** method.

When the **monitorCommandActivity** parameter is **true**, the **CompositeCommand** class exhibits the following behavior:

- **CanExecute**. Returns **true** only when all active commands can be executed. Child commands that are inactive will not be considered at all.
- **Execute**. Executes all active commands. Child commands that are inactive will not be considered at all.

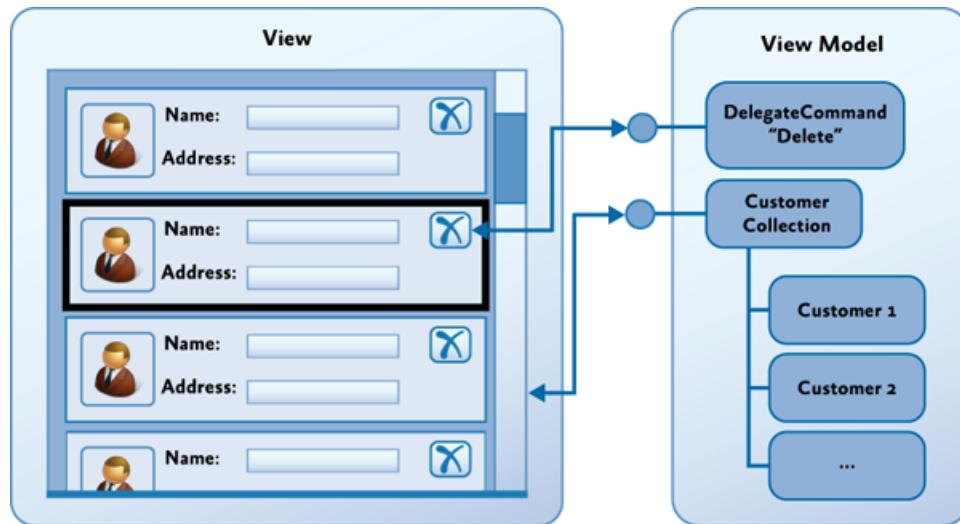
You can use this functionality to implement the example described earlier. By implementing the **IActiveAware** interface on your child view models, you will be notified when your child view becomes active or inactive with the region. When the child view's active status changes, you can update the

active status of the child commands. Then, when the user invokes the **Zoom** composite command, the **Zoom** command on the active child view will be invoked.

Commands Within Collections

Another common scenario you will often encounter when displaying a collection of items in a view is when you need the UI for each item in the collection to be associated with a command at the parent view level (instead of the item level).

For example, in the application shown in the following illustration, the view displays a collection of items in a **ListBox** control, and the data template used to display each item defines a **Delete** button that allows the user to delete individual items from the collection.



Binding commands within collections

Because the view model implements the **Delete** command, the challenge is to wire up the **Delete** button in the UI for each item, to the **Delete** command implemented by the view model. The difficulty arises because the data context for each of the items in the **ListBox** references the item in the collection instead of the parent view model that implements the **Delete** command.

One approach to this problem is to bind the button in the data template to the command in the parent view using the **ElementName** binding property to ensure that the binding is relative to the parent control and not relative to the data template. The following XAML illustrates this technique.

XAML

```

<Grid x:Name="root">
    <ListBox ItemsSource="{Binding Path=Items}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Button Content="{Binding Path=Name}"
                    Command="{Binding ElementName=root, Path=DataContext.DeleteCommand}" />
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>

```

```
</ListBox>
</Grid>
```

The content of button control in the data template is bound to the **Name** property on the item in the collection. However, the command for the button is bound via the root element's data context to the **Delete** command. This allows the button to be bound to the command at the parent view level instead of at the item level. You can use the **CommandParameter** property to specify the item to which the command is to be applied, or you can implement the command to operate on the currently selected item (via a **CollectionView**).

Interaction Triggers and Commands

An alternative approach to commands is to use Blend for Visual Studio 2013 interaction triggers and the **InvokeCommandAction** action.

XAML

```
<Button Content="Submit" IsEnabled="{Binding CanSubmit}">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Click">
            <i:InvokeCommandAction Command="{Binding SubmitCommand}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

This approach can be used for any control to which you can attach an interaction trigger. It is especially useful if you want to attach a command to a control that does not implement the **ICommandSource** interface, or when you want to invoke the command on an event other than the default event. Again, if you need to supply parameters for your command, you can use the **CommandParameter** property.

The following shows how to use the Blend EventTrigger configured to listen to the **ListBox**'s **SelectionChanged** event. When this event occurs, the **SelectedCommand** is invoked by the **InvokeCommandAction**.

XAML

```
<ListBox ItemsSource="{Binding Items}" SelectionMode="Single">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="SelectionChanged">
            <i:InvokeCommandAction Command="{Binding SelectedCommand}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</ListBox>
```

Command-Enabled Controls vs. Behaviors

WPF controls that support commands allow you to declaratively hook up a control to a command. These controls will invoke the specified command when the user interacts with the control in a specific way. For example, for a **Button** control, the command will be invoked when the user clicks the button. This event associated with the command is fixed and cannot be changed.

Behaviors also allow you to hook up a control to a command in a declarative fashion. However, behaviors can be associated with a range of events raised by the control, and they can be used to conditionally invoke an associated command object or a command method in the view model. In other words, behaviors can address many of the same scenarios as command-enabled controls, and they may provide a greater degree of flexibility and control.

You will need to choose when to use command-enabled controls and when to use behaviors, as well as which kind of behavior to use. If you prefer to use a single mechanism to associate controls in the view with functionality in the view model or for consistency, you might consider using behaviors, even for controls that inherently support commands.

If you only need to use command-enabled controls to invoke commands on the view model, and if you are happy with the default events to invoke the command, behaviors may not be required. Similarly, if your developers or UI designers will not be using Blend for Visual Studio 2013, you may favor command-enabled controls (or custom attached behaviors) because of the additional syntax required for Blend behaviors.

Passing EventArgs Parameters to the Command

When you need to invoke a command in response to an event raised by a control located in the view, you can use Prism's **InvokeCommandAction**. Prism's **InvokeCommandAction** differs from the class of the same name in the Blend SDK in two ways. First, the Prism **InvokeCommandAction** updates the enabled state of the associated control based on the return value of the command's **CanExecute** method. Second, the Prism **InvokeCommandAction** uses the **EventArgs** parameter passed to it from the parent trigger, passing it to the associated command if the **CommandParameter** is not set.

Sometimes you need to pass a parameter to the command that comes from the parent trigger, such as the **EventArgs** from the **EventTrigger**. In that scenario you cannot use Blend's **InvokeCommandAction** action.

In the following code you can see that Prism's **InvokeCommandAction** has a property called **TriggerParameterPath** that is used to specify the member (possibly nested) of the parameter passed as the command parameter. In the following example, the **AddedItems** property of the **SelectionChanged** EventArgs will be passed to the **SelectedCommand** command.

XAML

```
<ListBox Grid.Row="1" Margin="5" ItemsSource="{Binding Items}"
SelectionMode="Single">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="SelectionChanged">
            <!-- This action will invoke the selected command in the view model and
            pass the parameters of the event to it. -->
            <prism:InvokeCommandAction Command="{Binding SelectedCommand}"
TriggerParameterPath="AddedItems" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
```

```
</ListBox>
```

Handling Asynchronous Interactions

Your view model will often need to interact with services and components within your application that communicate asynchronously instead of synchronously. This is especially true if you interacting with web services or other resources over the network, or if your application uses background tasks to perform calculations or I/O. Performing these operations asynchronously ensures that your application remains responsive which is essential for delivering a good user experience.

When the user initiates an asynchronous request or background task, it is difficult to predict when the response will arrive (or even if it will arrive) and, very often, what thread it will return on. Because the UI can be updated only in the UI thread, you will often need to update the UI by dispatching a request on the UI thread.

Retrieving Data and Interacting with Web Services

When interacting with web services or other remote access technologies, you will often encounter the **IAsyncResult** pattern. In this pattern, instead of invoking a method, such as **GetQuestionnaire**, you use the pair of methods **BeginGetQuestionnaire** and **EndGetQuestionnaire**. To initiate the asynchronous call, you call **BeginGetQuestionnaire**. To get the results or determine if there was an exception when invoking the target method, you call **EndGetQuestionnaire** when the call is complete.

To determine when to call **EndGetQuestionnaire**, you can either poll for completion or (preferably) specify a callback during the call to **BeginGetQuestionnaire**. With the callback approach, your callback method will be called when the execution of the target method is complete, allowing you to call **EndGetQuestionnaire** from there, as shown here.

C#

```
IAsyncResult asyncResult =
this.service.BeginGetQuestionnaire(GetQuestionnaireCompleted, null // object state,
not used in this example);

private void GetQuestionnaireCompleted(IAsyncResult result)
{
    try
    {
        questionnaire = this.service.EndGetQuestionnaire(ar);
    }
    catch (Exception ex)
    {
        // Do something to report the error.
    }
}
```

It is important to note that in the calls to the **End** method (in this case, **EndGetQuestionnaire**), any exceptions that occurred during the execution of the request will be raised. Your application must

handle these and may need to report them in a thread-safe way via the UI. If you do not handle these, the thread will end and you will not be able to process the results.

Because the response usually is not on the UI thread, if you plan to modify anything that will affect UI state, you will need to dispatch the response to the UI thread using either the thread **Dispatcher** or the **SynchronizationContext** objects. In WPF, you will commonly use the dispatcher.

In the following code example, the **Questionnaire** object is retrieved asynchronously, and then it is set as the data context for the **QuestionnaireView**. You can use the **CheckAccess** method of the dispatcher to see whether you are on the UI thread. If you are not, you will need to use the **BeginInvoke** method to have the request carried out on the UI thread.

C#

```
var dispatcher = System.Windows.Deployment.Current.Dispatcher;
if (dispatcher.CheckAccess())
{
    QuestionnaireView.DataContext = questionnaire;
}
else
{
    dispatcher.BeginInvoke(
        () => { Questionnaire.DataContext = questionnaire; });
}
```

The Model-View-ViewModel Reference Implementation (MVVM RI) shows an example of how to consume an **IAsyncResult**-based service interface similar to the preceding examples. It also wraps the service to provide a simpler callback mechanism for the consumer and handles the dispatch of the callback to the caller's thread. For example, the following code example shows retrieval of the questionnaire.

C#

```
this.questionnaireRepository.GetQuestionnaireAsync(
    (result) =>
{
    this.Questionnaire = result.Result;
});
```

The **result** object returned wraps the result retrieved in addition to errors that may have occurred. The following code example shows how the errors could be evaluated.

C#

```
this.questionnaireRepository.GetQuestionnaireAsync(
    (result) =>
{
    if (result.Error == null) {
        this.Questionnaire = result.Result;
        ...
    }
});
```

```

    else
    {
        // Handle error.
    }
})

```

User Interaction Patterns

Frequently, an application needs to notify the user of the occurrence of an event or ask for confirmation before proceeding with an operation. These interactions are often brief interactions designed to simply inform them of a change in the application or to obtain a simple response from them. Some of these interactions may appear modal to the user, such as when displaying a dialog box or a message box, or they may appear non-modal to the user, such as when displaying a toast notification or a pop-up window.

There are multiple ways to interact with the user in these cases, but implementing them in an MVVM-based application in a way that preserves a clean separation of concerns can be challenging. For example, in a non-MVVM application, you would often use the **MessageBox** class in the UI's code-behind file to simply prompt the user for a response. In an MVVM application, this would not be appropriate because it would break the separation of concerns between the view and the view model.

In terms of the MVVM pattern, the view model is responsible for initiating an interaction with the user and for consuming and processing any response, while the view is responsible for actually managing the interaction with the user using whatever user experience is appropriate. Preserving the separation of concerns between the presentation logic implemented in the view model, and the user experience implemented by the view, helps to improve testability and flexibility.

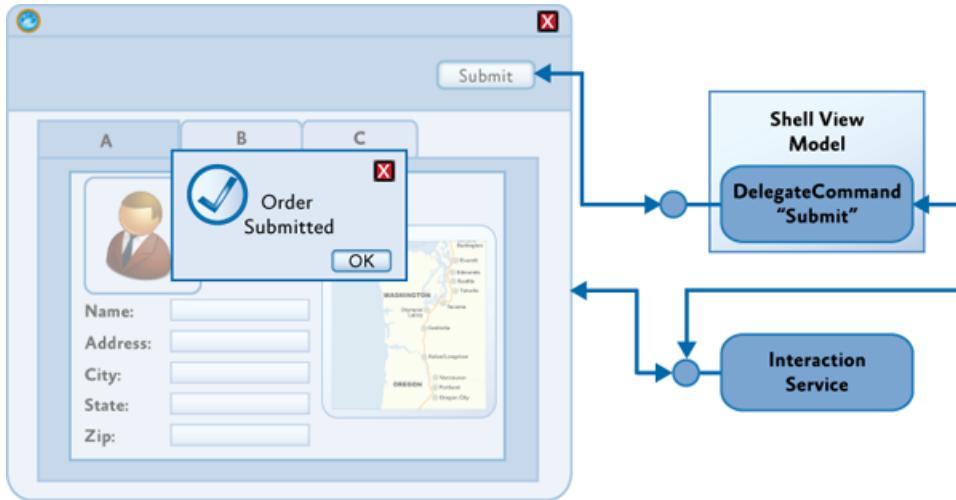
There are two common approaches to implementing these kinds of user interactions in the MVVM pattern. One approach is to implement a service that can be used by the view model to initiate interaction with the user, thereby preserving its independence on the view's implementation. Another approach uses events raised by the view model to express the intent to interact with the user, along with components in the view that are bound to these events and that manage the visual aspects of the interaction. Each of these approaches is described in the following sections.

Using an Interaction Service

In this approach, the view model relies on an interaction service component to initiate interaction with the user via a message box. This approach supports a clean separation of concerns and testability by encapsulating the visual implementation of the interaction in a separate service component. Typically, the view model has a dependency on an interaction service interface. It frequently acquires a reference to the interaction service's implementation via dependency injection or a service locator.

After the view model has a reference to the interaction service, it can programmatically request interaction with the user whenever necessary. The interaction service implements the visual aspects of the interaction, as shown in the following illustration. Using an interface reference in the view model allows for different implementations to be used, according to the implementation requirements of the

user interface. For example, implementations of the interaction service for WPF could be provided, allowing for greater re-use of the application's presentation logic.



Using an interaction service to interact with the user

Modal interactions, such as where the user is presented with a **MessageBox** or modal pop-up window to obtain a specific response before execution can proceed, can be implemented in a synchronous way, using a blocking method call, as shown in the following code example.

C#

```

var result =
    interactionService.ShowMessageBox(
        "Are you sure you want to cancel this operation?",
        "Confirm",
        MessageBoxButton.OK );
if (result == MessageBoxResult.Yes)
{
    CancelRequest();
}
  
```

However, one of the disadvantages of this approach is that it forces a synchronous programming model. An alternative asynchronous implementation allows for the view model to provide a callback to execute on completion of the interaction. The following code illustrates this approach.

C#

```

interactionService.ShowMessageBox(
    "Are you sure you want to cancel this operation?",
    "Confirm",
    MessageBoxButton.OK,
    result =>
    {
        if (result == MessageBoxResult.Yes)
        {
  
```

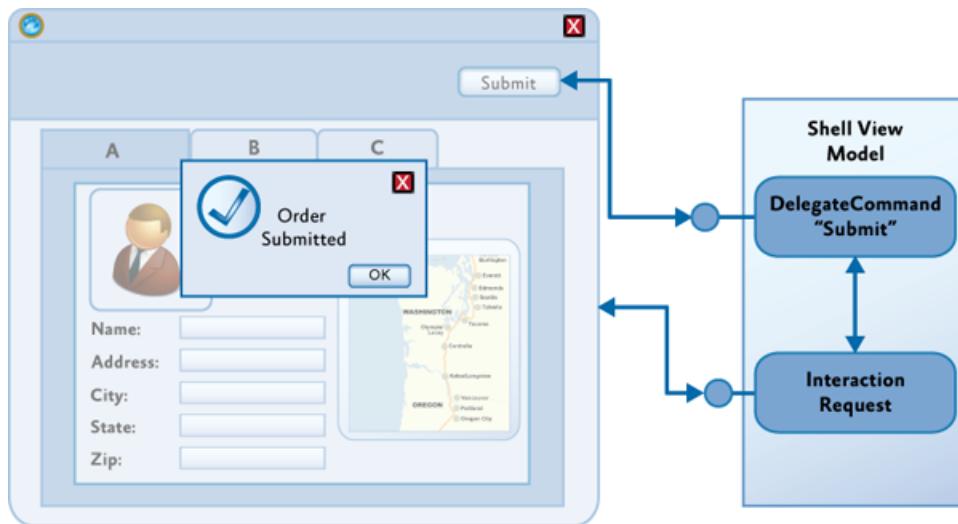
```

        CancelRequest();
    }
});
```

The asynchronous approach provides greater flexibility when implementing the interaction service by allowing modal and non-modal interactions to be implemented. For example, in WPF, the **MessageBox** class can be used to implement a truly modal interaction with the user.

Using Interaction Request Objects

Another approach to implementing simple user interactions in the MVVM pattern is to allow the view model to make interaction requests directly to the view itself via an interaction request object coupled with a behavior in the view. The interaction request object encapsulates the details of the interaction request, and its response, and communicates with the view via events. The view subscribes to these events to initiate the user experience portion of the interaction. The view will typically encapsulate the user experience of the interaction in a behavior that is data-bound to the interaction request object provided by the view model, as shown in the following illustration.



Using an interaction request object to interact with the user

This approach provides a simple, yet flexible, mechanism that preserves a clean separation between the view model and the view—it allows the view model to encapsulate the application's presentation logic, including any required user interactions, while allowing the view to fully encapsulate the visual aspects of the interaction. The view model's implementation, including its expected interactions with the user through view, can be easily tested, and the UI designer has a lot of flexibility in choosing how to implement the interaction within the view via the use of different behaviors that encapsulate the different user experiences for the interaction.

This approach is consistent with the MVVM pattern, enabling the view to reflect state changes it observes on the view model and using two-way data binding for communication of data between the two. The encapsulation of the non-visual elements of the interaction in an interaction request object,

and the use of a corresponding behavior to manage the visual elements of the interaction, are very similar to the way command objects and command behaviors are used.

This approach is the approached adopted by Prism. The Prism Library directly supports this pattern through the **IInteractionRequest** interface and the **InteractionRequest<T>** class. The **IInteractionRequest** interface defines an event to initiate the interaction. Behaviors in the view bind to this interface and subscribe to the event that it exposes. The **InteractionRequest<T>** class implements the **IInteractionRequest** interface and defines two **Raise** methods to allow the view model to initiate an interaction and to specify the context for the request, and optionally, a callback delegate.

Initiating Interaction Requests from the View Model

The **InteractionRequest<T>** class coordinates the view model's interaction with the view during an interaction request. The **Raise** method allows the view model to initiate the interaction and to specify a context object (of type **T**) and a callback method that is called after the interaction completes. The context object allows the view model to pass data and state to the view for it to be used during the interaction with the user. If a callback method was specified, the context object will be passed back to the view model; this allows any changes the user made during the interaction to be passed back to the view model.

C#

```
public interface IInteractionRequest
{
    event EventHandler<InteractionRequestedEventArgs> Raised;
}

public class InteractionRequest<T> : IInteractionRequest
    where T : INotification
{
    public event EventHandler<InteractionRequestedEventArgs> Raised;

    public void Raise(T context)
    {
        this.Raised(context, c => { });
    }

    public void Raise(T context, Action<T> callback)
    {
        var handler = this.Raised;
        if (handler != null)
        {
            handler(
                this,
                new InteractionRequestedEventArgs(
                    context,
                    () => { if (callback != null) callback(context); } ));
        }
    }
}
```

```

    }
}
```

Prism provides pre-defined context classes that support common interaction request scenarios. The **INotification** interface is used for all context objects. It is used when the interaction request is used to notify the user of an important event in the application. It provides two properties—**Title** and **Content**—which will be displayed to the user. Typically, notifications are one-way, so it is not expected that the user will change these values during the interaction. The **Notification** class is the default implementation of this interface.

The **IConfirmation** interface extends the **INotification** interface and adds a third property—**Confirmed**—which is used to signify that the user has confirmed or denied the operation. The **Confirmation** class, the provided **IConfirmation** implementation, is used to implement **MessageBox** style interactions where the user wants to obtain a yes/no response from the user. You can define a custom context class that implements the **INotification** interface to encapsulate whatever data and state you need to support the interaction.

To use the **InteractionRequest<T>** class, the view model class will create an instance of the **InteractionRequest<T>** class and define a read-only property to allow the view to data-bind against it. When the view model wants to initiate the request, it will call the **Raise** method, passing in the context object and, optionally, the callback delegate.

C#

```

public InteractionRequestViewModel()
{
    this.ConfirmationRequest = new InteractionRequest<IConfirmation>();
    ...
    // Commands for each of the buttons. Each of these raise a different interaction
    // request.
    this.RaiseConfirmationCommand = new DelegateCommand(this.RaiseConfirmation);
    ...
}

public InteractionRequest<IConfirmation> ConfirmationRequest { get; private set; }

private void RaiseConfirmation()
{
    this.ConfirmationRequest.Raise(
        new Confirmation { Content = "Confirmation Message", Title = "Confirmation" },
        c => { InteractionResultMessage = c.Confirmed ? "The user accepted." : "The
user cancelled." });
}
```

The [Interactivity QuickStart](#) illustrates how the **IInteractionRequest** interface and the **InteractionRequest<T>** class are used to implement user interactions between the view and view model (see `InteractionRequestViewModel.cs`).

Using Behaviors to Implement the Interaction User Experience

Because the interaction request object represents a logical interaction, the exact user experience for the interaction is defined in the view. Behaviors are often used to encapsulate the user experience for an interaction; this allows the UI designer to choose an appropriate behavior and to bind it to the interaction request object on the view model.

The view must be set up to detect an interaction request event, and then to present the appropriate visual display for the request. Triggers are used to initiate actions whenever a specific event is raised.

The standard **EventTrigger** provided by Blend can be used to monitor an interaction request event by binding to the interaction request objects exposed by the view model. However, the Prism Library defines a custom **EventTrigger**, named **InteractionRequestTrigger**, which automatically connects to the appropriate **Raised** event of the **IInteractionRequest** interface. This reduces the amount of Extensible Application Markup Language (XAML) needed and reduces the chance of inadvertently entering an incorrect event name.

After the event is raised, the **InteractionRequestTrigger** will invoke the specified action. For WPF, the Prism Library provides the **PopupWindowAction** class, which displays a pop-up window to the user. When the window is displayed, its data context is set to the context parameter of the interaction request. Using the **WindowContent** property of the **PopupWindowAction** class, you can specify the view that will be shown in the popup window. The title of the pop-up window is bound to the **Title** property of the context object.

Note: By default, the specific type of pop-up window displayed by the **PopupWindowAction** class depends on the type of the context object. For a **Notification** context object, a **DefaultNotificationWindow** is displayed, while for a **Confirmation** context object, a **DefaultConfirmationWindow** is displayed. The **DefaultNotificationWindow** displays a simple popup window to display the notification, while the **DefaultConfirmationWindow** also contains **Accept** and **Cancel** buttons to capture the user's response. You can override this behavior by specifying a custom pop-up window using the **WindowContent** property of the **PopupWindowAction** class.

The following example shows how the **InteractionRequestTrigger** and the **PopupWindowAction** are used to display a confirmation pop-up window to the user within the Interactivity QuickStart.

XAML

```
<i:Interaction.Triggers>
    <prism:InteractionRequestTrigger SourceObject="{Binding ConfirmationRequest,
Mode=OneWay}">
        <prism:PopupWindowAction IsModal="True" CenterOverAssociatedObject="True"/>
    </prism:InteractionRequestTrigger>
</i:Interaction.Triggers>
```

Note: The **PopupWindowAction** has three important properties, **IsModal**, which sets the popup to modal when set to true; **CenterOverAssociatedObject**, which displays the popup centered to the parent window when set to true. Finally, the **WindowContent** property, which is not specified, therefore the **DefaultConfirmationWindow** will be shown.

The **PopupWindowAction** sets the **Notification** object as the data context of the **DefaultNotificationWindow**, which displays the **Content** property of the **Notification** object. After the user closes the pop-up window, the context object is passed back to the view model, along with any updated values, via the callback method. In the confirmation example in the Interactivity QuickStart, the **DefaultConfirmationWindow** is responsible for setting the **Confirmed** property on the supplied **Confirmation** object to **true** when the **OK** button is clicked.

Different triggers and actions can be defined to support other interaction mechanisms. The implementation of the Prism **InteractionRequestTrigger** and **PopupWindowAction** classes can be used as a basis for the development of your own triggers and actions.

Advanced Construction and Wire-Up

To successfully implement the MVVM pattern, you will need to fully understand the responsibilities of the view, model, and view model classes so that you can implement your application's code in the correct classes. Implementing the correct patterns to allow these classes to interact (through data binding, commands, interaction requests, and so on) is also an important requirement. The final step is to consider how the view, view model, and model classes are instantiated and associated with each other at run time.

Choosing an appropriate strategy to manage this step is especially important if you are using a dependency injection container in your application. The Managed Extensibility Framework (MEF) and the Unity Application Block (Unity) both provide the ability to specify dependencies between the view, view model, and model classes and to have them fulfilled by the container at run time.

Typically, you define the view model as a dependency of the view, so that when the view is constructed (using the container) it automatically instantiates the required view model. In turn, any components or services that the view model depends on will also be instantiated by the container. After the view model is successfully instantiated, the view then sets it as its data context.

Creating the View and View Model Using MEF

Using MEF, you can specify the view's dependency on a view model using the **import** attribute, and you can specify the concrete view model type to be instantiated via an **export** attribute. You can either import the view model into the view via a property or as a constructor argument.

For example, the **Shell** view in the StockTrader Reference Implementation declares a write-only property for the view model, together with an **import** attribute. When the view is instantiated, MEF creates an instance of the appropriate exported view model and sets the property value. The property setter assigns the view model as the view's data context, as shown here.

C#

```
[Import]
ShellViewModel ViewModel
{
    set { this.DataContext = value; }
}
```

The view model is defined and exported, as shown here.

C#

```
[Export]
public class ShellViewModel : BindableBase
{
    ...
}
```

An alternative approach is to define an importing constructor on the view, as shown here.

C#

```
public Shell()
{
    InitializeComponent();
}

[ImportingConstructor]
public Shell(ShellViewModel viewModel) : this()
{
    this.DataContext = viewModel;
}
```

The view model will then be instantiated by MEF and passed as an argument to the view's constructor.

Note: You can use property injection or constructor injection in both MEF and Unity; however, you may find property injection to be simpler because you do not have to maintain two constructors. Design-time tools, such as Visual Studio and Expression Blend, require that controls have a default parameter-less constructor in order to display them in the designer. Any additional constructors that you define should ensure that the default constructor is called so that view can be properly initialized via the **InitializeComponent** method.

[Creating the View and View Model Using Unity](#)

Using Unity as your dependency injection container is similar to using MEF, and both property-based and constructor-based injection are supported. The principal difference is that the types are typically not implicitly discovered at run time; instead, they have to be registered with the container.

Typically, you define an interface on the view model so the view model's specific concrete type can be decoupled from the view. For example, the view can define its dependency on the view model via a constructor argument, as shown here.

C#

```
public Shell()
{
    InitializeComponent();
}

public Shell(ShellViewModel viewModel)
: this()
{
    this.DataContext = viewModel;
}
```

Note: The default parameter-less constructor is necessary to allow the view to work in design-time tools, such as Visual Studio and Blend for Visual Studio 2013.

Alternatively, you can define a write-only view model property on the view, as shown here. Unity will instantiate the required view model and call the property setter after the view is instantiated.

C#

```
public Shell()
{
    InitializeComponent();
}

[Dependency]
public ShellViewModel ViewModel
{
    set { this.DataContext = value; }
}
```

The view model type is registered with the Unity container, as shown here.

C#

```
IUnityContainer container;
container.RegisterType<ShellViewModel>();
```

The view can then be instantiated through the container, as shown here.

C#

```
IUnityContainer container;
var view = container.Resolve<Shell>();
```

Creating the View and View Model Using an External Class

Often, you will find it useful to define a controller or service class to coordinate the instantiation of the view and view model classes. This approach can be used with a dependency injection container, such as MEF or Unity, or when the view explicitly creates its required view model.

This approach is particularly useful when implementing navigation in your application. In this case, the controller is associated with a placeholder control or region in the UI, and it coordinates the construction and placement of views into that placeholder or region.

For example, a service class can be used to build views using a container and show them in the main page. In this example, views are specified by view names. Navigation is initiated via a call to the **ShowView** method on the UI service, as shown in this simple example.

C#

```
private void NavigateToQuestionnaireList()
{
    // Ask the UI service to go to the "questionnaire list" view.
    this.uiService.ShowView(ViewNames.QuestionnaireTemplatesList);
}
```

The UI service is associated with a placeholder control in the UI of the application; it encapsulates the creation of the required view and coordinates its appearance in the UI. The **ShowView** of the **UIService** creates an instance of the view via the container (so that its view model and other dependencies can be fulfilled) and then displays it in the proper location, as shown here.

C#

```
public void ShowView(string viewName)
{
    var view = this.ViewFactory.GetView(viewName);
    this.MainWindow.CurrentView = view;
}
```

Note: Prism provides extensive support for navigation within regions. Region navigation uses a mechanism very similar to the preceding approach, except that the region manager is responsible for coordinating the instantiation and placement of the view in the specific region. For more information, see the section, [View-Based Navigation](#) in [Navigation](#).

Testing MVVM Applications

Testing models and view models from MVVM applications is the same as testing any other classes, and the same tools and techniques—such as unit testing and mocking frameworks—can be used. However, there are some testing patterns that are typical to model and view model classes and can benefit from standard testing techniques and test helper classes.

Testing INotifyPropertyChanged Implementations

Implementing the **INotifyPropertyChanged** interface allows views to react to changes originated in models and view models. These changes are not limited to domain data shown in controls; they are also used to control the view, such as view model states that cause animations to be started or controls to be disabled.

Simple Cases

Properties that can be updated directly by the test code can be tested by attaching an event handler to the **PropertyChanged** event and checking whether the event is raised after setting a new value for the property. Helper classes, such as the **PropertyChangeTracker** class, can be used to attach a handler and collect the results; this avoids repetitive tasks when writing tests. The following code example shows a test using this type of helper class.

C#

```
var changeTracker = new PropertyChangeTracker(viewModel);

viewModel.CurrentState = "newState";

CollectionAssert.Contains(changeTracker.ChangedProperties, "CurrentState");
```

Properties that are the result of a code-generation process that guarantees the implementation of the **INotifyPropertyChanged** interface, such as those in code generated by a model designer, typically do not need to be tested.

Computed and Non-Settable Properties

When properties cannot be set by test code—such as properties with non-public setters or read-only, calculated properties—the test code needs to stimulate the object under test cause the change in the property and its corresponding notification. However, the structure of the test is the same as that of the simpler cases, as shown in the following code example, where a change in a model objects causes a property in a view model to change.

C#

```
var changeTracker = new PropertyChangeTracker(viewModel);

var question = viewModel.Questions.First() as OpenQuestionViewModel;
question.Question.Response = "some text";

CollectionAssert.Contains(changeTracker.ChangedProperties, "UnansweredQuestions");
```

Whole Object Notifications

When you implement the **INotifyPropertyChanged** interface, it is allowed for an object to raise the **PropertyChanged** event with a null or empty string as the changed property name to indicate that all properties in the object may have changed. These cases can be tested just like the cases that notify individual property names.

Testing **INotifyDataErrorInfo** Implementations

There are several mechanisms available to enable bindings to perform input validation, such as throwing exceptions when properties are set, implementing the **IDataErrorInfo** interface, and implementing the **INotifyDataErrorInfo** interface. Implementing the **INotifyDataErrorInfo** interface allows for greater

sophistication because it supports indicating multiple errors per property and performing asynchronous and cross-property validation; as such, it also requires the most testing.

There are two aspects to testing **INotifyDataErrorInfo** implementations: testing that the validation rules are correctly implemented and testing that the requirements for implementations of the interface, such as raising the **ErrorsChanged** event when the result for the **GetErrors** method would be different, are met.

Testing Validation Rules

Validation logic is usually simple to test, because it is typically a self-contained process where the output depends on the input. For each property with validation rules associated, there should be tests on the results of invoking the **GetErrors** method with the validated property name for valid values, invalid values, boundary values, and so on. If the validation logic is shared, like when expressing validation rules declaratively using the data annotation's validation attribute, the more exhaustive tests can be concentrated on the shared validation logic. On the other hand, custom validation rules must be thoroughly tested.

C#

```
// Invalid case
var notifyErrorInfo = (INotifyDataErrorInfo)question;

question.Response = -15;

Assert.IsTrue(notifyErrorInfo.GetErrors("Response").Cast<ValidationResult>().Any());

// Valid case
var notifyErrorInfo = (INotifyDataErrorInfo)question;

question.Response = 15;
Assert.IsFalse(notifyErrorInfo.GetErrors("Response").Cast<ValidationResult>().Any());
```

Cross-property validation rules follow the same pattern, typically requiring more tests to accommodate the combination of values for the different properties.

Testing the Requirements for INotifyDataErrorInfo Implementations

Besides producing the right values for the **GetErrors** method, implementations of the **INotifyDataErrorInfo** interface must ensure the **ErrorsChanged** event is raised appropriately, such as when the result for **GetErrors** would be different. Additionally, the **HasErrors** property must reflect the overall error state of the object implementing the interface.

There is no mandatory approach for implementing the **INotifyDataErrorInfo** interface. However, implementations that rely on objects that accumulate validation errors and perform the necessary notifications are typically preferred because they are simpler to test. This is because it is not necessary to verify that the requirements for all the members of the **INotifyDataErrorInfo** interface are met for

each validation rule on each validated property (as long, of course, as the error management object is properly tested).

Testing the interface requirements should involve at least the following verifications:

- The **HasErrors** property reflects the overall error state of the object. Setting a valid value for a previously invalid property does not result in a change for this property if other properties still have invalid values.
- The **ErrorsChanged** event is raised when the error state for a property changes, as reflected by a change in the result for the **GetErrors** method. The error state change could be going from a valid state (that is, no errors) to an invalid state and vice versa, or it can go from an invalid state to a different invalid state. The updated result for **GetErrors** is available for handlers of the **ErrorsChanged** event.

When testing implementations for the **INotifyPropertyChanged** interface, helper classes, such as the **NotifyDataErrorInfoTestHelper** class in the MVVM sample projects, usually make writing tests for implementations of the **INotifyDataErrorInfo** interface easier by handling repetitive housekeeping operations and standard checks. They are particularly useful when the interface is implemented without relying on some kind of reusable errors manager. The following code example shows this type of helper class.

C#

```
var helper =
    new NotifyDataErrorInfoTestHelper<NumericQuestion, int?>(
        question,
        q => q.Response);

helper.ValidatePropertyChange(
    6,
    NotifyDataErrorInfoBehavior.Nothing);
helper.ValidatePropertyChange(
    20,
    NotifyDataErrorInfoBehavior.FiresErrorsChanged
    | NotifyDataErrorInfoBehavior.HasErrors
    | NotifyDataErrorInfoBehavior.HasErrorsForProperty);
helper.ValidatePropertyChange(
    null,
    NotifyDataErrorInfoBehavior.FiresErrorsChanged
    | NotifyDataErrorInfoBehavior.HasErrors
    | NotifyDataErrorInfoBehavior.HasErrorsForProperty);
helper.ValidatePropertyChange(
    2,
    NotifyDataErrorInfoBehavior.FiresErrorsChanged);
```

Testing Asynchronous Service Calls

When implementing the MVVM pattern, view models usually invoke operations on services, often asynchronously. Tests for code that invokes these operations typically use mocks or stubs as replacements for the actual services.

The standard patterns used to implement asynchronous operations provide different guarantees regarding the thread in which notifications about the status of an operation occur. Although the [Event-based Asynchronous design pattern](#) guarantees that handlers for the events are invoked on a thread that is appropriate for the application, the [IAsyncResult design pattern](#) does not provide any such guarantees forcing the view model code that originates the call to ensure any changes that would affect the view are posted to the UI thread.

Dealing with threading concerns requires more complicated, and, therefore, usually harder to test, code. It also usually requires the tests themselves to be asynchronous. When notifications are guaranteed to occur in the UI thread, either because the standard event-based asynchronous pattern is used or because view models rely on a service access layer to marshal notifications to the appropriate thread, tests can be simplified and can essentially play the role of a "dispatcher for the UI thread."

The way services are mocked depends on the asynchronous event pattern used to implement their operations. If a method-based based pattern is used, mocks for the service interface created using a standard mocking framework are usually enough, but if the event-based pattern is used, mocks based on a custom class that implements the methods to add and remove handlers for the service events are usually preferred.

The following code example shows a test for the appropriate behavior on the successful completion of an asynchronous operation notified in the UI thread using mocks for services. In this example, the test code captures the callback supplied by the view model when it makes the asynchronous service call. The test then simulates the completion of that call later in the test by invoking the callback. This approach allows testing of a component that uses an asynchronous service without the complexity of making your tests asynchronous.

C#

```
questionnaireRepositoryMock
    .Setup(
        r =>
            r.SubmitQuestionnaireAsync(
                It.IsAny<Questionnaire>(),
                It.IsAny<Action<IOperationResult>>())
    .Callback<Questionnaire, Action<IOperationResult>>(
        (q, a) => callback = a);

uiServiceMock
    .Setup(svc => svc.ShowView(ViewNames.QuestionnaireTemplatesList))

    .Callback<string>(viewName => requestedViewName = viewName);
submitResultMock
```

```

    .Setup(sr => sr.Error)
    .Returns<Exception>(null);
CompleteQuestionnaire(viewModel);
viewModel.Submit();
// Simulate callback posted to the UI thread.
callback(submitResultMock.Object);
// Check expected behavior - request to navigate to the list view.
Assert.AreEqual(ViewNames.QuestionnaireTemplatesList, requestedViewName);

```

Note: Using this testing approach only exercises the functional capabilities of the objects under test; it does not test that the code is thread safe.

More Information

For more information about the logical tree, see [Trees in WPF](#) on MSDN.

For more information about attached properties, see [Attached Properties Overview](#) on MSDN.

For more information about MEF, see [Managed Extensibility Framework Overview](#) on MSDN.

For more information about Unity, see [Unity Application Block](#) on MSDN.

For more information about **DelegateCommand**, see [Implementing the MVVM Pattern](#).

For more information about using Microsoft Expression Blend behaviors, see [Working with built-in behaviors](#) on MSDN.

For more information about creating custom behaviors with Microsoft Expression Blend, see [Creating Custom Behaviors](#) on MSDN.

For more information about creating custom triggers and actions with Microsoft Expression Blend, see [Creating Custom Triggers and Actions](#) on MSDN.

For more information about using the dispatcher in WPF , see [Threading Model](#) and [The Dispatcher Class](#) on MSDN.

For more information about region navigation, see the section, [View-Based Navigation](#) in [Navigation](#).

For more information about the Event-based Asynchronous pattern, see [Event-based Asynchronous Pattern Overview](#) on MSDN.

For more information about the IAsyncResult design pattern, see [Asynchronous Programming Overview](#) on MSDN.

7: Composing the User Interface

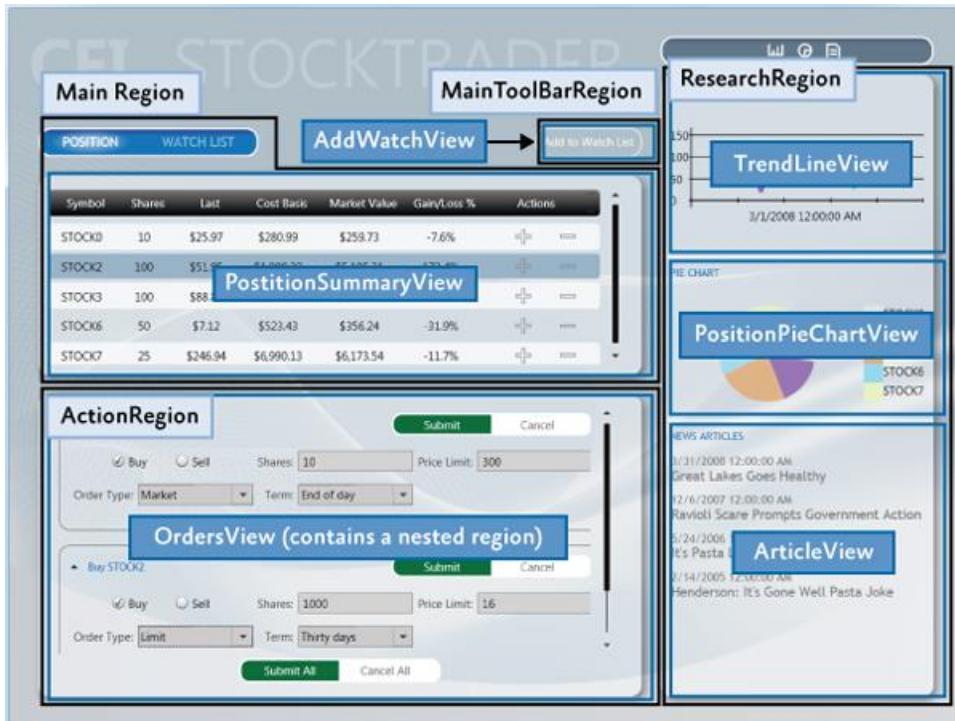
A composite application user interface (UI) is composed from loosely coupled visual components known as *views* that are typically contained in the application modules, but they do not need to be. If you divide your application into modules, you need some way to loosely compose the UI, but you might choose to use this approach even if the views are not in modules. To the user, the application presents a seamless user experience and delivers a fully integrated application.

To compose your UI, you need an architecture that allows you to create a layout composed of loosely coupled visual elements generated at run time. Additionally, the architecture should provide strategies for these visual elements to communicate in a loosely coupled fashion.

An application UI can be built by using one of the following paradigms:

- All required controls for a form are contained in a single Extensible Application Markup Language (**XAML**) file, composing the form at design time.
- Logical areas of the form are separated into distinct parts, typically user controls. The parts are referenced by the form, and the form is composed at design time.
- Logical areas of the form are separated into distinct parts, typically user controls. The parts are unknown to the form and are dynamically added to the form at run time. Applications that use this methodology are known as composite applications using UI composition patterns.

The Stock Trader Reference Implementation (Stock Trader RI) is composed by loading multiple views that come from different modules into regions exposed by the shell, as shown in the following illustration.



Stock Trader RI regions and views

UI Layout Concepts

The root object in a composite application is known as the *shell*. The shell acts as a master page for the application. The shell contains one or more regions. Regions are place holders for content that will be loaded at run time. Regions are attached to UI elements such as a **ContentControl**, **ItemsControl**, **TabControl** or a custom control and manage the UI element's content. Region content can be loaded automatically or on-demand, depending on the application requirements.

Typically, a region's content is a view. A view encapsulates a portion of your UI that you would like to keep as decoupled as possible from other parts of the application. You can define a view as a user control, data template, or even a custom control.

A region manages the display and layout of views. Regions can be accessed in a decoupled way by their name and support dynamically adding or removing views. A region is attached to a hosting control. Think of regions as containers into which views are dynamically loaded.

The following sections introduce the high-level core concepts for composite application development.

Shell

The shell is the application root object that contains the primary UI content. In a Windows Presentation Foundation (WPF) application, the shell is the **Window** object.

The shell plays the role of a master page providing the layout structure for the application. The shell contains one or more named regions where modules can specify the views that will appear. It can also define certain top-level UI elements, such as the background, main menu, and toolbar.

The shell defines the overall appearance of the application. It might define styles and borders that are present and visible in the shell layout itself, and it might also define styles, templates, and themes that will be applied to the views that are plugged into the shell.

Typically, the shell is a part of the WPF application project. The assembly that contains the shell might or might not reference the assemblies that contain the views to be loaded in the shell's regions.

Views

Views are the main unit of UI construction within a composite application. You can define a view as a user control, page, data template, or custom control. A view encapsulates a portion of your UI that you would like to keep as decoupled as possible from other parts of the application. You can choose what goes in a view based on encapsulation or a piece of functionality, or you can choose to define something as a view because you will have multiple instances of that view in your application.

Because of the content model of WPF, there is nothing specific to the Prism Library required to define a view. The easiest way to define a view is to define a user control. To add a view to the UI, you simply need a way to construct it and add it to a container. WPF provides mechanisms to do this. The Prism Library adds the ability to define a region into which a view can be dynamically added at run time.

Composite Views

A view that supports specific functionality can become complicated. In that case, you might want to divide the view into several child views and have the parent view handle constructing itself by using the child views as parts. The application might do this statically at design time, or it might support having modules add child views through a contained region at run time. When you have a view that is not fully defined in a single view class, you can refer to that as a composite view. In many situations, a composite view is responsible for constructing the child views and for coordinating the interactions between them. You can design child views that are more loosely coupled from their sibling views and their parent composite view by using the Prism Library commands and the event aggregator.

Views and Design Patterns

Although the Prism Library does not require that you use them, you should consider using one of several UI design patterns when implementing a view. The Stock Trader RI and QuickStarts demonstrate the Model-View-ViewModel (MVVM) pattern as a way to implement a clean separation between the view layout and the view logic.

The MVVM UI design pattern is recommended because it is a natural fit for the Microsoft XAML platforms. The dependency property system and rich data binding stack of these platforms enable the view and view model to communicate in a loosely coupled manner.

Separating the logic from the view is important for testability and maintainability, and it improves the developer-designer workflow.

If you create a view with a user control or custom control and put all the logic in the code-behind file, your view can be difficult to test because you have to create an instance of the view to unit test the logic. This is a problem particularly if the view derives from, or depends on, running WPF components as

part of its execution context. To make sure that you can unit test the view logic in isolation without these dependencies, you need to be able to create a mockup of the view to remove the dependencies on the execution context, which requires separate classes for the view and the logic.

If you define a view as a data template, there is no code associated with the view itself. Therefore, you have to put the associated logic somewhere else. The same clean separation of logic from layout that is required for testability also helps make the view easier to maintain.

Note: Unit testing and UI automation testing are two different types of testing with different coverage.

Unit testing best practices recommend that the object be tested in isolation. To achieve object isolation, you need a mockup or stub for each external dependency. Then granular unit tests are run against the object.

UI automation testing runs the application, applies gestures to the UI, and then tests for the expected results. This type of test verifies that UI elements are correctly connected to the application logic.

Separating the logic from the view provides a clean separation of concerns. In addition to testability considerations, this separation enables designers to work on the UI independently of the developer. For more information about MVVM, see [Implementing the MVVM Pattern](#).

Commands, UI Triggers, Actions, and Behaviors

When a view is implemented with its logic in the code-behind file, you add event handlers to service UI interactions. However, when you use MVVM, the view model cannot directly handle events raised by the UI. To route UI gesture events to the view model, you can use commands or UI triggers, actions, and behaviors.

Commands

Commands separate the semantics and the object that invokes a command from the logic that executes the command. Built into commands is the ability to indicate whether an action is available. Commands in the UI are data bound to **ICommand** properties on the view model. For more information about commands, see [Commands](#) in [Implementing the MVVM Pattern](#).

UI Triggers, Actions, and Behaviors

Triggers, actions, and behaviors are part of the **Microsoft.Expression.Interactivity** namespace and are shipped with Blend for Visual Studio 2013. They are also part of the Blend SDK. Triggers, actions, and behaviors provide a comprehensive API for handling UI events or commands, and then routing them to the **ICommand** properties methods exposed by the **DataContext**. For more information about UI triggers, actions, and behaviors, see sections [Invoking Command Objects from the View](#) in [Implementing the MVVM Pattern](#) and [Interaction Triggers and Events to Commands](#) in [Advanced MVVM Scenarios](#).

User Interactions

User interactions are interactions that the application presents to the user. These interactions are typically popup windows presented to the user. In MVVM scenarios these user interactions can be generated either from the view or from the view model. Prism provides **InteractionRequests** and **InteractionRequestTriggers** for cases when the view model needs to request a user interaction, and the **InvokeCommandAction** action for when the view needs to invoke a command when a specified event is fired.

For more information about user interactions, examples, and how to use them, see the [Interactivity QuickStart](#).

Data Binding

Data binding is one of the most important framework features of the XAML platforms. To successfully develop applications on the XAML platforms, you need a solid understanding of data binding.

Data binding takes full advantage of the intrinsic change notification provided by the dependency property system. When combined with the Common Language Runtime (CLR) class implementation of the **INotifyPropertyChanged** interface, change notification enables codeless interaction between the target and source objects participating in the data binding.

Data binding enables dissimilar target and source types to data bind by using a value converter to convert one type to the other type. Data binding has multiple validation hooks within its pipeline that you can use to validate user input.

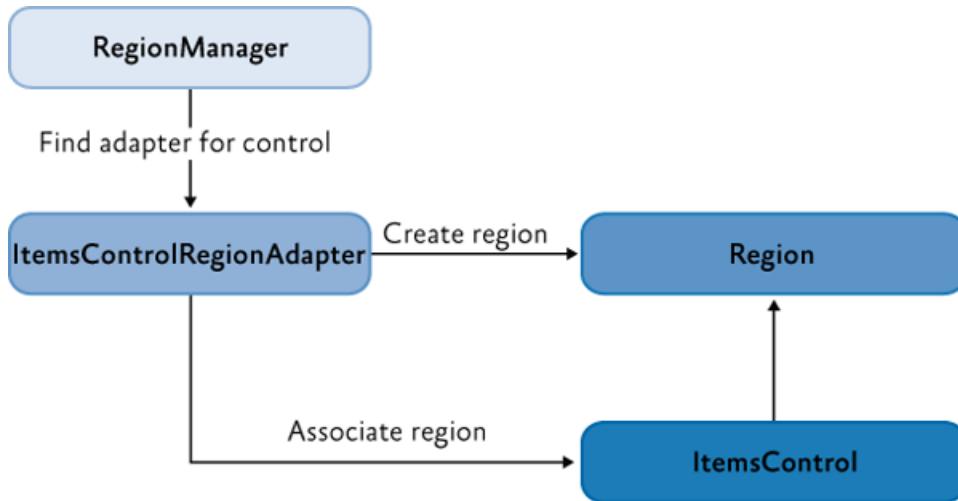
You are strongly encouraged to read the [Dependency Properties Overview](#) and [Data Binding Overview](#) topics on MSDN. A full understand of these two topics is critical to successfully developing applications on the Microsoft XAML platforms. For more information about data binding, see [Data Binding](#) in [Implementing the MVVM Pattern](#).

Regions

Regions are enabled in the Prism Library through a region manager, regions, and region adapters. The next sections describe how they work together.

Region Manager

The **RegionManager** class is responsible for creating and maintaining a collection of regions for the host controls. The **RegionManager** uses a control-specific adapter that associates a new region with the host control. The following illustration shows the relationship between the region, control, and adapter set up by the **RegionManager**.



Region, control, and adapter relationship

The **RegionManager** can create regions in code or in XAML. The **RegionManager.RegionName** attached property is used to create a region in XAML by applying the attached property to the host control.

Applications can contain one or more instances of a **RegionManager**. You can specify the **RegionManager** instance into which you want to register the region. This is useful if you want to move the control around in the visual tree and do not want the region to be cleared when the attached property value is removed.

The **RegionManager** provides a **RegionContext** attached property that permits its regions to share data.

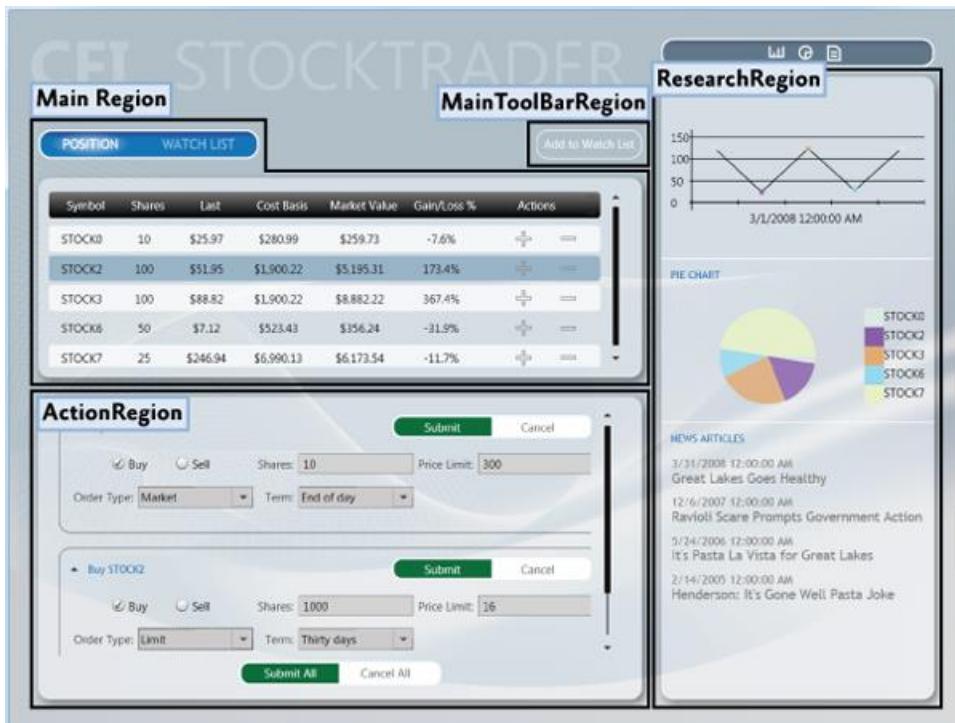
Region Implementation

A region is a class that implements the **IRegion** interface. The term *region* represents a container that can hold dynamic data that is presented in a UI. A region allows the Prism Library to place dynamic content contained in modules in predefined placeholders in a UI container.

Regions can hold any type of UI content. A module can contain UI content presented as a user control, a data type that is associated with a data template, a custom control, or any combination of these. This lets you define the appearance for the UI areas and then have modules place content in these predetermined areas.

A region can contain zero or more items. Depending on the type of host control the region is managing, one or more of the items could be visible. For example, a **ContentControl** can display only a single object. However, the region in which it is located can contain many items, and an **ItemsControl** can display multiple items. This allows each item in the region to be visible in the UI.

In the following illustration, the Stock Trader RI shell contains four regions: **MainRegion**, **MainToolbarRegion**, **ResearchRegion**, and **ActionRegion**. These regions are populated by the various modules in the application—the content can be changed at any time.



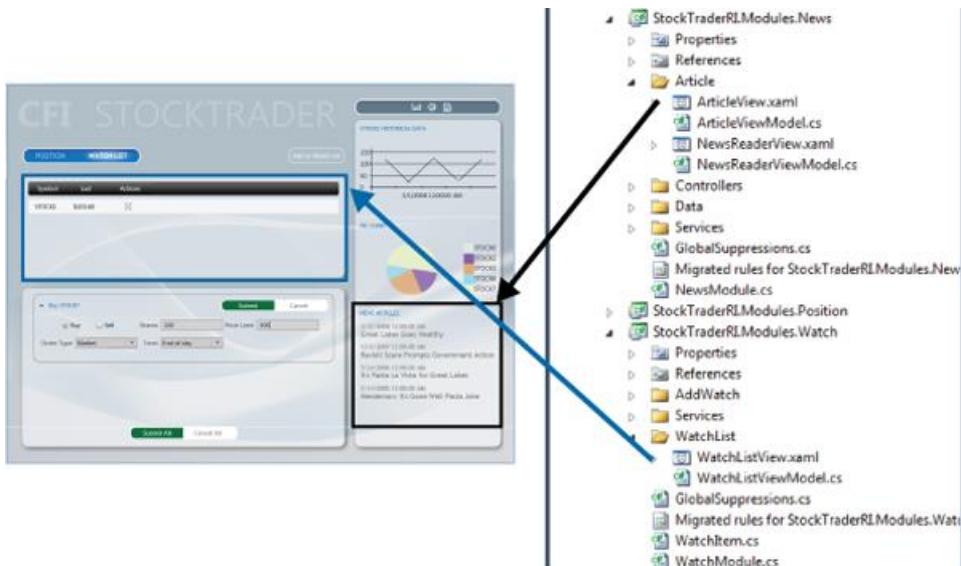
Stock Trader RI regions

Module User Control to Region Mapping

To demonstrate how modules and content are associated with regions, see the following illustration. It shows the association of **WatchModule** and the **NewsModule** with the corresponding regions in the shell.

The **MainRegion** contains the **WatchListView** user control, which is contained in the **WatchModule**. The **ResearchRegion** also contains the **ArticleView** user control, which is contained in the **NewsModule**.

In applications created with the Prism Library, mappings like this will be a part of the design process because designers and developers use them to determine what content is proposed to be in a specific region. This allows designers to determine the overall space needed and any additional items that must be added to ensure that the content will be viewable in the allowable space.



Module user control to region mapping

Default Region Functionality

While you do not need to fully understand region implementations to use them, it might be useful to understand how controls and regions are associated and the default region functionality: for example, how a region locates and instantiates views, how views can be notified when they are the active view, or how view lifetime can be tied to activation.

The following sections describe the region adapter and region behaviors.

Region Adapter

To expose a UI control as a region, it must have a region adapter. Region adapters are responsible for creating a region and associating it with the control. This allows you to use the **IRegion** interface to manage the UI control contents in a consistent way. Each region adapter adapts a specific type of UI control. The Prism Library provides the following three region adapters:

- **ContentControlRegionAdapter.** This adapter adapts controls of type **System.Windows.Controls.ContentControl** and derived classes.
- **SelectorRegionAdapter.** This adapter adapts controls derived from the class **System.Windows.Controls.Primitives.Selector**, such as the **System.Windows.Controls.TabControl** control.
- **ItemsControlRegionAdapter.** This adapter adapts controls of type **System.Windows.Controls.ItemsControl** and derived classes.

Region Behaviors

The Prism Library introduces the concept of region behaviors. These are pluggable components that give a region most of its functionality. Region behaviors were introduced to support view discovery and

region context (described later in this topic), and to create an API that is consistent across both WPF and Silverlight. Additionally, behaviors provide an effective way to extend a region's implementation.

A region behavior is a class that is attached to a region to give the region additional functionality. This behavior is attached to the region and remains active for the lifetime of the region. For example, when an **AutoPopulateRegionBehavior** is attached to a region, it automatically instantiates and adds any **ViewTypes** that are registered against regions with that name. For the lifetime of the region, it keeps monitoring the **RegionViewRegistry** for new registrations. It is easy to add custom region behaviors or replace existing behaviors, either on a system-wide or a per-region basis.

The next sections describe the default behaviors that are automatically added to all regions. One behavior, the **SelectorItemsSourceSyncBehavior**, is only attached to controls that derive from the **Selector**.

Registration Behavior

The **RegionManagerRegistrationBehavior** is responsible for making sure that the region is registered to the correct **RegionManager**. When a view or control is added to the visual tree as a child of another control or region, any region defined in the control should be registered in the **RegionManager** of the parent control. When the child control is removed, the registered region is unregistered.

Auto-Population Behavior

There are two classes responsible for implementing view discovery. One of them is the **AutoPopulateRegionBehavior**. When it is attached to a region, it retrieves all view types that are registered under the name of the region. It then creates instances of those views and adds them to the region. After the region is created, the **AutoPopulateRegionBehavior** monitors the **RegionViewRegistry** for any newly registered view types for that region name.

If you want to have more control over the view discovery process, consider creating your own implementation of the **IRegionViewRegistry** and the **AutoPopulateRegionBehavior**.

Region Context Behaviors

The region context functionality is contained within two behaviors: the **SyncRegionContextWithHostBehavior** and the **BindRegionContextToDependencyObjectBehavior**. These behaviors are responsible for monitoring changes to the context that were made on the region, and then synchronizing the context with a context dependency property attached to the view.

Activation Behavior

The **RegionActiveAwareBehavior** is responsible for notifying a view if it is active or inactive. The view must implement **IActiveAware** to receive these change notifications. This active aware notification is one-directional (it travels from the behavior to the view). The view cannot affect its active state by changing the active property on the **IActiveAware** interface.

Region Lifetime Behavior

The **RegionMemberLifetimeBehavior** is responsible for determining if an item should be removed from the region when it is deactivated. The **RegionMemberLifetimeBehavior** monitors the region's

ActiveViews collection to discover items that transition into a deactivated state. The behavior checks the removed items for **IRegionMemberLifetime** or the **RegionMemberLifetimeAttribute** (in that order) to determine if it should be kept alive on removal.

If the item in the collection is a **System.Windows.FrameworkElement**, it will also check its **DataContext** for **IRegionMemberLifetime** or the **RegionMemberLifetimeAttribute**.

The region items are checked in the following order:

1. **IRegionMemberLifetime.KeepAlive** value
 2. **DataContext's IRegionMemberLifetime.KeepAlive** value
 3. **RegionMemberLifetimeAttribute.KeepAlive** value
 4. **DataContext's RegionMemberLifetimeAttribute.KeepAlive** value
-

Control-Specific Behaviors

The **SelectorItemsSourceSyncBehavior** is used only for controls that derive from **Selector**, such as a tab control in WPF. It is responsible for synchronizing the views in the region with the items of the selector, and then synchronizing the active views in the region with the selected items of the selector.

Extending the Region Implementation

The Prism Library provides extension points that allow you to customize or extend the default behavior of the provided APIs. For example, you can write your own region adapters, region behaviors, or change the way the Navigation API parses URIs. For more information about extending the Prism Library, see [Extending the Prism Library](#).

View Composition

View composition is the constructing of a view. In composite applications, views from multiple modules have to be displayed at run time in specific locations within the application UI. To achieve this, you need to define the locations where the views will appear and how the views will be created and displayed in those locations.

Views can be created and displayed in the locations either automatically through view discovery, or programmatically through view injection. These two techniques determine how individual views are mapped to named locations within the application UI.

View Discovery

In view discovery, you set up a relationship in the **RegionViewRegistry** between a region's name and the type of a view. When a region is created, the region looks for all the **ViewTypes** associated with the region and automatically instantiates and loads the corresponding views. Therefore, with view discovery, you do not have explicit control over when the views that correspond to a region are loaded and displayed.

View Injection

In view injection, your code obtains a reference to a region, and then programmatically adds a view into it. Typically, this is done when a module initializes or as a result of a user action. Your code will query a **RegionManager** for a specific region by name and then inject views into it. With view injection, you have more control over when views are loaded and displayed. You also have the ability to remove views from the region. However, with view injection, you cannot add a view to a region that has not yet been created.

Navigation

The Prism Library 4.0 contains Navigation APIs. The Navigation APIs simplify the view injection process by allowing you to navigate a region to an URI. The Navigation API instantiates the view, adds it to the region, and then activates it. Additionally, the Navigation API allows navigating back to a previously created view contained in a region. For more information about the Navigation APIs, see [Navigation](#).

When to Use View Discovery vs. View Injection

Choosing which view loading strategy to use for a region depends on the application requirements and the function of the region.

Use view discovery in the following situations:

- Automatic view loading is desired or required.
- Single instances of a view will be loaded into the region.

Use view injection in the following situations:

- Your application uses the Navigation APIs.
- You need explicit or programmatic control over when a view is created and displayed, or you need to remove a view from a region; for example, as a result of application logic or navigation.
- You need to display multiple instances of the same views in a region, where each view instance is bound to different data.
- You need to control which instance of a region a view is added to. For example, you want to add a customer detail view to a specific customer detail region. (This scenario requires implementing scoped regions as described later in this topic.)

UI Layout Scenarios

In composite applications, views from multiple modules are displayed at run time in specific locations within the application UI. To achieve this, you need to define the locations where the views will appear and how the views will be created and displayed in those locations.

The decoupling of the view and the location in the UI in which it will be displayed allows the appearance and layout of the application to evolve independently of the views that appear within the region.

The next sections describe the core scenarios you will encounter when you develop a composite application. When appropriate, examples from the Stock Trader RI will be used to demonstrate a solution for the scenario.

Implementing the Shell

The shell is the application root object in which the primary UI content is contained. In a Windows Presentation Foundation (WPF) application, the shell is the **Window** object.

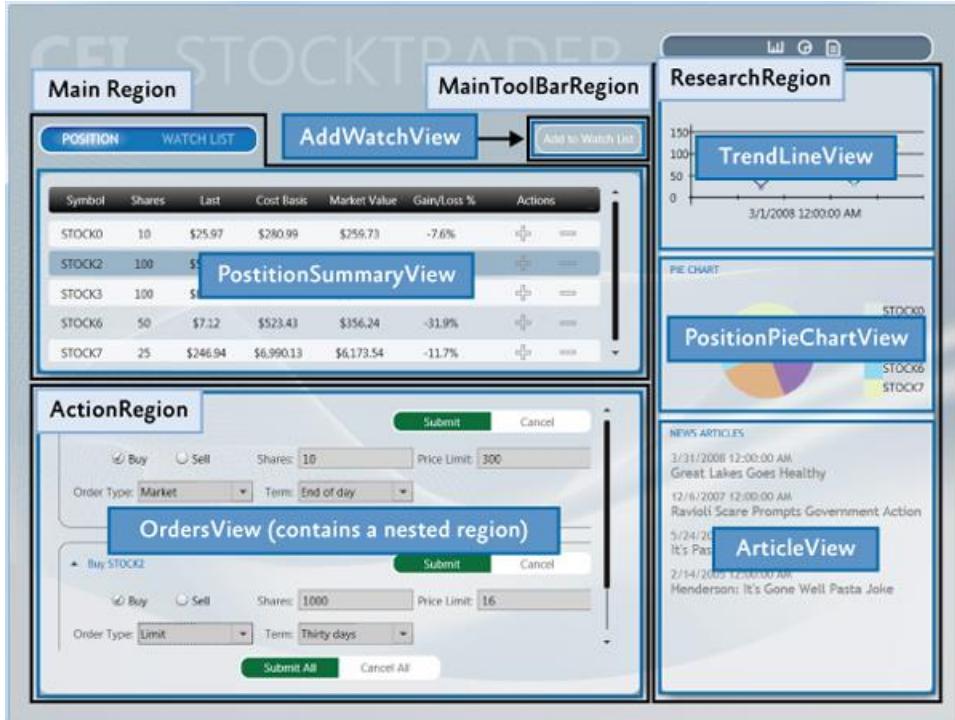
A shell can contain named regions where modules can specify the views that will appear. It can also define certain top-level UI elements, such as the main menu and toolbar. The shell defines the overall structure and appearance for the application, and is similar to an ASP.NET master page control. It could define styles and borders that are present and visible in the shell layout itself, and it could also define styles, templates, and themes that are applied to the views that are plugged into the shell.

You do not need to have a distinct shell as part of your application architecture to use the Prism Library. If you are building a completely new composite application, implementing a shell provides a well-defined root and initialization pattern for setting up the main UI of your application. However, if you are adding Prism Library features to an existing application, you do not have to change the basic architecture of your application to add a shell. Instead, you can alter your existing window definitions or controls to add regions that can pull in views as needed.

You can also have more than one shell in your application. If your application is designed to open more than one top-level window for the user, each top-level window acts as shell for the content it contains.

Stock Trader RI Shell

The WPF Stock Trader RI has a shell as its main window. In the following illustration, the shell and views are highlighted. The shell is the main window that appears when the Stock Trader RI starts and which contains all the views. It defines the regions into which modules add their views and a couple of top-level UI items, including the CFI Stock Trader title and the Watch List tear-off banner.



Stock Trader RI shell window, regions, and views

The shell implementation in the Stock Trader RI is provided by Shell.xaml, its code-behind file Shell.xaml.cs, and its view model ShellViewModel.cs. Shell.xaml includes the layout and UI elements that are part of the shell, including definitions of regions to which modules add their views.

The following XAML shows the structure and main XAML elements that define the shell. Notice that the **RegionName** attached property is used to define the four regions and that the window background image provides a background for the shell.

XAML

```
<!--Shell.xaml (WPF) -->
<Window x:Class="StockTraderRI.Shell">

    <!--shell background -->
    <Window.Background>
        <ImageBrush ImageSource="Resources/background.png" Stretch="UniformToFill"/>
    </Window.Background>

    <Grid>

        <!-- logo -->
        <Canvas x:Name="Logo" ...>
            <TextBlock Text="CFI" ... />
            <TextBlock Text="STOCKTRADER" .../>
        </Canvas>

        <!-- main bar -->
```

```

<ItemsControl
  x:Name="MainToolbar"
  prism:RegionManager.RegionName="{x:Static inf:RegionNames.MainToolBarRegion}">
</ItemsControl>

<!-- content -->
<Grid>
  <Controls:AnimatedTabControl
    x:Name="PositionBuySellTab"
    prism:RegionManager.RegionName="{x:Static inf:RegionNames.MainRegion}" />
</Grid>

<!-- details -->
<Grid>
  <ContentControl
    x:Name="ActionContent"
    prism:RegionManager.RegionName="{x:Static inf:RegionNames.ActionRegion}">
  </ContentControl>
</Grid>

<!-- sidebar -->
<Grid x:Name="SideGrid">
  <Controls:ResearchControl
    prism:RegionManager.RegionName="{x:Static inf:RegionNames.ResearchRegion}">
  </Controls:ResearchControl>
</Grid>

</Grid>
</Window>

```

The implementation of the **Shell** code-behind file is very simple. The **Shell** is exported so that when the bootstrapper creates it, its dependencies will be resolved by the Managed Extensibility Framework (MEF). The shell has its single dependency—the **ShellViewModel**—injected during construction, as shown in the following example.

C#

```

// Shell.xaml.cs
[Export]
public partial class Shell : Window
{
  public Shell()
  {
    InitializeComponent();
  }

  [Import]
  ShellViewModel ViewModel
  {

```

```
    set
    {
        this.DataContext = value;
    }
}
```

C#

```
// ShellViewModel.cs
[Export]
public class ShellViewModel : BindableBase
{
    // This is where any view model logic for the shell would go.
}
```

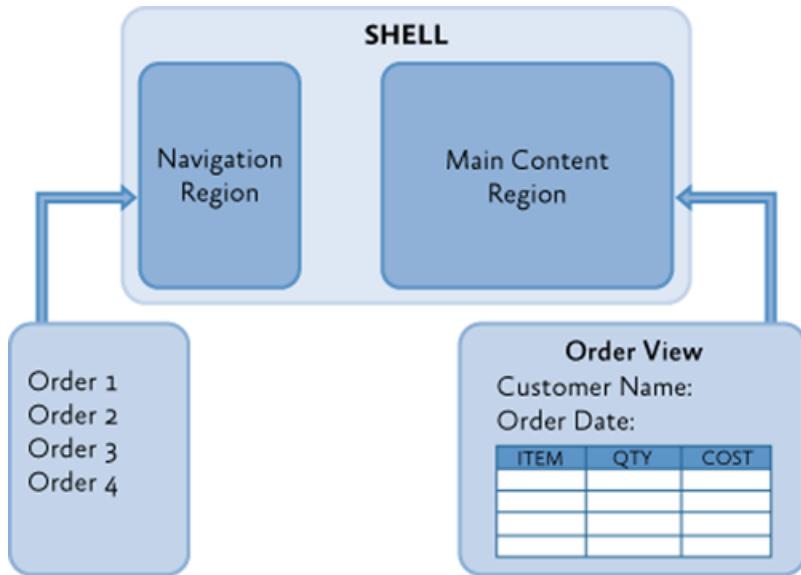
The minimal code in the code-behind file illustrates the power and simplicity of the composite application architecture and loose coupling between the shell and its constituent views.

Defining Regions

You define where views will appear by defining a layout with named locations, known as regions. Regions act as placeholders for one or more views that will be displayed at run time. Modules can locate and add content to regions in the layout without knowing how and where the region is displayed. This allows the layout to change without affecting the modules that add the content to the layout.

Regions are defined by assigning a region name to a WPF control, either in XAML as shown in the previous Shell.xaml file or in code. Regions can be accessed by their region name. At run time, views are added to the named Region control, which then displays the view or views according to the layout strategy that the views implement. For example, a tab control region will lay out its child views in a tabbed arrangement. Regions support the addition or removal of views. Views can be created and displayed in regions either programmatically or automatically. In the Prism Library, the former is achieved through view injection and the latter through view discovery. These two techniques determine how individual views are mapped to the named regions within the application UI.

The shell of the application defines the application layout at the highest level; for example, by specifying the locations for the main content and the navigation content, as shown in the following illustration. Layout within these high-level views is similarly defined, allowing the overall UI to be recursively composed.



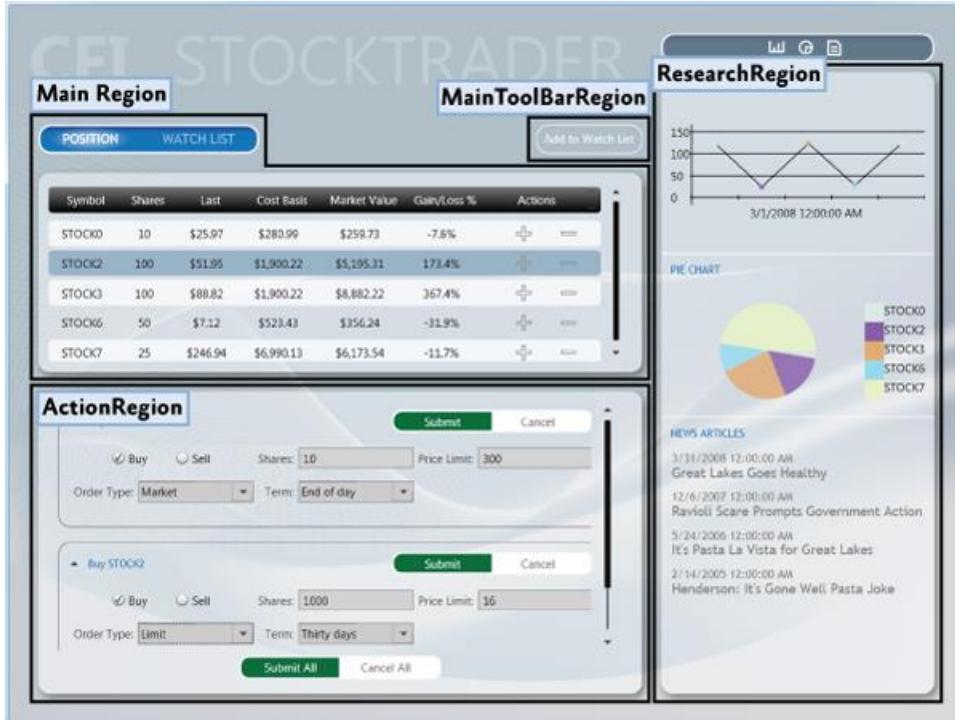
A template shell

Regions are sometimes used to define locations for multiple views that are logically related. In this scenario, the region control is typically an **ItemsControl**-derived control that will display the views according to the layout strategy that it implements, such as in a stacked or tabbed layout arrangement.

Regions can also be used to define a location for a single view; for example, by using a **ContentControl**. In this scenario, the region control displays only one view at a time, even if more than one view is mapped to that region location.

Stock Trader RI Shell Regions

The Stock Trader RI shows the use of both the single view and the multiple view layout approaches. You can see both in the shell for the application, which defines locations for the application's high-level views. The following illustration shows the regions defined by the Stock Trader RI shell.



Stock Trader RI shell regions

A multiple-view layout is also demonstrated in the Stock Trader RI when the application is buying or selling a stock. The Buy/Sell area is a list-style region that shows multiple buy/sell views (**OrderCompositeView**) as part of its list, as shown in the following illustration.



ItemsControl region

The shell's **ActionRegion** contains the **OrdersView**. The **OrdersView** contains the **Submit All** and **Cancel All** buttons as well as the **OrdersRegion**. The **OrdersRegion** is attached to a **ListBox** control which displays multiple **OrderCompositeViews**.

IRegion

A region is a class that implements the **IRegion** interface. The region is the container that holds content to be displayed by a control. The following code shows the **IRegion** interface.

C#

```
public interface IRegion : INavigateAsync, INotifyPropertyChanged
{
    IVsViewsCollection Views { get; }
    IVsViewsCollection ActiveViews { get; }
    object Context { get; set; }
    string Name { get; set; }
    Comparison<object> SortComparison { get; set; }
    IRegionManager Add(object view);
    IRegionManager Add(object view, string viewName);
    IRegionManager Add(object view, string viewName, bool createRegionManagerScope);
    void Remove(object view);
    void Deactivate(object view);
    object GetView(string viewName);
    IRegionManager RegionManager { get; set; }
    IRegionBehaviorCollection Behaviors { get; }
    IRegionNavigationService NavigationService { get; set; }
}
```

Adding a Region in XAML

The **RegionManager** supplies an attached property that you can use for simple region creation in XAML. To use the attached property, you must load the Prism Library namespace into the XAML and then use the **RegionName** attached property. The following example shows how to use the attached property in a window with an **AnimatedTabControl**.

Notice the use of the **x:Static** markup extension to reference the **MainRegion** string constant. This practice eliminates magic strings in the XAML.

XAML

```
<!--(WPF) -->
<Controls:AnimatedTabControl
    x:Name="PositionBuySellTab"
    prism:RegionManager.RegionName="{x:Static inf:RegionNames.MainRegion}">
```

Adding a Region by Using Code

The **RegionManager** can register regions directly without using XAML. The following code example shows how to add a region to a control from the code-behind file. First a reference to the region manager is obtained. Then, using the **RegionManager** static methods **SetRegionManager** and

SetRegionName, the region is attached to the UI's **ActionContent** control and then that region is named **ActionRegion**.

C#

```
IRegionManager regionManager = ServiceLocator.Current.GetInstance<IRegionManager>();
RegionManager.SetRegionManager(this.ActionContent, regionManager);
RegionManager.SetRegionName(this.ActionContent, "ActionRegion");
```

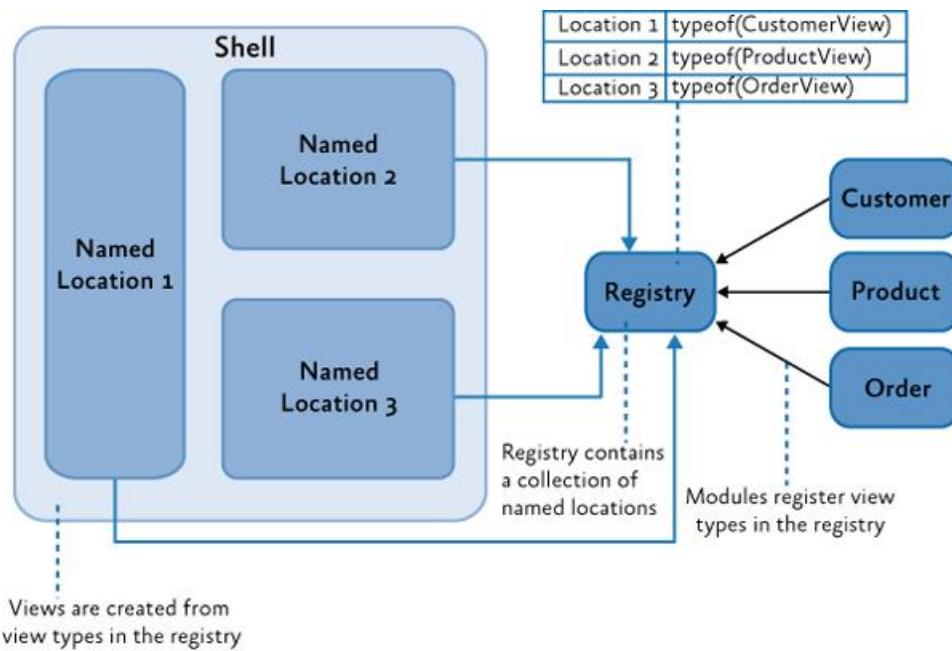
Displaying Views in a Region When the Region Loads

With the view discovery approach, modules can register views (view models or presentation models) for a specific named location. When that location is displayed at run time, any views that have been registered for that location will be created and displayed within it automatically.

Modules register views with a registry. The parent view queries this registry to discover the views that were registered for a named location. After they are discovered, the parent view places those views on the screen by adding them to the placeholder control.

After the application is loaded, the composite view is notified to handle the placement of new views that have been added to the registry.

The following illustration shows the view discovery approach.



The Prism Library defines a standard registry, **RegionViewRegistry**, to register views for these named locations.

To show a view in a region, register the view with the region manager, as shown in the following code example. You can directly register a view type with the region, in which case the view will be

constructed by the dependency injection container and added to the region when the control hosting the region is loaded.

C#

```
// View discovery
this.regionManager.RegisterViewWithRegion("MainRegion", typeof(EmployeeView));
```

Optionally, you can provide a delegate that returns the view to be shown, as shown in the next example. The region manager will display the view when the region is created.

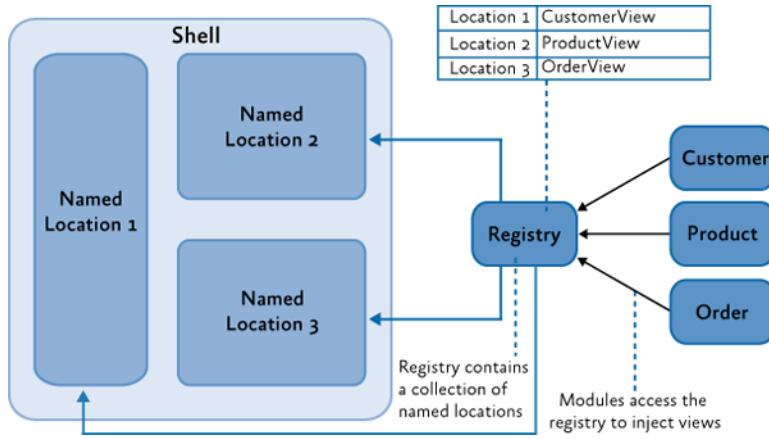
C#

```
// View discovery
this.regionManager.RegisterViewWithRegion("MainRegion", () =>
this.container.Resolve<EmployeeView>());
```

The UI Composition QuickStart has a walkthrough in the EmployeeModule ModuleInit.cs file that demonstrates how to use the **RegisterViewWithRegion** method.

Displaying Views in a Region Programmatically

In the view injection approach, views are programmatically added or removed from a named location by the modules that manage them. To enable this, the application contains a registry of named locations in the UI. A module can use the registry to look up one of the locations and then programmatically inject views into it. To make sure that locations in the registry can be accessed similarly, each of the named locations adheres to a common interface used to inject the view. The following illustration shows the view injection approach.



View injection

The Prism Library defines a standard registry, **RegionManager**, and a standard interface, **IRegion**, for access these locations.

To use view injection to add a view to a region, get the region from the region manager, and then call the **Add** method, as shown in the following code. With view injection, the view is displayed only after the view is added to a region, which can happen when the module is loaded or when a user action completes a predefined action.

C#

```
// View injection
IRegion region = regionManager.Regions["MainRegion"];

var ordersView = container.Resolve<OrdersView>();
region.Add(ordersView, "OrdersView");
region.Activate(ordersView);
```

In addition to the Stock Trader RI, the UI Composition QuickStart has a walkthrough for implementing view injection.

Navigation

The Prism Library 5.0 includes Navigation APIs that provide a rich and consistent API for implementing navigation in a WPF application.

Region navigation is a form of view injection. When a navigation request is processed, it will attempt to locate a view in the region that can fulfill the request. If it cannot find a matching view, it calls the application container to create the object, and then injects the object into the target region and activates it.

The following code example from the Stock Trader RI **ArticleViewModel** illustrates how to initiate a navigation request.

C#

```
this.regionManager.RequestNavigate(RegionNames.SecondaryRegion,
    new Uri("/NewsReaderView", UriKind.Relative));
```

For more information about region navigation, see [Navigation](#). The [View-Switching Navigation QuickStart](#) and [State-Based Navigation QuickStart](#) are also examples of implementing application navigation.

Ordering Views in a Region

Whether it uses view discovery or view Injection, an application might need to order how views appear in a **TabControl**, **ItemsControl**, or any other control that displays multiple active views. By default, views appear in the order that they were registered and added into the region.

When a composite application is built, views are often registered from different modules. Declaring dependencies between modules can help alleviate the problem, but when modules and views do not have any real interdependencies, declaring an artificial dependency couples modules unnecessarily.

To allow views to participate in ordering themselves, the Prism Library provides the **ViewSortHint** attribute. This attribute contains a string **Hint** property that allows a view to declare a hint of how it should be ordered in the region.

When displaying views, the **Region** class uses a default view sorting routine that uses the hint to order the views. This is a simple case-sensitive ordinal sort. Views that have the sort hint attribute are ordered

ahead of those without. Also, those without the attribute appear in the order they were added to the region.

If you want to change how views are ordered, the **Region** class provides a **SortComparison** property that you can set with your own **Comparison<object>** delegate method. It is important to note that the ordering of the region's **Views** and **ActiveViews** properties are reflected in the UI because adapters such as the **ItemsControlRegionAdapter** bind directly to these properties. A custom region adapter could implement its own sorting and filter that will override how the region orders views.

The View Switching QuickStart demonstrates a simple numbering scheme to order the views in the left-hand-side navigation region. The following code examples show **ViewSortHint** applied to each of the navigation item views.

C#

```
[Export]
[ViewSortHint("01")]
public partial class EmailNavigationItemView

[Export]
[ViewSortHint("02")]
public partial class CalendarNavigationItemView

[Export]
[ViewSortHint("03")]
public partial class ContactsDetailNavigationItemView

[Export]
[ViewSortHint("04")]
public partial class ContactsAvatarNavigationItemView
```

Sharing Data Between Multiple Regions

The Prism Library provides multiple approaches to communicating between views, depending on your scenario. The region manager provides the **RegionContext** property as one of these approaches.

RegionContext is useful when you want to share context between a parent view and child views that are hosted in a region. **RegionContext** is an attached property. You set the value of the context on the region control so that it can be made available to all child views that are displayed in that region control. The region context can be any simple or complex object and can be a data-bound value. The **RegionContext** can be used with either view discovery or view injection.

Note: The **DataContext** property in WPF is used to set the local data context for the view. It allows the view to use data binding to communicate with a view model, local presenter, or model. **RegionContext** is used to share context between multiple views and is not local to a single view. It provides a simple mechanism for sharing context between multiple views.

The following code shows how the **RegionContext** attached property is used in XAML.

XAML

```
<TabControl AutomationProperties.AutomationId="DetailsTabControl"
    prism:RegionManager.RegionName="{x:Static local:RegionNames.TabRegion}"
    prism:RegionManager.RegionContext="{Binding Path=SelectedEmployee.EmployeeId}"
    ...>
```

You can also set the **RegionContext** in code, as shown in the following example.

C#

```
RegionManager.Regions["Region1"].Context = employeeId;
```

To retrieve the **RegionContext** in a view, the **GetObservableContext** static method of the **RegionContext** class is used. It passes the view as a parameter and then accesses its **Value** property, as shown in the following code example.

C#

```
private void GetRegionContext()
{
    this.Model.EmployeeId = (int)RegionContext.GetObservableContext(this).Value;
}
```

The value of the **RegionContext** can be changed from within a view by simply assigning a new value to its **Value** property. Views can opt to be notified of changes to the **RegionContext** by subscribing to the **PropertyChanged** event on the **ObservableObject** that is returned by the **GetObservableContext** method. This allows multiple views to be kept in synchronization when their **RegionContext** is changed. The following code example demonstrates subscribing to the **PropertyChanged** event.

C#

```
ObservableObject<object> viewRegionContext =
    RegionContext.GetObservableContext(this);
viewRegionContext.PropertyChanged += this.ViewRegionContext_OnPropertyChanged;

private void ViewRegionContext_OnPropertyChanged(object sender,
                                                PropertyChangedEventArgs args)
{
    if (args.PropertyName == "Value")
    {
        var context = (ObservableObject<object>) sender;
        int newValue = (int)context.Value;
    }
}
```

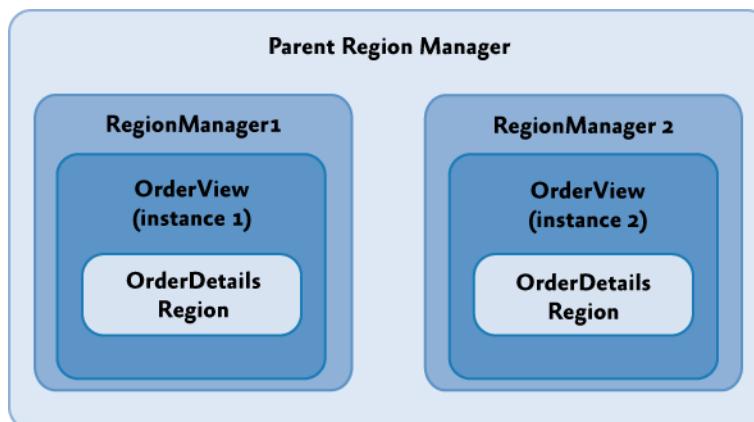
Note: The **RegionContext** is set as an attached property on the content object hosted in the region. This means that the content object has to derive from **DependencyObject**. In the preceding example, the view is a visual control, which ultimately derives from **DependencyObject**.

If you choose to use WPF data templates to define your view, the content object will represent the **ViewModel** or **PresentationModel**. If your view model or presentation model needs to retrieve the **RegionContext**, it will need to derive from the **DependencyObject** base class.

Creating Multiple Instances of a Region

Scoped regions are available only with view injection. You should use them if you need a view to have its own instance of a region. Views that define regions with attached properties automatically inherit their parent's **RegionManager**. Usually, this is the global **RegionManager** that is registered in the shell window. If the application creates more than one instance of that view, each instance would attempt to register its region with the parent **RegionManager**. **RegionManager** allows only uniquely named regions; therefore, the second registration would produce an error.

Instead, use scoped regions so that each view will have its own **RegionManager** and its regions will be registered with that **RegionManager** rather than the parent **RegionManager**, as shown in the following illustration.



Parent and scoped RegionManagers

To create a local **RegionManager** for a view, specify that a new **RegionManager** should be created when you add your view to a region, as illustrated in the following code example.

C#

```
IRegion detailsRegion = this.regionManager.Regions["DetailsRegion"];
View view = new View();
bool createRegionManagerScope = true;
IRegionManager detailsRegionManager = detailsRegion.Add(view, null,
createRegionManagerScope);
```

The **Add** method will return the new **RegionManager** that the view can retain for further access to the local scope.

Creating Views

The visual representation of your application can take many forms, including user controls, custom controls, and data templates, to name a few. In the case of the Stock Trader RI, user controls are typically used to represent distinct sections on the main window, but this is not a standard. In your application, you should use an approach that you are most familiar with and that fits into how you work as a designer. Regardless of the predominating visual representation in your application, you will inevitably use a combination of user controls, custom controls, and data templates in your overall design. The following figure shows where the Stock Trader RI uses these various items. This illustration also serves as a reference for the following sections, which describe each of the items.



Stock Trader RI usage of user controls, custom controls, and data templates

User Controls

Both Blend for Visual Studio 2013 and Visual Studio 2013 provide rich support for creating user controls. User controls created with these tools are therefore recommended for creating UI content with the Prism Library. As mentioned earlier in this topic, the Stock Trader RI uses them extensively to create content that will be inserted into regions. The **WatchListView.xaml** user control is a good example of a simple UI representation that is contained inside the **WatchModule**. This control is a very simple control that is straightforward to create using this model.

Custom Controls

In some situations, a user control is too limiting. In these cases, custom layout or extensibility is more important than ease of creation. This is where custom controls are useful. In the Stock Trader RI, the pie chart control is a good example of this. This control is composed from data derived from the positions and shows a chart of the overall portfolio. This type of control is a little more challenging than a user control to create, and it has limited visual design support in Blend for Visual Studio 2013 and Visual Studio 2013, compared to a user control.

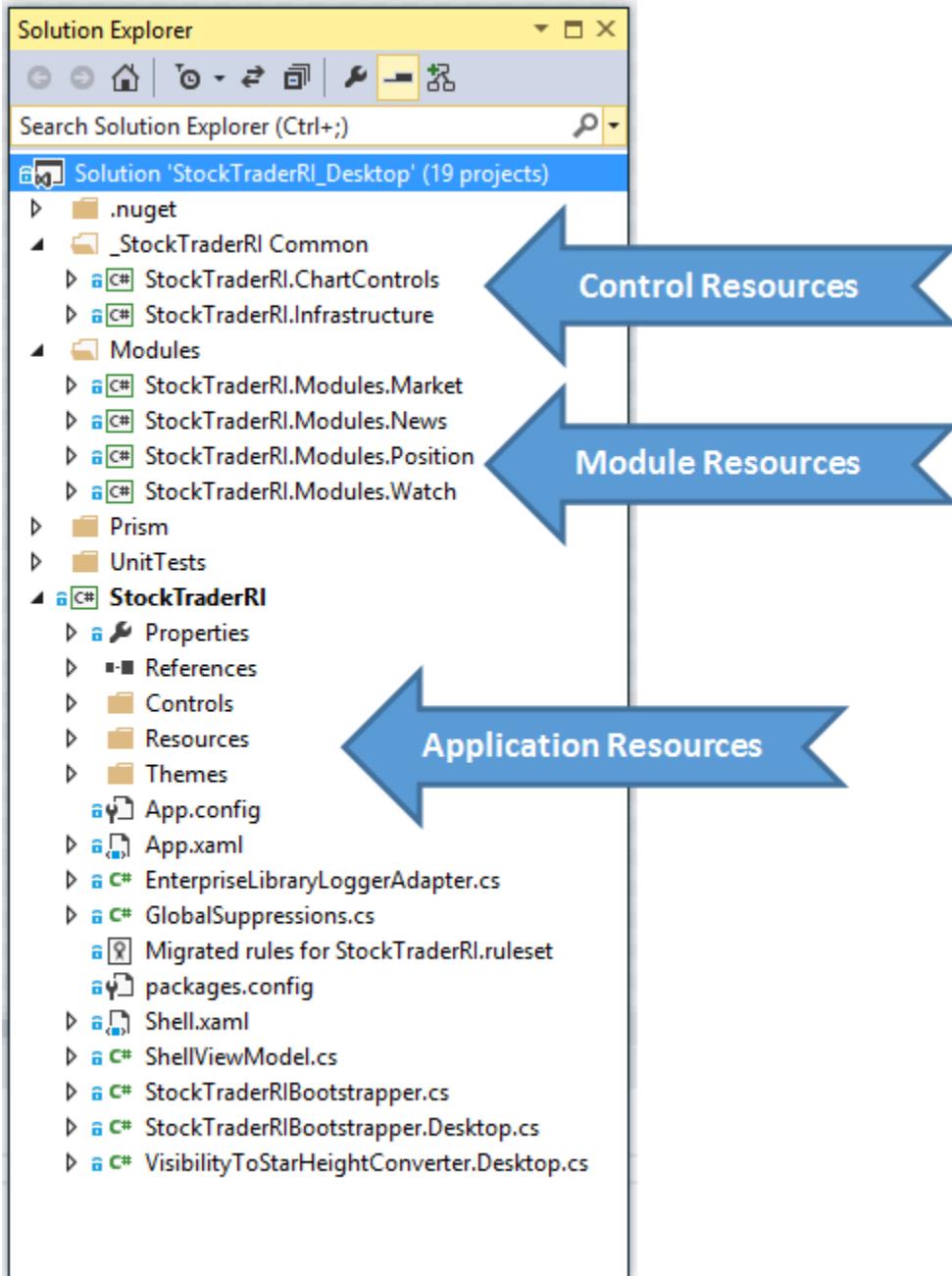
Data Templates

Data templates are an important part of most types of data-driven applications. The use of data templates for list-based controls is prevalent throughout the Stock Trader RI. In many cases, you can use a data template to create complete visual representations without needing to create any type of control. The **ResearchRegion** uses a data template to show articles and, in conjunction with an **Items** style, provides an indication of which item was selected.

Blend for Visual Studio 2013 and Visual Studio 2013 have full visual design support for data templates.

Resources

Resources such as styles, resource dictionaries, and control templates can be scattered throughout an application. This is especially true with a composite application. When you consider where to place resources, pay special attention to dependencies between UI elements and the resources they need. The Stock Trader RI solution, shown in the following figure, contains labels that indicate the various areas where resources can live.



Resource distribution across a solution

Application Resources

Typically, application resources are resources that are available to an application as a whole. These resources tend to be focused on the root application, but they can also provide default styling on a type basis for modules or controls. An example of this is a text box style that is applied to the text box type in the root application. This style will be available to all text boxes in the application unless the style is overridden at the module or control level.

Module Resources

Module resources play the same role as root application resources in that they can apply to all items in a module. Using resources at this level can provide a consistent appearance across the entire module and can also allow for reuse in more specific instances that span one or more visual components. The use of resources at the module level should be contained within the individual module. Creating dependencies between modules can lead to issues that are difficult to locate when UI elements appear incorrectly.

Control Resources

Control resources are usually contained in control libraries and can be used by all the controls in the control library. These resources tend to have the most limited scope because control libraries typically contain very specific controls and do not contain user controls. (In an application created with the Prism Library, user controls are typically placed in the modules in which they are used.)

UI Design Guidance

The goal of this topic is to provide some high-level guidance to the XAML designer and developer who are building an application with the Prism Library and WPF. This topic describes UI layout, visual representation, data binding, resources, and the presentation model. After reading this topic, you should have a high-level understanding of how to approach designing the UI of an application based on the Prism Library and some of the techniques that can help you create a maintainable UI in composite applications.

Guidelines for Designing User Interfaces

The layout of composite applications created with the Prism Library builds on the standard principals of WPF—the layout uses the concepts of panels that contain related items. However, with composite applications, the content inside the various panels is dynamic and is not known during design time. This forces designers and developers to create page structures that can contain layout content and then design each of the elements that fit into the layout separately. As a designer or developer, this means that you have to think about two main layout concepts in the Prism Library: container composition and regions.

Container Composition

Container composition is really just an extension of the containment model that WPF inherently provides. The term *container* can mean any element, including a window, page, user control, panel, custom control, control template, or data template, that can contain other elements.

How you visualize your UI can vary from implementation to implementation, but you will find recurring themes that stand out. You will create a window, page, or user control that contains both fixed content and dynamic content. The fixed content will consist of the overall structure of the containing UI element, and the dynamic content will be what is placed inside a region.

For example, the WPF Stock Trader RI has a startup window named Shell.xaml that contains the overall structure for the application. The next illustration shows the shell loaded in Blend for Visual Studio 2013.

Notice that only the fixed portion of the UI is visible. The remaining sections of the shell are dynamically inserted into the various regions by the modules as the application loads.

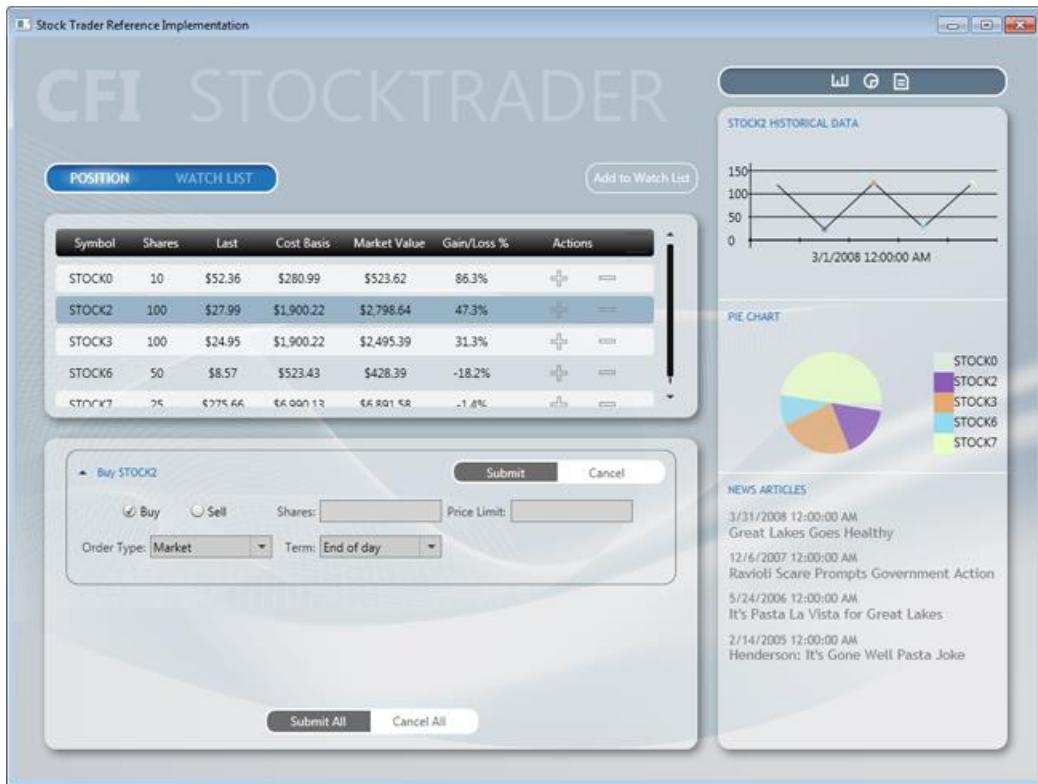
The design-time experience is a little limited in this type of application, but the fact that you know content will be placed in the various regions at run time is something that you need to design for. To see an example of this, compare the designer view of the main page in the next illustration to the run-time view in the illustration that follows it. In the designer view, the page is mostly empty. Contrast that with the run-time view, where there is a position area that contains a tab control with position data, and a trend line, pie chart, and news area pertaining to the selected stocks. The differences between the designer view and run-time view demonstrate the challenges designers and developers face when they create applications built with the Prism Library.

The items cannot be seen during design time; therefore, determining how big they are and how they fit into the overall appearance of the application is a little difficult. Consider the following as you create the layout for your containers:

- Are there any size constraints that will limit how large content can be? If there are, consider using containers that support scrolling.
- Consider using an expander and **ScrollViewer** combination for situations in which a large amount of dynamic content needs to fit into a confined area.
- Pay close attention to how content enlarges as the screen content grows to ensure that the appearance of your application is appealing in any resolution.



Stock Trader RI main window in Blend for Visual Studio 2013



Stock Trader RI main window during run time

Viewing Composite Application at Design Time

The two previous figures illustrate one of the challenges of working with high-level views that are composed at run time. Each UI element in a composite application must be designed separately. This makes it hard to visualize how the composite page or window will look at run time. To visualize the composite view in its composed state, you can create a test project with a page or window that contains all the UI elements for the view you want to test.

Additionally, consider using the design-time sample data features in Blend for Visual Studio 2013 and Visual Studio 2013 to populate UI elements with data. Design-time data is very helpful when you work with data templates, list controls, charts, or graphs. For more information, see the section [Guidelines for Design-Time Sample Data](#).

Layout

Consider the following when you design the layout of a composite application:

- The shell defines the main layout of the application. Each area of the layout is a region and should be kept as an empty container. Do not place content inside regions at design time because content will be loaded there at run time.

- The shell should contain the background, titles, and the footer. Think of the shell as an ASP.NET master page.
- Control containers that act as regions are decoupled from the views that they contain. Therefore, you should be able to change the size of the views without modifying the controls, and you should be able to change the size of the controls without modifying the views. You should consider the following when defining the size of a view:
 - If a view will be used in several regions or if it is uncertain where it will be used, design it with dynamic width and height.
 - If the views have fixed sizes, the regions of the shell should use dynamic sizes.
 - If the shell regions have fixed sizes, the views should use dynamic sizes.
 - Views might require a fixed height and dynamic width. An example of this is the **PositionPieChart** view located in the sidebar of the Stock Trader RI.
 - Other views might require a dynamic height and width. For example, the **NewsReader** views in the sidebar of the Stock Trader RI. The height itself depends on the title's length, and the width should always adapt to the region's size (sidebar width). The same applies to the **PositionSummaryView** view, where the grid's width should adapt to the screen size and the height should adapt to the number of rows in the grid.
- Views should generally have transparent backgrounds, allowing the shell background to provide the application visual background.
- Always use named resources for assigning colors, brushes, fonts and font sizes, rather than directly assigning the property value in XAML. This makes application maintenance much easier over time. It also allows an application to respond to changes in resource dictionaries at run time.

Animation

Consider the following when you use animation in the shell or views:

- You can animate the layout of the shell, but you will have to animate its contents and views separately.
- Design and animate each view separately.
- Use soft or gentle animations to provide a visual clue that a UI element is being brought into view or being removed from view. This gives an application a polished look and feel.

Blend for Visual Studio 2013 offers a rich set of behaviors, easing functions, and an outstanding editing experience for animating and transitioning UI elements based on visual state changes or events. For more information, see [VisualStateManager Class](#) on MSDN.

Run-Time Optimization

Consider the following tips for performance optimization:

- Place any common resources in the App.xaml file or a merged dictionary to avoid duplicating the styles.
-

Design-Time Optimizations

The following sections describe design-time scenarios and provide solutions for making the most of the design-time experience.

Large Solutions with Many XAML Resources

In large applications with many XAML resources that are part of the solution, visual designer load time can be affected, sometimes significantly. This performance slowdown exists because the visual designer must parse all merged XAML resources. The solution to this problem is to move all XAML resources to another solution, compile that solution, and then reference the new XAML resource DLL from the large solution. Because the XAML resources are in a binary referenced assembly, the visual designer does not parse the XAML resources, thus improving design-time performance. When moving XAML resources to an external assembly, you might want to consider exposing **ComponentResourceKeys** for your resources. For more information, see [ComponentResourceKey Markup Extension](#) on MSDN.

XAML Assets

XAML is a powerful and expressive language for creating assets such as images, diagrams, drawings, and 3-D scenes. Some developers and designers prefer creating XAML assets instead of using .ico, .jpg, or .png image files. One reason that they prefer the XAML approach is to take advantage of the resolution independence of XAML rendering. Another is that they can use one tool set, Blend for Visual Studio 2013, to create all the required assets and design their applications.

If the solution has many of these assets, design-time visual designer loading can be affected. Moving assets to a separate DLL solves the performance problem. Moving the assets also enables reuse across multiple solutions.

Visual Designers and Referenced Assemblies

An unfortunate side-effect of moving XAML resources and assets to a binary referenced assembly is that the Blend for 2013 and Visual Studio 2013 property editors do not list resources located in binary referenced assemblies. This means that you will not be able to pick a named resource from one of the resource pickers provided by the tools. Instead, you will need to type the name of the resource.

Guidelines for Creating Designer Friendly Views

The following are some of the characteristics of a designer friendly (also known as a *blendable* or *toolable*) application:

- It provides a productive editing experience by using the Visual Studio and Blend designers.
- It is tooling-enabled. For example, it allows you to use the binding builder.

- It provides design-time sample data when required.
 - It allows code to be executed at design time without causing unhandled exceptions.
-

The following actions are performed many times during an editing session. User code that is not designer friendly will cause one or more of these actions to fail, thus reducing the productivity and creativity of a developer or designer.

- Design surface actions:
 - Constructing objects
 - Loading objects
 - Setting property values
 - Performing design surface gestures
 - Using a control as the root element
 - Hosting a control inside another control
 - Opening, closing, and reopening a XAML file repeatedly
 - Rebuilding the project
 - Reloading the designer
 - Binding builder actions:
 - Discovering the **DataContext**
 - Listing the available data sources
 - Listing data source type properties
 - Design-time sample data actions:
 - Using controls on the design surface to correctly display sample data
-

Coding for Design Time

To give you a rich design-time experience, the Visual Studio and Blend designers instantiate objects and run code at design time. However, null reference exceptions caused by code that attempts to access a reference type before it has been instantiated cause a high percentage of loading failures and unnecessary design time exceptions.

The following table lists the main causes of poor design-time experiences. By avoiding the following issues and using the techniques to mitigate these problems, your design-time experience and productivity will be greatly enhanced, and the developer-to-designer workflow will be much smoother.

Avoid This in User Code	Visual	Blend for
-------------------------	--------	-----------

	Studio 2013	Visual Studio 2013
Spinning multiple threads at design time. For example, instantiating and starting a Timer in a constructor or Loaded event at design time.	🚫	🚫
Using controls that cause stack overflows at design time.	🚫	🚫
Using controls that attempt to recursively load themselves.	🚫	🚫
Throwing null reference exceptions in converters or data template selectors.	🚫	🚫
Throwing null reference or other exceptions in constructors. These are caused by: <ul style="list-style-type: none"> Using code that calls into the business or data layers to return data from a database or over the network at design time. Attempting to resolve dependencies by using MEF, inversion of control (IoC), or a Service Locator before bootstrapping or container initialization code has run. 	🚫	🚫
Throwing null reference or other exceptions inside the Loaded events of controls or user controls. This happens when you make assumptions about the state of the control that might be true at run time but are not true at design time.	🚫	🚫
Attempting to access the Application or Application.Current object at design time.	🚫	🚫
Creating very large projects.	✓	🚫

Mitigating Problems in Design-Time User Code

A few defensive coding practices will eliminate most of the issues described in the preceding table. However, before you can mitigate problems in design-time user code, you must understand that your application controls and code are being executed by the designer in isolation, inside an uninitialized application domain. *Uninitialized* in this case means that the usual startup, bootstrapping, or initialization code has not run.

When your application executes at run time, the startup code in App.xaml.cs or App.xaml.vb is run. If you have code in there that the rest of your application depends on, this code will not have been executed at design time. If you have not anticipated this in your code, unwanted exceptions will occur. (This is why attempting to access the **Application** or **Application.Current** object in user code at design time will result in exceptions.) To mitigate these issues:

- Never assume that referenced objects will be instantiated in design-time code. In code that can be executed at design time, always perform a null check before accessing any reference object.
- If your code accesses the **Application** or **Application.Current** objects, perform a null reference check before accessing the object.
- If your constructors or **Loaded** event handlers need to run complex code or code that accesses a database or calls out to the network, consider one of the following solutions:

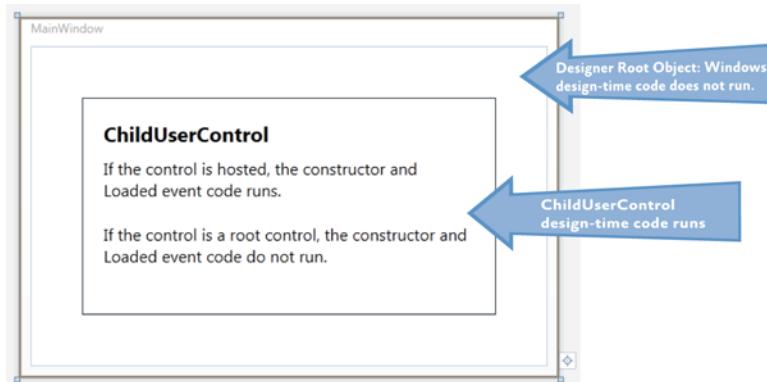
- Wrap the code inside a check that determines if the code is running at design time by calling the **System.ComponentModel DesignerProperties** method, **DesignerProperties.GetIsInDesignMode**.
- Instead of running the code directly in the constructor or **Loaded** event handler, abstract the calls to a class behind an interface, and then use one of many techniques to resolve that dependency differently at design time, run time, and test time.

For example, instead of calling out to a data service directly to retrieve data, wrap the data service calls in a class that exposes the methods through an interface. Then, at design time, resolve the interface with a mock or design-time object.

Understanding when User Control Code Executes at Design-Time

Both Blend and Visual Studio use mockups of the root object displayed in a designer pane. This is necessary to provide the required design experience. Because the root object is mocked, its constructor and **Loaded** event code are not executed at design time. However, the remaining controls in the scene are constructed normally, and their **Loaded** event is raised just like at run time.

In the following illustration, the root **Windows** constructor and **Loaded** event code will not be executed. The child user controls constructor and **Loaded** event code will be executed.



These concepts are important, especially if you are building composite applications or applications that are built dynamically at run time.

Most application views are coded and designed independently. Because they are designed independently, they are typically the root object in the designer. Because of this, their constructor and **Loaded** event code never executes.

However, if you take that same user control and place it on a design surface as a child of another control, the once isolated user control code is now executing at design time. If you have not followed the above practices for mitigating design-time code problems, the now hosted user control could become unfriendly and cause designer load issues.

Design-Time Properties

The built-in "d:" design-time properties provide a smooth road to a successful design-time tooling experience.

The problem we need to solve is how to provide a shape to the Binding Builder tools at design-time. In this case, the shape is an instantiated **Type** that the Binding Builder can reflect on, and then list those properties for selection when building a binding.

Shape is also provided by design-time sample data. Sample data is covered in the section, [Guidelines for Design-Time Sample Data](#).

The following sections describe how to use the **d:DataContext** property and the **d:DesignInstance** markup extension.

The "d:" in the property and markup extension is the alias for the design namespace that the design properties are members of. For more information see the MSDN topic, [Design-Time Attributes in the WPF Designer](#).

The "d:" properties and markup extensions cannot be created or extended in user code; they can only be used in XAML. The "d:" properties and markup extensions are not compiled into your application; they are used only by the Visual Studio and Blend tooling.

d:DataContext Property

d:DataContext, specifies a design-time data context for a control and its children. When specifying **d:DataContext**, you should always provide the same shape to the design-time **DataContext** as the run-time **DataContext**.

If both a **DataContext** and a **d:DataContext** are specified for a control, the tooling will use the **d:DataContext**.

d:DesignInstance Markup Extension

If markup extensions are new to you, read [Markup Extensions and WPF XAML](#) on MSDN.

d:DesignInstance returns an instantiated Type ("shape") that you will want to assign as the data source for binding to controls in the designer. The type does not need to be creatable to be used for establishing shape. The following table explains the **d:DesignInstance** markup extension properties.

Markup Extension Property	Definition
Type	Name of the Type that will be created. Type is the default parameter in the constructor.
IsDesignTimeCreatable	Can the specified Type be created? If false, a faux Type will be created rather than the real Type. The default is false.
CreateList	If true, returns a generic list of the specified Type. The default is false.

Typical d:DataContext Scenario

The following three code examples demonstrate a repeatable pattern for wiring up views and view models and enabling the designer's tooling.

The **PersonViewModel** is a dependency that the **PersonView** has at run time. While the view model in the example is incredibly simple, real-world view models typically have one or more external dependencies that must be resolved, and those dependencies are typically injected into their constructor.

When the **PersonView** is constructed, its dependency **PersonViewModel** will be built and its dependencies resolved by MEF or a dependency injection container.

Note: If the view model has no external dependencies that need to be resolved, the view model can be instantiated in the view's XAML, and its **DataContext** and the **d:DataContext** are not required.

C#

```
// PersonViewModel.cs
[Export]
public class PersonViewModel {

    public String FirstName { get; set; }
    public String LastName { get; set; }

}
```

C#

```
// PersonView.xaml.cs
[Export]
public partial class PersonView : UserControl
{
    public PersonView()
    {
        InitializeComponent();
    }

    [Import]
    public PersonViewModel ViewModel
    {
        get { return this.DataContext as PersonViewModel; }
        set { this.DataContext = value; }
    }
}
```

This is a good pattern for wiring up a view and view model; however, it leaves the view unaware of its **DataContext**'s shape (view model) at design time.

In the following XAML example, you can see the **d:DesignInstance** markup extension used on the **Grid** to return a faux instance of **PersonViewModel** that is then exposed by the **d:DataContext**. As a result, all

child controls of the **Grid** will inherit the **d:DataContext**, enabling the designer tooling to discover and use its types and properties, resulting in a more productive design experience for developers and designers.

XAML

```
<!--PersonView.xaml -->
<UserControl
    xmlns:local="clr-namespace:WpfApplication1"
    x:Class="WpfApplication1.PersonView"
    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"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">

    <Border BorderBrush="LightGray" BorderThickness="1" CornerRadius="10" Padding="10">

        <Grid d:DataContext="{d:DesignInstance local:PersonViewModel}">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <Label Grid.Column="0" Grid.Row="0" Content="First Name" />
            <Label Grid.Column="0" Grid.Row="1" Content="Last Name" />

            <TextBox
                Grid.Column="1" Grid.Row="0" Width="150" MaxLength="50"
                HorizontalAlignment="Left" VerticalAlignment="Top"
                Text="{Binding Path=FirstName, Mode=TwoWay}" />
            <TextBox
                Grid.Column="1" Grid.Row="1" Width="150" MaxLength="50"
                HorizontalAlignment="Left" VerticalAlignment="Top"
                Text="{Binding Path=LastName, Mode=TwoWay}" />

        </Grid>
    </Border>
</UserControl>
```

Attached Property and ViewModel Locator Solution

There are several alternative techniques for associating a view and view model available from the developer community. One of the challenges is that solutions that work great at run time do not always work at design time. One such solution is the use of attached properties and view model locators to assign a view's **DataContext**. The view model locator is required so that the view model can be constructed and have its dependencies resolved.

The problem with this solution is that you must also include the **d:DataContext – d:DesignInstance** combination because the visual designer tooling cannot be reflected in the results of the attached property the way that it can with the **d:DesignInstance**.

Regardless of which technique you implement in your applications for resolving shape at design time, the most important goal is to be consistent throughout your application. Consistency will make application maintenance much easier and will lead to a successful designer-developer workflow.

Guidelines for Design-Time Sample Data

The WPF and Silverlight Designer team published an in-depth, scenario-based training article that discusses the use of sample data in WPF and Silverlight projects. The article, [Sample Data in the WPF and Silverlight Designer](#), is available on MSDN.

Using Design-Time Sample Data

If you use a visual design tool, such as Blend or Visual Studio 2013, design-time sample data becomes very important. The views can be populated with data and images, making the design task easier and quicker to accomplish. This results in improved productivity and creativity.

Empty list controls that contain data templates will not be visible unless they are populated with data, making the task of editing the empty controls more time consuming because you need to run the application to see how the last edit will look at run time.

Sample Data Sources

You can use sample data from any of the following sources:

- Blend for Visual Studio 2013 XML sample data
- Blend for Visual Studio 2013 and Visual Studio 2013 XAML sample data
- XAML resources
- Code

The data from each of these sources is described in the following subsections.

Blend XML Sample Data

Blend gives you the capability to quickly create an XML schema and populate a corresponding XML file. This is accomplished without any dependency on any project classes.

The purpose of this type of sample data is to let designers start their projects quickly, without waiting for a developer or before application classes are available for consumption.

While most sample data is supported in both the Blend and Visual Studio designers, XML sample data is a Blend feature and does not render in the Visual Studio designer.

Note: XML sample data file is not compiled or added to the assembly when built; however, the XML schema is compiled into the built assembly.

Blend for Visual Studio 2013 and Visual Studio 2013 XAML Sample Data

Beginning in Expression Blend 4 and Visual Studio 2010, the **d:DesignData** markup extension was added to enable the design-time loading of XAML sample data.

Sample data XAML files contain XAML that instantiates one or more types and assigns values to properties.

d:DesignData has a **Source** property that takes a uniform resource identifier (URI) to the sample data XAML file located in the project. The **d:DesignData** markup extension loads the XAML file, parses it, and then returns an object graph. The object graph can be consumed by the **d:DataContext** property, **CollectionViewSource d:DesignSource** property, or **DomainDataSource d:DesignData** property.

One of the challenges that the **d:DesignData** markup extension overcomes is that it can create sample data for non-creatable user types. For example, WCF Rich Internet Application (RIA) Services entity-derived objects cannot be created in code. In addition, developers might have their own types that are not creatable, but would still like to have sample data for these types.

You can change how **d:DesignData** processes your sample data file by setting the **Build Action** property on the sample data file in the **Solution Explorer** as follows:

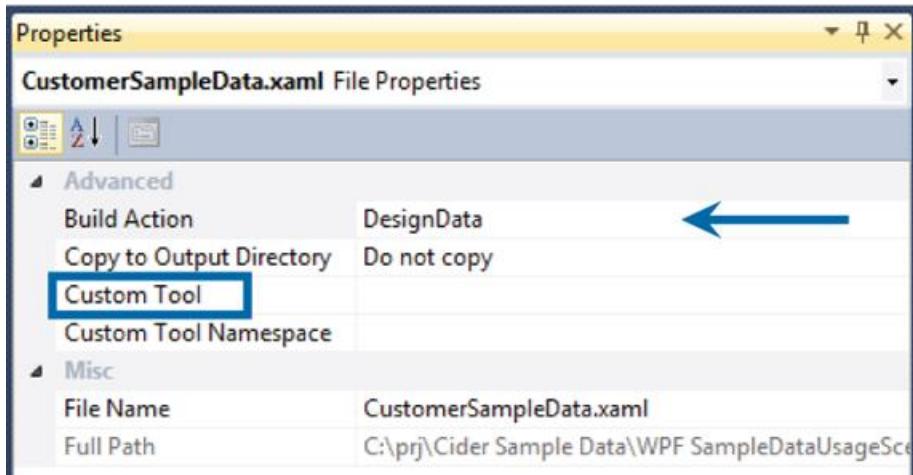
- **Build Action = DesignData** – faux types will be created
- **Build Action = DesignWithDataTimeCreatableTypes** – real types will be created

When Blend is used to create sample data for a class, it creates a XAML sample data file with the **Build Action** set to **DesignData**. If you require real types, open the solution in Visual Studio and change the **Build Action** for the sample data file to **DesignWithDataTimeCreatableTypes**.

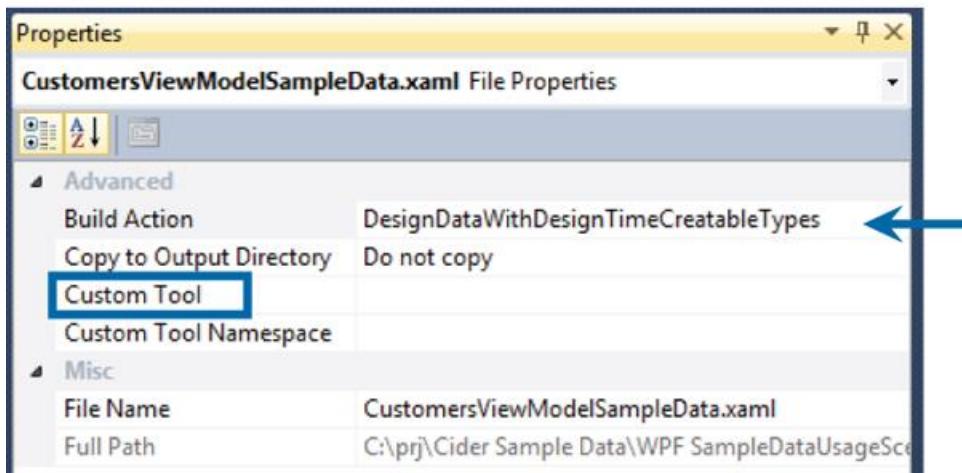
Note: In the next illustration, the **Custom Tool** property is empty. This is required for sample data to work correctly. By default, Blend correctly sets this property to empty.

When you use Visual Studio 2013 to add a sample data file, you typically add a new resource dictionary item and edit from there. In this case, you must set the **Build Action** and clear the **Custom Tool** property.

Faux type sample data property settings

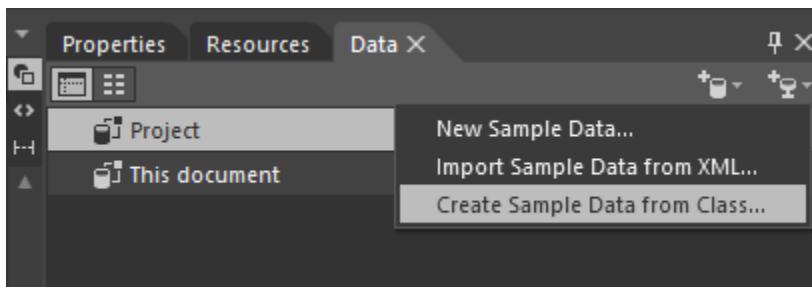


Real type sample data property settings



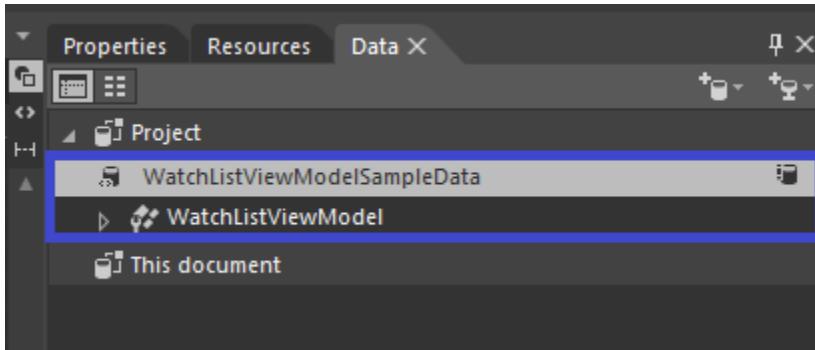
Sample data file properties

Expression Blend provides tooling for quickly creating and binding XAML sample data. The XAML sample data can be used and viewed in the Visual Studio 2013 designer, as shown in the following illustration.



Defining sample data in Blend for Visual Studio 2013

After it generates the sample data, the data will appear in the Data pane, as shown in the following illustration.



Data pane

You can then drag it onto the root element of the view, such as the **UserControl**, and have it set the **d:DataContext** property. You can also drop sample data collections onto items controls, and Blend will wire up the sample data to the control.

Note: XAML sample data files are not compiled into or included in built assemblies.

XAML Resource

You can create a resource in XAML that instantiates the desired types, and then bind that resource to a **DataContext** or list control.

This technique can be used to quickly create throw-away sample data that is used for editing a data template that would take longer to edit without the sample data.

Code

If you prefer creating sample data in code, you can write a class that exposes properties or methods that return sample data to their consumer. For example, you could write a **Customers** class that in its default empty constructor populated itself with multiple instances of the **Customer** class. Each of the **Customer** instances would have the appropriate property values set also.

One technique that you can use to consume the sample data class described previously is to use the **d:DataContext**, **d:DesignInstance** combination, ensuring that you set the **d:DesignInstance IsDesignTimeCreatable** property to **True**. The reason **IsDesignTimeCreatable** must be **True** is that you want the customers constructor to be executed so that the code to populate the class will run. If **customers** is treated as a faux type, the **customers** code will never be run and only the "shape" will be discoverable by the tooling.

The following XAML example instantiate the **Customers** class, and then sets it as the **d:DataContext**. Child controls of this **Grid** can consume data exposed by the **Customers** class.

XAML

```
<Grid d:DataContext="{d:DesignInstance local:Customers, IsDesignTimeCreatable=True}">
```

UI Layout Key Decisions

When you begin a composite application project, there are some UI design decisions that you need to make that will be difficult to change later. Generally, these decisions are application-wide and their consistency helps developers and designer productivity.

The following are the important UI layout decisions:

- Decide on application flow and define regions accordingly.
 - Decide which type of view loading each region will use.
 - Decide if you want to use the Region Navigation APIs.
 - Decide which UI Design pattern you will use (MVVM, presentation model, and so on).
 - Decide on a sample data strategy.
-

More Information

For more information about extending the Prism Library, see [Extending the Prism Library](#).

For more information about commands, see [Commands](#) in [Implementing the MVVM Pattern](#).

For more information about data binding, see [Data Binding](#) in [Implementing the MVVM Pattern](#).

For more information about region navigation, see [Navigation](#).

For more information about the guidelines discussed in this topic, see the following:

- [Dependency Properties Overview](#) on MSDN.
- Data binding; see:
 - [Data Binding Overview](#) on MSDN.
 - [Data Binding in WPF](#) in *MSDN Magazine*.
- [Data Templating Overview](#) on MSDN.
- [Resources Overview](#) on MSDN.
- [UserControl Class](#) on MSDN.
- [VisualStateManager Class](#) on MSDN.
- [Customizing Controls For Windows Presentation Foundation](#) in *MSDN Magazine*.
- [ComponentResourceKey Markup Extension](#) on MSDN.
- [Design-Time Attributes in the WPF Designer](#) on MSDN.
- [Markup Extensions and WPF XAML](#) on MSDN.
- [Sample Data in the WPF and Silverlight Designer](#) on MSDN.

- [Learning the Visual Studio WPF and Silverlight Designer](#). This contains tutorials and articles on layout, resources, data binding, sample data, debugging data bindings, object data sources, and master-detail forms.

8: Navigation

As the user interacts with a rich client application, its user interface (UI) will be continuously updated to reflect the current task and data that the user is working on. The UI may undergo considerable changes over time as the user interacts with and completes various tasks within the application. The process by which the application coordinates these UI changes is often referred to as *navigation*. This topic describes how to implement navigation for composite Model-View-ViewModel (MVVM) applications using the Prism library.

Frequently, navigation means that certain controls in the UI are removed, while other controls are added. In other cases, navigation may mean that the visual state of one or more existing controls is updated—for example, some controls may be simply hidden or collapsed, while other controls are shown or expanded. Similarly, navigation may mean that the data being displayed by a control is updated to reflect the current state of the application—for example, in a master-detail scenario, the data displayed in the detail view will be updated based on the currently selected item in the master view. All of these scenarios can be considered navigation because the user interface is updated to reflect the user's current task and the application's current state.

Navigation within an application can result from the user's interaction with the UI (via mouse events or other UI gestures) or from the application itself as a result of internal logic-driven state changes. In some cases, navigation may involve very simple UI updates that require no custom application logic. In other cases, the application may implement complex logic to programmatically control navigation to ensure that certain business rules are enforced—for example, the application may not allow the user to navigate away from a certain form without first ensuring that the data entered is correct.

Implementing the required navigation behavior in a Windows Presentation Foundation (WPF) application can often be relatively straightforward because it provides direct support for navigation. However, navigation can be more complex to implement in applications that use the Model-View-ViewModel (MVVM) pattern or in composite applications that use multiple loosely-coupled modules. Prism provides guidance on implementing navigation in these situations.

Navigation in Prism

Navigation is defined as the process by which the application coordinates changes to its UI as a result of the user's interaction with the application or internal application state changes.

UI updates can be accomplished by adding or removing elements from the application's visual tree, or by applying state changes to existing elements within the visual tree. WPF is a very flexible platform, and it is often possible to implement a particular navigation scenario using this approach. However, the approach that will be most appropriate for your application depends on multiple factors.

Prism differentiates between the two styles of navigation described earlier. Navigation accomplished via state changes to existing controls in the visual tree is referred to as *state-based navigation*. Navigation accomplished via the addition or removal of elements from the visual tree is referred to as *view-based*

navigation. Prism provides guidance on implementing both styles of navigation, focusing on the case where the application is using the Model-View-ViewModel (MVVM) pattern to separate the UI (encapsulated in the view) from the presentation logic and data (encapsulated in the view model).

State-Based Navigation

In state-based navigation, the view that represents the UI is updated either through state changes in the view model or through the user's interaction within the view itself. In this style of navigation, instead of replacing the view with another view, the view's state is changed. Depending on how the view's state is changed, the updated UI may feel to the user like navigation.

This style of navigation is suitable in the following situations:

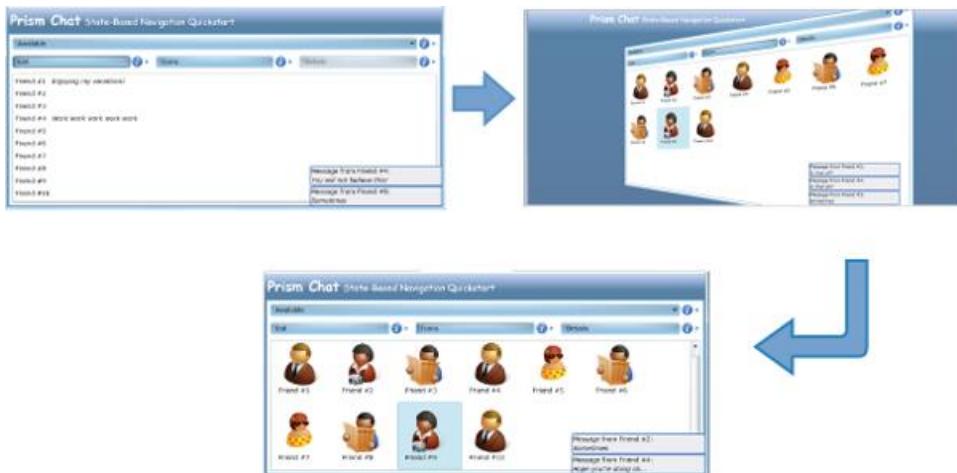
- The view needs to display the same data or functionality in different styles or formats.
- The view needs to change its layout or style based on the underlying state of the view model.
- The view needs to initiate limited modal or non-modal interaction with the user within the context of the view.

This style of navigation is not suitable for situations in which the UI has to present different data to the user or when the user has to perform a different task. In these situations, it is better to implement separate views (and view models) to represent the data or task, and then to navigate between them using view-based navigation, as described later on in this topic. Similarly, this style of navigation is not suitable if the number of UI state changes required to implement the navigation are overly complex because the view's definition can become large and difficult to maintain. In this case, it is better to implement the navigation across separate views by using view-based navigation.

The following sections describe the typical situations in which state-based navigation can be used. Each of these sections refers to the State-Based Navigation QuickStart, which implements an instant messaging-style application that allows users to manage and chat with their contacts.

Displaying Data in Different Formats or Styles

Your application may often need to present the same data to the user, but in different formats or styles. In this case, you can use a state-based navigation within the view to switch between the different styles, potentially using an animated transition between them. For example, the State-Based Navigation QuickStart allows users to choose how their contacts are displayed—either as a simple text list or as avatars (icons). Users can switch between these visual representations by clicking the **List** button or the **Avatars** button. The view provides an animated transition between the two representations, as shown in the following illustration.



Contact view navigation in the State-Based Navigation QuickStart

Because the view is presenting the same data, but in a different visual representation, the view model is not required to be involved in the navigation between representations. In this case, navigation is entirely handled within the view itself. This approach provides the UI designer with a lot of flexibility to design a compelling user experience without requiring changes to the application's code.

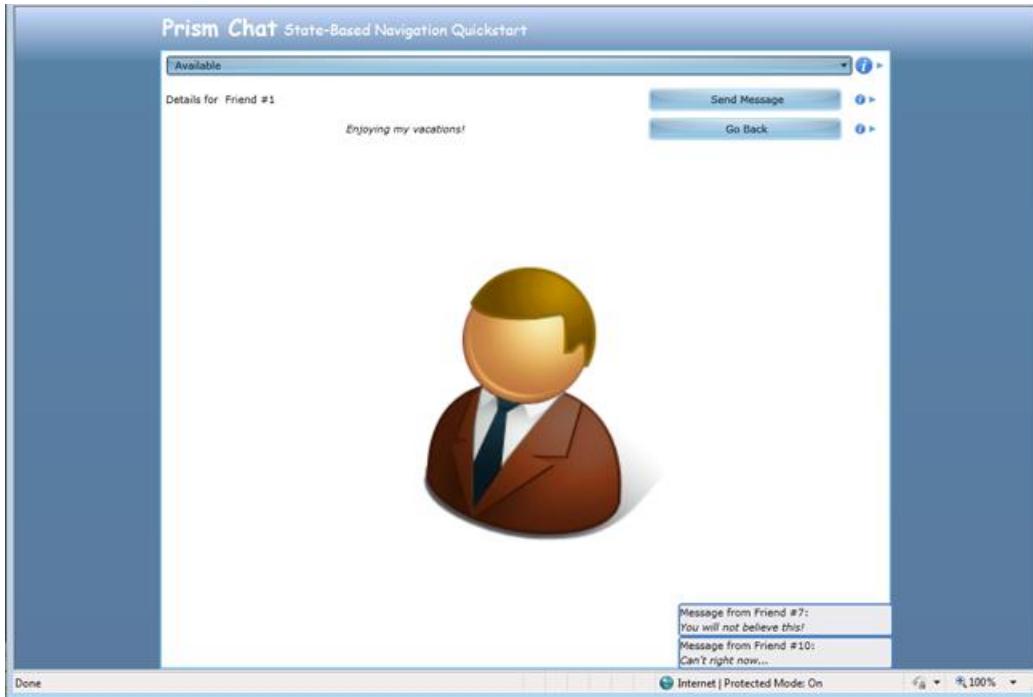
Blend behaviors provide a good way to implement this style of navigation within a view. The State-Based Navigation QuickStart application uses Blend's **DataStateBehavior** data-bound to a radio button to switch between two visual states that are defined using the visual state manager, one button to show the contacts as a list and one button to show the contacts as icons.

XAML

```
<ei:DataStateBehavior Binding="{Binding IsChecked, ElementName=ShowAsListButton}"
    Value="True"
    TrueState="ShowAsList"
    FalseState="ShowAsIcons"/>
```

As the user clicks the **Contacts** or **Avatar** radio buttons, the visual state is toggled between the **ShowAsList** visual state and the **ShowAsIcons** visual state. The flip transition animation between these states is also defined using the visual state manager.

Another example of this style of navigation is shown by the State-Based Navigation QuickStart application when the user switches to the details views for the currently selected contact. The following illustration shows an example of this.

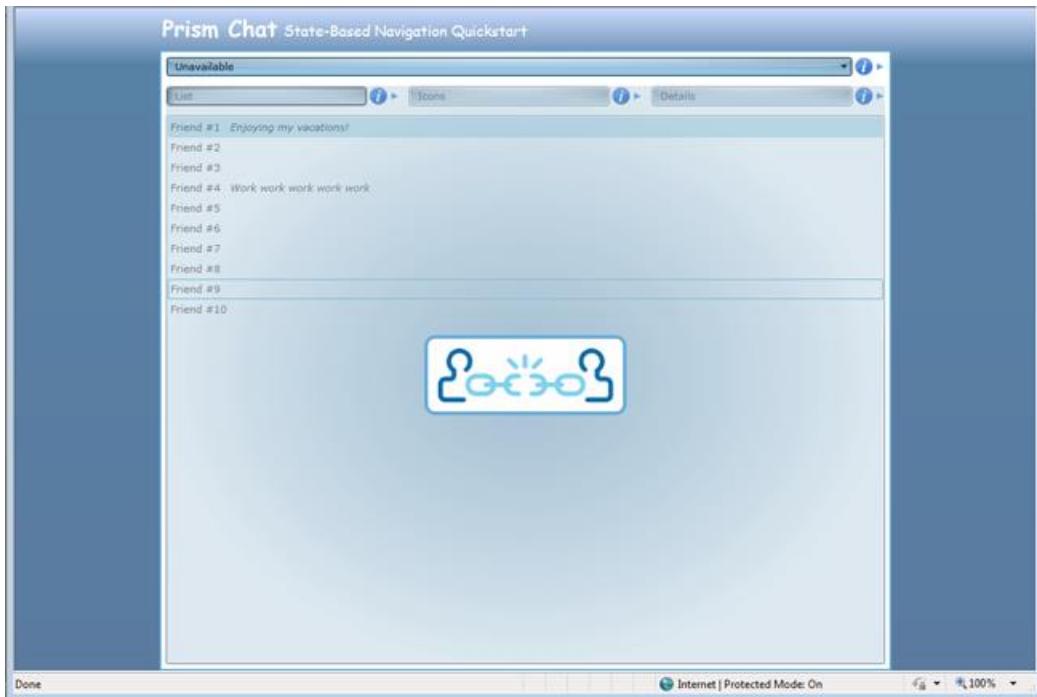


The Contact Details view in the State-Based Navigation QuickStart

Again, this can be easily implemented using the Blend **DataStateBehavior**; however, this time it is bound to the **ShowDetails** property on the view model, which toggles between the **ShowDetails** and **ShowContacts** visual states using a flip transition animation.

Reflecting Application State

Similarly, the view within an application may sometimes need to change its layout or style based on changes to an internal application state, which in turn is represented by a property on a view model. An example of this scenario is shown in the State-Based Navigation QuickStart where the user's connection status is represented on the Chat view model class using a **ConnectionStatus** property. As the user's connection status changes, the view is informed (via a property change notification event) allowing the view to visually represent the current connection state appropriately, as shown in the following illustration.



Connection state representation in the State-Based Navigation QuickStart

To implement this, the view defines a **DataStateBehavior** data bound to the view model's **ConnectionStatus** property to toggle between the appropriate visual states.

XAML

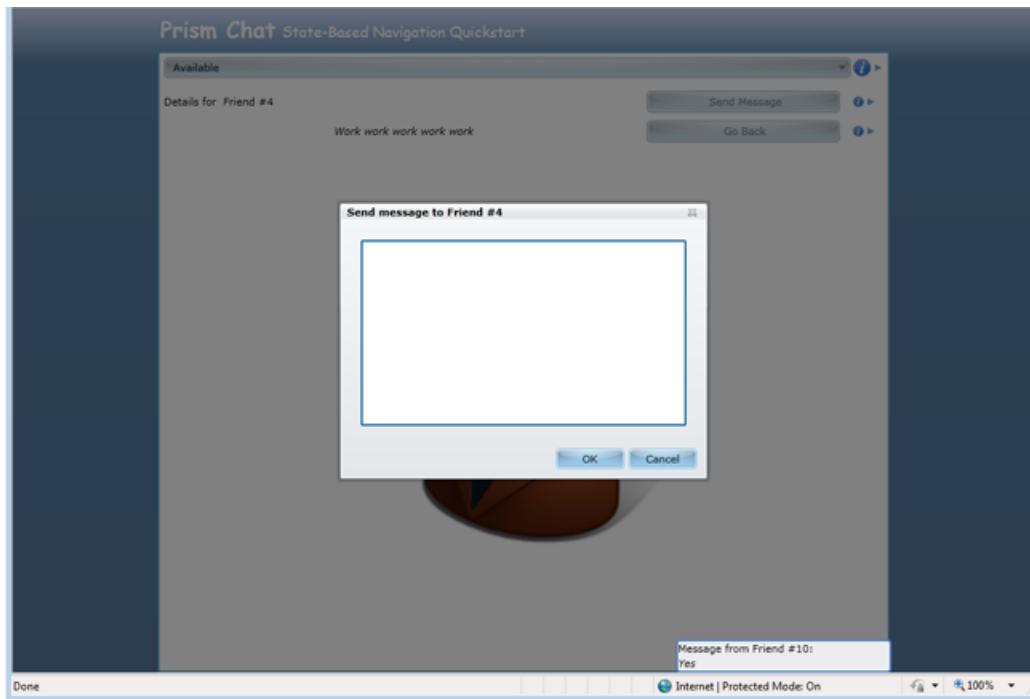
```
<ei:DataStateBehavior Binding="{Binding ConnectionStatus}"
    Value="Available"
    TrueState="Available" FalseState="Unavailable"/>
```

Note that the connection state can be changed by the user via the UI or by the application according to some internal logic or event. For example, the application may move to an "unavailable" state if the user does not interact with the view within a certain time period or when the user's calendar indicates that he or she is in a meeting. The State-Based Navigation QuickStart simulates this scenario by switching the connection status randomly using a timer. When the connection status is changed, the property on the view model is updated, and the view is informed via a property changed event. The UI is then updated to reflect the current connection status.

All the preceding examples involve defining visual states in the view and switching between them as a result of the user's interaction with the view or via changes in properties defined by the view model. This approach allows the UI designer to implement navigation-like visual behavior in the view without requiring the view to be replaced or requiring any code changes to the application's code. This approach is suitable when the view is required to render the same data in different styles or layouts. It is not suitable for situations in which the user is to be presented with different data or application functionality or when navigating to a different part of the application.

Interacting With the User

Frequently, an application will need to interact with the user in a limited way. In these situations, it is often more appropriate to interact with the user within the context of the current view, instead of navigating to a new view. For example, in the State-Based Navigation QuickStart, the user is able to send a message to a contact by clicking the **Send Message** button. The view then displays a pop-up window that allows the user to type the message, as shown in the following illustration. Because this interaction with the user is limited and logically takes place within the context of the parent view, it can be easily implemented as state-based navigation.



Interacting with the user using a pop-up window in the State-Based Navigation QuickStart

To implement this behavior, the State-Based Navigation QuickStart implements a **SendMessage** command, which is bound to the **Send Message** button. When this command is invoked, the view model interacts with the view to display the pop-up window. This is achieved using the Interaction Request pattern described in [Implementing the MVVM Pattern](#).

The following code example shows how the view in the State-Based Navigation QuickStart application responds to the **SendMessageRequest** interaction request object provided by the view model. When the request event is received, the **SendMessageChildWindow** is displayed as a popup window.

XAML

```
<prism:InteractionRequestTrigger SourceObject="{Binding SendMessageRequest}">
    <prism:PopupWindowAction IsModal="True">
        <prism:PopupWindowAction.WindowContent>
            <vs:SendMessagePopupView />
        </prism:PopupWindowAction.WindowContent>
```

```
</prism:PopupWindowAction>
</prism:InteractionRequestTrigger>
```

View-Based Navigation

Although state-based navigation can be useful for the scenarios outlined earlier, navigation within an application will most often be accomplished by replacing one view within the application's UI with another. In Prism, this style of navigation is referred to as view-based navigation.

Depending on the requirements of the application, this process can be fairly complex and require careful coordination. The following are common challenges that often have to be addressed when implementing view-based navigation:

- The target of the navigation—the container or host control of the views to be added or removed—may handle navigation differently as views are added or removed from it, or they may visually represent navigation in different ways. In many cases, the navigation target will be a simple **Frame** or **ContentControl**, and navigated views will simply be displayed within these controls. However, there are many scenarios where the target for the navigation operation is a different type of container control, such as a **TabControl** or a **ListBox** control. In these cases, navigation may require the activation or selection of an existing view or the addition of new view is a specific way.
- The application will also often have to define how the view to be navigated to is identified. For example, in a web application, the page to be navigated to is often directly identified by a Uniform Resource Identifier (URI). In a client application, the view can be identified by type name, resource location, or in a variety of different ways. Furthermore, in a composite application, which is composed from loosely coupled modules, the views will often be defined in separate modules. Individual views will need to be identified in a way that does not introduce tight coupling and dependencies between modules.
- After the view is identified, the process by which the new view is instantiated and initialized has to be carefully coordinated. This can be particularly important when using the MVVM pattern. In this case, the view and view model may need to be instantiated and associated with each other via the view's data context during the navigation operation. In the case when the application is leveraging a dependency injection container, such as the Unity Application Block (Unity) or the Managed Extensibility Framework (MEF), the instantiation of the views and/or view models (and other dependent classes) may have to be achieved using a specific construction mechanism.
- The MVVM pattern provides a separation between the application's UI and its presentation and business logic. However, the navigational behavior of the application will often span UI and presentation logic parts of the application. The user will often initiate navigation from within the view, and the view will be updated as a result of that navigation, but navigation will often also need to be initiated or coordinated from within the view model. The ability to cleanly separate the navigational behavior of the application across the view and view model is an important aspect to consider.

- An application will also often need to pass parameters or context to the view so that it can be initialized properly. For example, if the user navigates to a view to update the details of a specific customer, the customer's ID or data will have to be passed to the view so that it can display the correct information.
- Many applications will also have to carefully coordinate navigation to ensure that certain business rules are obeyed. For example, users may be prompted before navigating away from a view so that they can correct any invalid data or be prompted to submit or discard any data changes that they have made within that view. This process requires careful coordination between the previous view and the new view.
- Lastly, most modern applications allow the user to easily navigate backward (or forward) to previously displayed views. Similarly, some applications implement their workflows using a sequence of views or forms and allow users to navigate forward or backward through them, adding or updating data as they go, before completing the task and submitting all their changes at one time. These scenarios require some kind of journaling (or history) mechanism so that the sequence of navigation can be stored, replayed, or pre-defined.

Prism provides support and guidance for these challenges by extending Prism's region mechanism to support navigation. The following sections provide a brief summary of Prism regions and describe how they have been extended to support view-based navigation.

Prism Region Overview

Prism regions are designed to support the development of composite applications (that is, applications that consist of multiple modules) by allowing the application's overall UI to be constructed in a loosely-coupled way. Regions allow views defined in a module to be displayed within the application's UI without requiring the module to have explicit knowledge of the application's overall UI structure. They allow the layout of the application's UI to be changed easily, thereby allowing the UI designer to choose the most appropriate UI design and layout for the application without requiring changes in the modules themselves.

Prism regions are essentially named placeholders within which views can be displayed. Any control in the application's UI can be declared a region by simply adding a **RegionName** attached property to it, as shown here.

XAML

```
<ContentControl prism:RegionManager.RegionName="MainRegion" ... />
```

For each control specified as a region, Prism creates a **Region** object to represent the region and a **RegionAdapter** object, which manages the placement and activation of views into the specified control. The Prism Library provides **RegionAdapter** implementations for most of the common WPF controls. You can create a custom **RegionAdapter** to support additional controls or when you need to define a custom behavior. The **RegionManager** class provides access to the **Region** objects within the application.

In many cases, the region control will be a simple control, such as a **ContentControl**, that can display one view at a time. In other cases, the **Region** control will be a control that is able to display multiple views at the same time, such as a **TabControl** or a **ListBox** control.

The region adapter manages a list of views within the associated region. One or more of these views will be displayed in the region control according to its defined layout strategy. Views can be assigned a name that can be used to retrieve that view later on. The region adapter manages the active state of the views within the region. The active view is the view that is the selected or top-most view—for example, in a **TabControl**, the active view is the one displayed in the selected tab; in a **ContentControl**, the active view is the view that is currently displayed as the control's content.

Note: The active state of a view is important to consider during navigation. Frequently, you will want the active view to participate in navigation so that it can save data before the user navigates away from it, or so that it can confirm or cancel the navigation operation.

Previous versions of Prism allowed views to be displayed in a region in two ways. The first, called *view injection*, allows views to be programmatically displayed in a region. This approach is useful for dynamic content, where the view to be displayed in the region changes frequently, according to the application's presentation logic.

View injection is supported through the **Add** method on the **Region** class. The follow code example shows how you can obtain a reference to a **Region** object via the **RegionManager** class and programmatically add a view to it. In this example, the view is created using a dependency injection container.

C#

```
IRegionManager regionManager = ...;
IRegion mainRegion = regionManager.Regions["MainRegion"];
InboxView view = this.container.Resolve<InboxView>();
mainRegion.Add(view);
```

The second method, called *view discovery*, allows a module to register a view type against a region name. Whenever a region with the specified name is displayed, an instance of the specified view will be automatically created and displayed in the region. This approach is useful for relatively static content, where the view to be displayed in a region does not change.

View discovery is supported through the **RegisterViewWithRegion** method on the **RegionManager** class. This method allows you to specify a callback method that will be called when the named region is shown. The following code example shows how you can create a view (via the dependency injection container) when the main region is first shown.

C#

```
IRegionManager regionManager = ...;
regionManager.RegisterViewWithRegion("MainRegion", () =>
    container.Resolve<InboxView>());
```

For a detailed overview of Prisms region support and information about how to leverage regions to compose the application's UI using view injection and discovery, see [Composing the User Interface](#). The rest of this topic describes how regions have been extended to support view-based navigation, and how this addresses the various challenges described earlier.

Basic Region Navigation

Both view injection and view discovery can be considered to be limited forms of navigation—view injection is a form of explicit, programmatic navigation, and view discovery is a form of implicit or deferred navigation. However, in Prism 4.0, regions have been extended to support a more general notion of navigation, based on URIs and an extensible navigation mechanism.

Navigation within a region means that a new view is to be displayed within that region. The view to be displayed is identified via a URI, which, by default, refers to the name of the view to be created. You can programmatically initiate navigation using the **RequestNavigate** method defined by the **INavigateAsync** interface.

Note: Despite its name, the **INavigateAsync** interface does not represent asynchronous navigation that's carried out on a separate background thread. Instead, the **INavigateAsync** interface represents the ability to perform pseudo-asynchronous navigation. The **RequestNavigate** method may return synchronously following the completion of navigation operation, or it may return while a navigation operation is still pending, as in the case where the user needs to confirm the navigation. By allowing you to specify callbacks and continuations during navigation, Prism provides a mechanism to enable these scenarios without requiring the complexity of navigating on a background thread.

The **INavigateAsync** interface is implemented by the **Region** class, allowing you to initiate navigation within that region.

C#

```
IRegion mainRegion = ...;
mainRegion.RequestNavigate(new Uri("InboxView", UriKind.Relative));
```

You can also call the **RequestNavigate** method on the **RegionManager**, which allows you to specify the name of the region to be navigated. This convenient method obtains a reference to the specified region and then calls the **RequestNavigate** method, as shown in the preceding code example.

C#

```
IRegionManager regionManager = ...;
regionManager.RequestNavigate("MainRegion",
    new Uri("InboxView", UriKind.Relative));
```

By default, the navigation URI specifies the name of a view that is registered in the container.

Using MEF, you can simply export the view type with the specified name.

C#

```
[Export("InboxView")]
public partial class InboxView : UserControl
```

During navigation, the specified view is instantiated, via the container or MEF, along with its corresponding view model and other dependent services and components. After the view is instantiated, it is then added to the specified region and activated (activation is described in more detail later in this topic).

Note: The preceding description illustrates view-first navigation, where the URI refers to the name of the view type, as it is exported or registered with the container. With view-first navigation, the dependent view model is created as a dependency of the view. An alternative approach is to use view model-first navigation, where the navigation URI refers to the name of the view model type, as it is exported or registered with the container. View model-first navigation is useful when the view is defined as a data template, or when you want your navigation scheme to be defined independently of the views.

The **RequestNavigate** method also allows you to specify a callback method, or a delegate, which will be called when navigation is complete.

C#

```
private void SelectedEmployeeChanged(object sender, EventArgs e)
{
    ...
    regionManager.RequestNavigate(RegionNames.TabRegion,
        "EmployeeDetails", NavigationCompleted);
}
private void NavigationCompleted(NavigationResult result)
{
    ...
}
```

The **NavigationResult** class defines properties that provide information about the navigation operation. The **Result** property indicates whether or not navigation succeeded. If navigation failed, the **Error** property provides a reference to any exception that was thrown during navigation. The **Context** property provides access to the navigation URI and any parameters it contains, and a reference to the navigation service that coordinated the navigation operation.

View and View Model Participation in Navigation

Frequently, the views and view models in your application will want to participate in navigation. The **INavigationAware** interface enables this. You can implement this interface on the view or (more commonly) the view model. By implementing this interface, your view or view model can opt-in to participate in the navigation process.

Note: In the description that follows, although a reference is made to calls to this interface during navigation between views, it should be noted that the **INavigationAware** interface will be called during navigation whether it is implemented by the view or by the view model.

During navigation, Prism checks to see whether the view implements the **INavigationAware** interface; if it does, it calls the required methods during navigation. Prism also checks to see whether the object

set as the view's **DataContext** implements this interface; if it does, it calls the required methods during navigation.

This interface allows the view or view model to participate in a navigation operation. The **INavigationAware** interface defines three methods.

C#

```
public interface INavigationAware
{
    bool IsNavigationTarget(NavigationContext navigationContext);
    void OnNavigatedTo(NavigationContext navigationContext);
    void OnNavigatedFrom(NavigationContext navigationContext);
}
```

The **IsNavigationTarget** method allows an existing (displayed) view or view model to indicate whether it is able to handle the navigation request. This is useful in cases where you can re-use an existing view to handle the navigation operation or when navigating to a view that already exists. For example, a view displaying customer information can be updated to display a different customer's information. For more information about using this method, see the section, [Navigating to Existing Views](#), later in this topic.

The **OnNavigatedFrom** and **OnNavigatedTo** methods are called during a navigation operation. If the currently active view in the region implements this interface (or its view model), its **OnNavigatedFrom** method is called before navigation takes place. The **OnNavigatedFrom** method allows the previous view to save any state or to prepare for its deactivation or removal from the UI, for example, to save any changes that the user has made to a web service or database.

If the newly created view implements this interface (or its view model), its **OnNavigatedTo** method is called after navigation is complete. The **OnNavigatedTo** method allows the newly displayed view to initialize itself, potentially using any parameters passed to it on the navigation URI. For more information, see the next section, [Passing Parameters During Navigation](#).

After the new view is instantiated, initialized, and added to the target region, it then becomes the active view, and the previous view is deactivated. Sometimes you will want the deactivated view to be removed from the region. Prism provides the **IRegionMemberLifetime** interface, which allows you to control the lifetime of views within regions by allowing you to specify whether deactivated views are to be removed from the region or simply marked as deactivated.

C#

```
public class EmployeeDetailsViewModel : IRegionMemberLifetime
{
    public bool KeepAlive
    {
        get { return true; }
    }
}
```

The **IRegionMemberLifetime** interface defines a single read-only property, **KeepAlive**. If this property returns **false**, the view is removed from the region when it is deactivated. Because the region no longer has a reference to the view, it then becomes eligible for garbage collection (unless some other component in your application maintains a reference to it). You can implement this interface on your view or your view model classes. Although the **IRegionMemberLifetime** interface is primarily intended to allow you to manage the lifetime of views within regions during activation and deactivation, the **KeepAlive** property is also considered during navigation after the new view is activated in the target region.

Note: Regions that can display multiple views, such as those that use an **ItemsControl** or a **TabControl**, will display both non-active and active views. Removal of a non-active view from these types of regions will result in the view being removed from the UI.

Passing Parameters During Navigation

To implement the required navigational behavior in your application, you will often need to specify additional data during navigation request than just the target view name. The **NavigationContext** object provides access to the navigation URI, and to any parameters that were specified within it or externally. You can access the **NavigationContext** from within the **IsNavigationTarget**, **OnNavigatedFrom**, and **OnNavigatedTo** methods.

Prism provides the **NavigationParameters** class to help specify and retrieve navigation parameters. The **NavigationParameters** class maintains a list of name-value pairs, one for each parameter. You can use this class to pass parameters as part of navigation URI or for passing object parameters.

The following code example shows how to add individual string parameters to the **NavigationParameters** instance so that it can be appended to the navigation URI.

C#

```
Employee employee = Employees.CurrentItem as Employee;
if (employee != null)
{
    var navigationParameters = new NavigationParameters();
    navigationParameters.Add("ID", employee.Id);
    _regionManager.RequestNavigate(RegionNames.TabRegion,
        new Uri("EmployeeDetailsView" + navigationParameters.ToString(),
        UriKind.Relative));
}
```

Additionally, you can pass object parameters by adding them to the **NavigationParameters** instance, and passing it as a parameter of the **RequestNavigate** method. This is shown in the following code.

C#

```
Employee employee = Employees.CurrentItem as Employee;
if (employee != null)
{
    var parameters = new NavigationParameters();
    parameters.Add("ID", employee.Id);
    parameters.Add("myObjectParameter", new ObjectParameter());
    regionManager.RequestNavigate(RegionNames.TabRegion,
        new Uri("EmployeeDetailsView", UriKind.Relative), parameters);
}
```

You can retrieve the navigation parameters using the **Parameters** property on the **NavigationContext** object. This property returns an instance of the **NavigationParameters** class, which provides an indexer property to allow easy access to individual parameters, independently of them being passed through the query or through the **RequestNavigate** method.

C#

```
public void OnNavigatedTo(NavigationContext navigationContext)
{
    string id = navigationContext.Parameters["ID"];
    ObjectParameter myParameter = navigationContext.Parameters["myObjectParameter"];
}
```

Navigating to Existing Views

Frequently, it is more appropriate for the views in your application to be re-used, updated, or activated during navigation, instead of replaced by a new view. This is often the case where you are navigating to the same type of view but need to display different information or state to the user, or when the appropriate view is already available in the UI but needs to be activated (that is, selected or made top-most).

For an example of the first scenario, imagine that your application allows the user to edit customer records, using the **EditCustomer** view, and the user is currently using that view to edit customer ID 123. If the customer decides to edit the customer record for customer ID 456, the user can simply navigate to the **EditCustomer** view and enter the new customer ID. The **EditCustomer** view can then retrieve the data for the new customer and update its UI accordingly.

An example of the second scenario is where the application allows the user to edit more than one customer record at a time. In this case, the application displays multiple **EditCustomer** view instances in a tab control—for example, one for customer ID 123 and another for customer ID 456. When the user navigates to the **EditCustomer** view and enters customer ID 456, the corresponding view will be activated (that is, its corresponding tab will be selected). If the user navigates to the **EditCustomer** view and enters customer ID 789, a new instance will be created and displayed in the tab control.

The ability to navigate to an existing view is useful for a variety of reasons. It is often more efficient to update an existing view instead of replace it with a new instance of the same type. Similarly, activating an existing view, instead of creating a duplicate view, provides a more consistent user experience. In addition, the ability to handle these situations seamlessly without requiring much custom code means that the application is easier to develop and maintain.

Prism supports the two scenarios described earlier via the **INavigationTarget** method on the **INavigationAware** interface. This method is called during navigation on all views in a region that are of the same type as the target view. In the preceding examples, the target type of the view is the **EditCustomer** view, so the **INavigationTarget** method will be called on all existing **EditCustomer** view instances currently in the region. Prism determines the target type from the view URI, which it assumes is the short type name of the target type.

Note: For Prism to determine the type of the target view, the view's name in the navigation URI should be the same as the actual target type's short type name. For example, if your view is implemented by the **MyApp.Views.EmployeeDetailsView** class, the view name specified in the navigation URI should be **EmployeeDetailsView**. This is the default behavior provided by Prism. You can customize this behavior by implementing a custom content loader class; you can do this by implementing the **IRegionNavigationContentLoader** interface or by deriving from the **RegionNavigationContentLoader** class.

The implementation of the **INavigationTarget** method can use the **NavigationContext** parameter to determine whether it can handle the navigation request. The **NavigationContext** object provides access to the navigation URI and the navigation parameters. In the preceding examples, the implementation of this method in the **EditCustomer** view model compares the current customer ID to the ID specified in the navigation request, and it returns **true** if they match.

C#

```
public bool IsNavigationTarget(NavigationContext navigationContext)
{
    string id = navigationContext.Parameters["ID"];
    return _currentCustomer.Id.Equals(id);
}
```

If the **INavigationTarget** method always returns **true**, regardless of the navigation parameters, that view instance will always be re-used. This allows you to ensure that only one view of a particular type will be displayed in a particular region.

Confirming or Cancelling Navigation

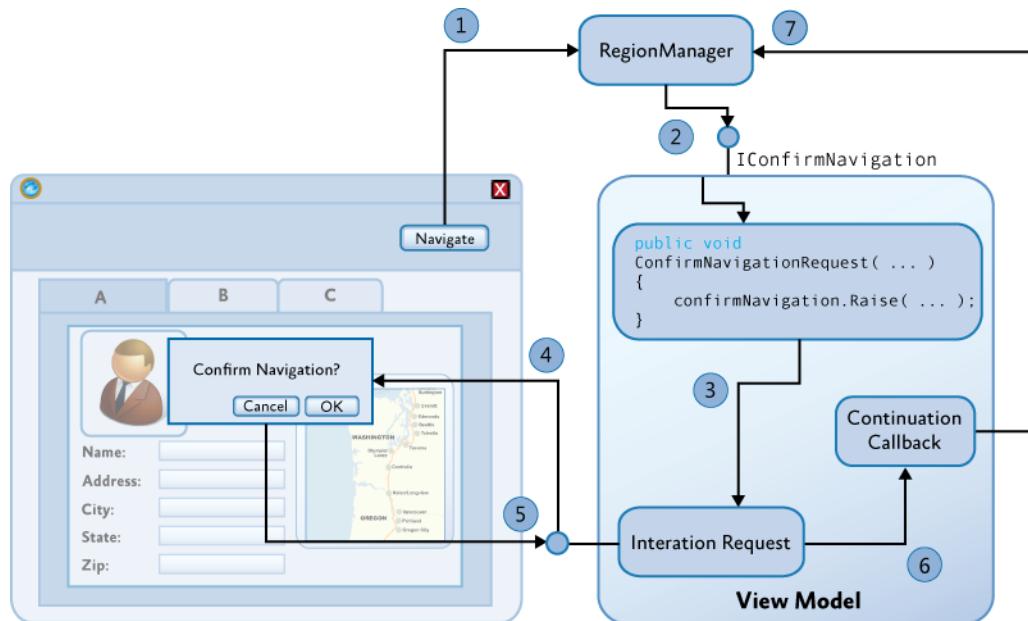
You will often find that you will need to interact with the user during a navigation operation, so that the user can confirm or cancel it. In many applications, for example, the user may try to navigate while in the middle of entering or editing data. In these situations, you may want to ask the user whether he or she wants to save or discard the data that has been entered before continuing to navigate away from

the page, or whether the user wants to cancel the navigation operation altogether. Prism supports these scenarios via the **IConfirmNavigationRequest** interface.

The **IConfirmNavigationRequest** interface derives from the **INavigationAware** interface and adds the **ConfirmNavigationRequest** method. By implementing this interface on your view or view model class, you allow them to participate in the navigation sequence in a way that allows them to interact with the user so that the user can confirm or cancel the navigation. You will often use an **Interaction Request** object, as described in [Using Interaction Request Objects](#) in [Advanced MVVM Scenarios](#), to display a confirmation pop-up window.

Note: The **ConfirmNavigationRequest** method is called on the active view or view model, similar to the **OnNavigatedFrom** method described earlier.

The **ConfirmNavigationRequest** method provides two parameters, a reference to the current navigation context as described earlier, and a callback method that you can call when you want navigation to continue. For this reason, the callback is known as a continuation callback. You can store a reference to the continuation callback so the application can call it after it finishes interacting with the user. If your application interacts with the user through an **Interaction Request** object, you can chain the call to the continuation callback to the callback from the interaction request. The following diagram illustrates the overall process.



Confirming Navigation Using an **InteractionRequest** Object

The following steps summarize the process of confirming navigation using an **InteractionRequest** object:

1. Navigation operation is initiated via a **RequestNavigate** call.
2. If the view or view model implements **IConfirmNavigation**, call **ConfirmNavigationRequest**.
3. The view model raises the interaction request event.

4. The view displays the confirmation pop-up window and awaits the user's response.
5. The interaction request callback is invoked when the user closes the pop-up window.
6. Continuation callback is invoked to continue or cancel the pending navigation operation.
7. The navigation operation is completed or canceled.

To illustrate this, look at the View-Switching Navigation Quick Start. This application provides the ability for the user to compose a new email using the **ComposeEmailView** and **ComposeEmailViewModel** classes. The view model class implements the **IConfirmNavigation** interface. If the user navigates, such as by clicking the **Calendar** button, when they are composing an email, the **ConfirmNavigationRequest** method will be called so that the view model can confirm the navigation with the user. To support this, the view model class defines an interaction request, as shown in the following code example.

C#

```
public class ComposeEmailViewModel : NotificationObject, IConfirmNavigationRequest
{
    . .
    private readonly InteractionRequest<Confirmation>
        confirmExitInteractionRequest;

    public ComposeEmailViewModel(IEmailService emailService)
    {
        . .
        this.confirmExitInteractionRequest = new
            InteractionRequest<Confirmation>();
    }

    public IIInteractionRequest ConfirmExitInteractionRequest
    {
        get { return this.confirmExitInteractionRequest; }
    }
}
```

In the **ComposeEmailView** class, an interaction request trigger is defined, and data is bound to the **ConfirmExitInteractionRequest** property on the view model. When the interaction request is made, a simple pop-up window will be displayed to the user.

XAML

```
<UserControl.Resources>
    <DataTemplate x:Key="ConfirmExitDialogTemplate">
        <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
            Text="{Binding}"/>
    </DataTemplate>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White">
    <ei:Interaction.Triggers>
```

```

<prism:InteractionRequestTrigger SourceObject="{Binding
    ConfirmExitInteractionRequest}">
    <prism:PopupWindowAction IsModal="True" CenterOverAssociatedObject="True"/>
</prism:InteractionRequestTrigger>
</ei:Interaction.Triggers>
...

```

The **ConfirmNavigationRequest** method on the **ComposeEmailViewModel** class is called if the user attempts to navigate while an email is being composed. The implementation of this method invokes the interaction request defined earlier so that the user can confirm or cancel the navigation operation.

C#

```

void IConfirmNavigationRequest.ConfirmNavigationRequest(
    NavigationContext navigationContext, Action<bool> continuationCallback)
{
    . . .
    this.confirmExitInteractionRequest.Raise(
        new Confirmation {Content = "...", Title = "..."},
        c => {continuationCallback(c.Confirmed);});
}

```

The callback for the interaction request is called when the user clicks the buttons in the confirmation pop-up window to confirm or cancel the operation. This callback simply calls the continuation callback, passing in the value of the **Confirmed** flag, and causing the navigation to continue or be canceled.

Note: It should be noted that after the interaction request event is raised, the **ConfirmNavigationRequest** method immediately returns so that the user can continue to interact with the UI of the application. When the user clicks the **OK** or **Cancel** buttons on the pop-up window, the callback method of the interaction request is made, which in turn calls the continuation callback to complete the navigation operation. All the methods are called on the UI thread. Using this technique, no background threads are required.

Using this mechanism, you can control if the navigation request is carried out immediately or is deferred, pending an interaction with the user or some other asynchronous interaction (for example, as a result of a web service request). To enable navigation to proceed, you can simply call the continuation callback method, passing **true** to indicate that it can continue. Similarly, you can pass **false** to indicate that the navigation should be canceled.

C#

```

void IConfirmNavigationRequest.ConfirmNavigationRequest(
    NavigationContext navigationContext, Action<bool> continuationCallback)
{
    continuationCallback(true);
}

```

If you want to defer navigation, you can store a reference to the continuation callback you can then call when the interaction with the user (or web service) completes. The navigation operation will be pending until you call the continuation callback.

If the user initiates another navigation operation in the meantime, the navigation request then becomes canceled. In this case, calling the continuation callback has no effect because the navigation operation to which it relates is no longer current. Similarly, if you decide not to call the continuation callback, the navigation operation will be pending until it is replaced with a new navigation operation.

Using the Navigation Journal

The **NavigationContext** class provides access to the region navigation service, which is responsible for coordinating the sequence of operations during navigation within a region. It provides access to the region in which navigation is taking place, and to the navigation journal associated with that region. The region navigation service implements the **IRegionNavigationService**, which is defined as follows.

C#

```
public interface IRegionNavigationService : INavigateAsync
{
    IRegion Region {get; set;}
    IRegionNavigationJournal Journal {get;}
    event EventHandler<RegionNavigationEventArgs> Navigating;
    event EventHandler<RegionNavigationEventArgs> Navigated;
    event EventHandler<RegionNavigationFailedEventArgs> NavigationFailed;
}
```

Because the region navigation service implements the **INavigateAsync** interface, you can initiate navigation within the parent region by calling its **RequestNavigate** method. The **Navigating** event is raised when a navigation operation is initiated. The **Navigated** event is raised when navigation within a region is completed. The **NavigationFailed** is raised if an error was encountered during navigation.

The **Journal** property provides access to the navigation journal associated with the region. The navigation journal implements the **IRegionNavigationJournal** interface, which is defined as follows.

C#

```
public interface IRegionNavigationJournal
{
    bool CanGoBack { get; }
    bool CanGoForward { get; }
    IRegionNavigationJournalEntry CurrentEntry { get; }
    INavigateAsync NavigationTarget { get; set; }
    void Clear();
    void GoBack();
    void GoForward();
    void RecordNavigation(IRegionNavigationJournalEntry entry);
}
```

You can obtain and store a reference to the region navigation service within a view during navigation via the **OnNavigatedTo** method call. By default, Prism provides a simple stack-based journal that allows you to navigate forward or backward within a region.

You can use the navigation journal to allow the user to navigate from within the view itself. In the following example, the view model implements a **GoBack** command, which uses the navigation journal within the host region. Therefore, the view can display a **Back** button that allows the user to easily navigate back to the previous view within the region. Similarly, you can implement a **GoForward** command to implement a wizard style workflow.

C#

```
public class EmployeeDetailsViewModel : INavigationAware
{
    ...
    private IRegionNavigationService navigationService;

    public void OnNavigatedTo(NavigationContext navigationContext)
    {
        navigationService = navigationContext.NavigationService;
    }

    public DelegateCommand<object> GoBackCommand { get; private set; }

    private void GoBack(object commandArg)
    {
        if (navigationService.Journal.CanGoBack)
        {
            navigationService.Journal.GoBack();
        }
    }

    private bool CanGoBack(object commandArg)
    {
        return navigationService.Journal.CanGoBack;
    }
}
```

You can implement a custom journal for a region if you need to implement a specific workflow pattern within that region.

Note: The navigation journal can only be used for region-based navigation operations that are coordinated by the region navigation service. If you use view discovery or view injection to implement navigation within a region, the navigation journal will not be updated during navigation and cannot be used to navigate forward or backward within that region.

Using the WPF Navigation Framework

Prism region navigation was designed to address a wide range of common scenarios and challenges that you may face when implementing navigation in a loosely-coupled, modular application that uses the MVVM pattern and a dependency injection container, such as Unity, or the Managed Extensibility

Framework (MEF). It also was designed to support navigation confirmation and cancelation, navigation to existing views, navigation parameters and navigation journaling.

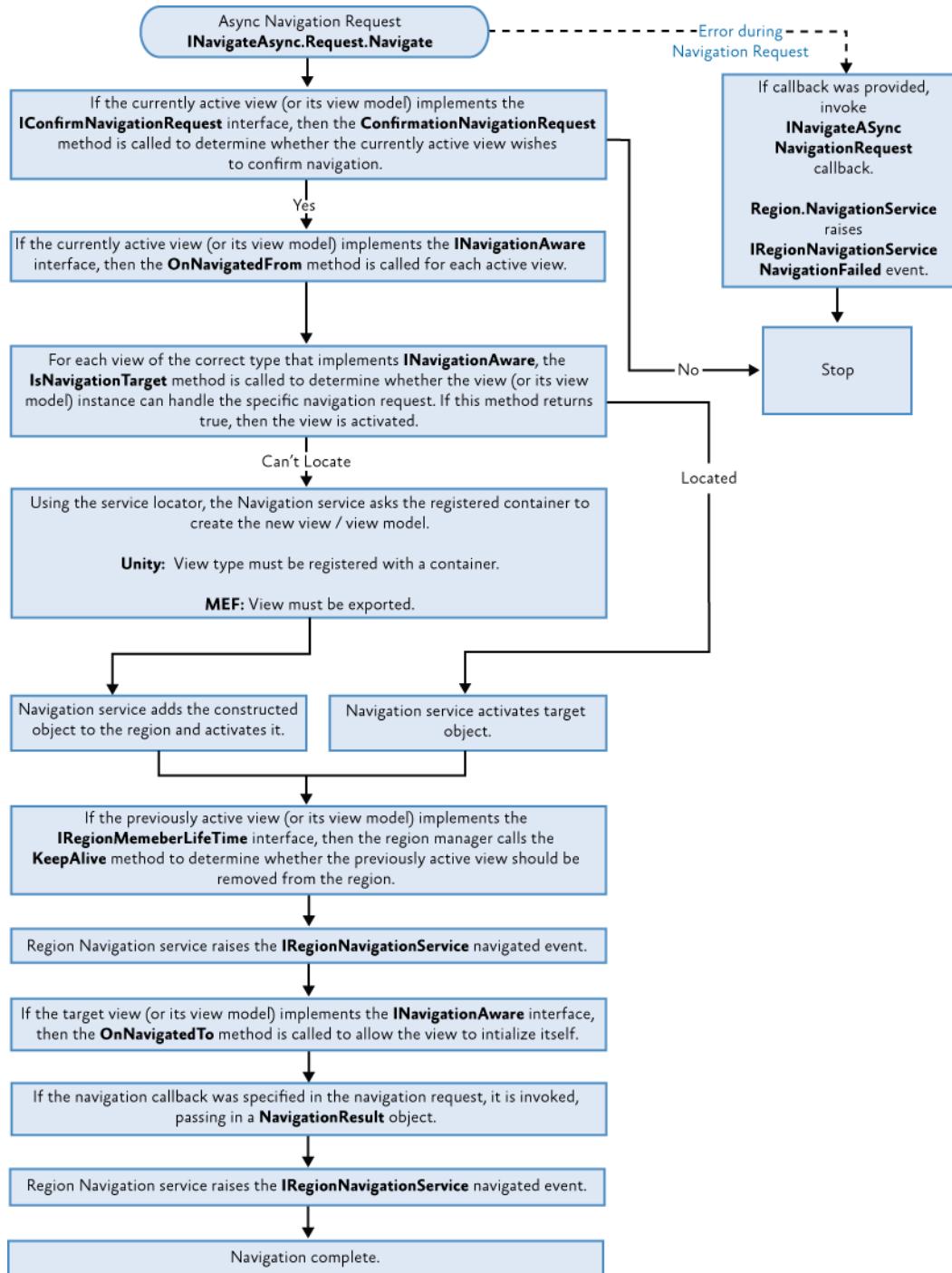
By supporting navigation within Prism regions, it also supports navigation within a wide range of layout controls and supports the ability to change the layout of the application's UI without affecting its navigation structure. It also supports pseudo-synchronous navigation, which allows for rich user interaction during navigation.

However, the Prism region navigation was not designed to replace WPF's navigation framework. Instead, Prism region navigation was designed to be used side-by-side with the WPF navigation framework.

The WPF navigation framework is difficult to use to support the MVVM pattern and dependency injection. It is also based on a **Frame** control that provides similar functionality in terms of journaling and navigation UI. You can use the WPF navigation framework alongside Prism region navigation, though it may be easier and more flexible to implement navigation using only Prism regions.

The Region Navigation Sequence

The following illustration provides an overview of the sequence of operations during a navigation operation. It is provided for reference so that you can see how the various elements of the Prism region navigation work together during a navigation request.



Prism region navigation sequence

More Information

For more information about Prism regions, see [Composing the User Interface](#).

For more information about the MVVM pattern and Interaction Request pattern, see [Implementing the MVVM Pattern](#) and [Advanced MVVM Scenarios](#).

For more information about the **Interaction Request** object, see [Using Interaction Request Objects](#) in [Advanced MVVM Scenarios](#).

For more information about the Visual State Manager, see [VisualStateManager Class](#) on MSDN.

For more information about using Microsoft Blend behaviors, see [Working with built-in behaviors](#) on MSDN.

For more information about creating custom behaviors with Microsoft Blend, see [Creating Custom Behaviors](#) on MSDN.

9: Communicating Between Loosely Coupled Components

松耦合组件的通信

When building large complex WPF applications, a common approach is to divide the functionality into discrete module assemblies. It is also desirable to minimize the use of static references between these modules, which can be accomplished through the use of delegate commands, region context, shared services, and event aggregator. This allows the modules to be independently developed, tested, deployed, and updated, and it forces loosely coupled communication. This topic provides guidance when to use delegate commands and routed commands and when to use event aggregator and .NET framework events.

When communicating between modules, it is important that you know the differences between the approaches so that you can best determine which approach to use in your particular scenario. The Prism Library provides the following communication approaches:

- **Solution commanding.** Use when there is an expectation of immediate action from the user interaction.
- **Region context.** Use this to provide contextual information between the host and views in the host's region. This approach is somewhat similar to the **DataContext**, but it does not rely on it.
- **Shared services.** Callers can call a method on the service which raises an event to the receiver of the message. Use this if none of the preceding is applicable.
- **Event aggregation.** For communication across view models, presenters, or controllers when there is not a direct action-reaction expectation.

Solution Commanding

If you need to respond to a user gesture, such as clicking on a command invoker (for example, a button or menu item), and if you want the invoker to be enabled based on business logic, use commanding.

Windows Presentation Foundation (WPF) provides **RoutedCommand**, which is good at connecting command invokers, such as menu items and buttons, with command handlers that are associated with the current item in the visual tree that has keyboard focus.

However, in a composite scenario, the command handler is often a view model that does not have any associated elements in the visual tree or is not the focused element. To support this scenario, the Prism Library provides **DelegateCommand**, which allows you to call a delegate method when the command is executed, and **CompositeCommand**, which allows you to combine multiple commands. These commands are different from the built-in **RoutedCommand**, which will route command execution and handling up and down the visual tree. This allows you to trigger a command at a point in the visual tree and handle it at a higher level.

The **CompositeCommand** is an implementation of **ICommand** so that it can be bound to invokers. **CompositeCommands** can be connected to several child commands; when the **CompositeCommand** is invoked, the child commands are also invoked.

CompositeCommands support enablement. **CompositeCommands** listen to the **CanExecuteChanged** event of each one of its connected commands. It then raises this event notifying its invoker(s). The invoker(s) reacts to this event by calling **CanExecute** on the **CompositeCommand**. The **CompositeCommand** then again polls all its child commands by calling **CanExecute** on each child command. If any call to **CanExecute** returns **false**, the **CompositeCommand** will return **false**, thus disabling the invoker(s).

How does this help you with cross module communication? Applications based on the Prism Library may have global **CompositeCommands** that are defined in the shell that have meaning across modules, such as **Save**, **Save All**, and **Cancel**. Modules can then register their local commands with these global commands and participate in their execution.

Note: **DelegateCommand** and **CompositeCommands** can be found in the Microsoft.Practices.Prism.Mvvm namespace which is located in the Prism.Mvvm NuGet package.

About WPF Routed Events and Routed Commands

A routed event is a type of event that can invoke handlers on multiple listeners in an element tree, instead of notifying only the object that directly subscribed to the event. WPF-routed commands deliver command messages through UI elements in the visual tree, but the elements outside the tree will not receive these messages because they only bubble up or down from the focused element or an explicitly stated target element. Routed events can be used to communicate through the element tree, because the event data for the event is perpetuated to each element in the route. One element could change something in the event data, and that change would be available to the next element in the route.

Therefore, you should use WPF routed events in the following scenarios: defining common handlers at a common root or defining your own custom control class.

Creating a Delegate Command

To create a delegate command, instantiate a **DelegateCommand** field in the constructor of your view model, and then expose it as an **ICommand** property.

C#

```
// ArticleViewModel.cs
public class ArticleViewModel : BindableBase
{
    private readonly ICommand showArticleListCommand;

    public ArticleViewModel(INewsFeedService newsFeedService,
                           IRegionManager regionManager,
                           IEventAggregator eventAggregator)
    {
```

```

        this.showArticleListCommand = new DelegateCommand(this.ShowArticleList);

    }

    public ICommand ShowArticleListCommand
    {
        get { return this.showArticleListCommand; }
    }
}

```

Creating a Composite Command

To create a composite command, instantiate a **CompositeCommand** field in the constructor, add commands to it, and then expose it as an **ICommand** property.

C#

```

public class MyViewModel : BindableBase
{
    private readonly CompositeCommand saveAllCommand;

    public ArticleViewModel(INewsFeedService newsFeedService,
                           IRegionManager regionManager,
                           IEventAggregator eventAggregator)
    {
        this.saveAllCommand = new CompositeCommand();
        this.saveAllCommand.RegisterCommand(new SaveProductsCommand());
        this.saveAllCommand.RegisterCommand(new SaveOrdersCommand());
    }

    public ICommand SaveAllCommand
    {
        get { return this.saveAllCommand; }
    }
}

```

Making a Command Globally Available

Typically, to create a globally available command, create an instance of the **DelegateCommand** or the **CompositeCommand** and expose it through a static class.

C#

```

public static class GlobalCommands
{
    public static CompositeCommand MyCompositeCommand = new CompositeCommand();
}

```

In your module, associate child commands to the globally available command.

C#

```

GlobalCommands.MyCompositeCommand.RegisterCommand(command1);
GlobalCommands.MyCompositeCommand.RegisterCommand(command2);

```

Note: To increase the testability of your code, you can use a proxy class to access the globally available commands and mock that proxy class in your tests.

Binding to a Globally Available Command

The following code example shows how to bind a button to the command in WPF.

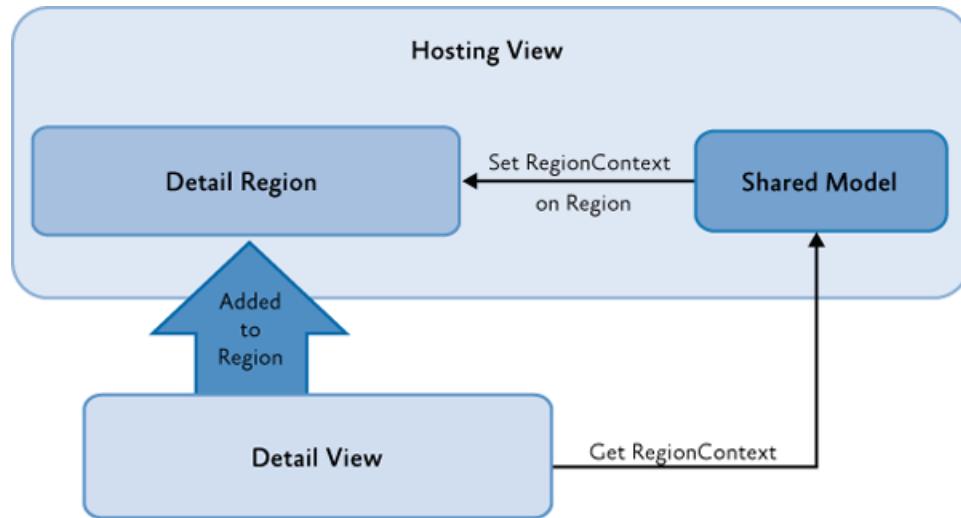
XAML

```
<Button Name="MyCompositeCommandButton" Command="{x:Static local:GlobalCommands.MyCompositeCommand}">Execute My Composite Command </Button>
```

Note: Another approach is to store the command as a resource inside the App.xaml file in the **Application.Resources** section. Then, in the view—which must be created after setting that resource—you can set **Command="{Binding MyCompositeCommand, Source={StaticResource GlobalCommands}}"** to add an invoker to the command.

Region Context

There are a lot of scenarios where you might want to share contextual information between the view that is hosting a region and a view that is inside a region. For example, a master detail–like view shows a business entity and exposes a region to show additional detail information for that business entity. The Prism Library uses a concept named **RegionContext** to share an object between the host of the region and any views that are loaded inside the region, as shown in the following illustration.



Using RegionContext

Depending on the scenario, you can choose to share a single piece of information (such as an identifier) or a shared model. The view can retrieve the **RegionContext**, and then sign up for change notifications. The view can also change the **RegionContext**'s value. There are several ways of exposing and consuming the **RegionContext**:

- You can expose **RegionContext** to a region in Extensible Application Markup Language (XAML).
- You can expose **RegionContext** to a region in code.

- You can consume **RegionContext** from a view inside a region.

Note: The Prism Library currently only supports consuming the **RegionContext** from a view inside a region if that view is a **DependencyObject**. If your view is not a **DependencyObject** (for example, you are using WPF automatic data templates and adding your view model directly in the region), consider creating a custom **RegionBehavior** to forward the **RegionContext** to your view objects.

About the Data Context Property

Data context is a concept that allows elements to inherit information from their parent elements about the data source that is used for binding. Child elements automatically inherit the **DataContext** of their parent element. The data flows down the visual tree.

Shared Services

Another method of cross-module communication is through shared services. When the modules are loaded, modules add their services to the service locator. Typically, services are registered and retrieved from a service locator by common interface types. This allows modules to use services provided by other modules without requiring a static reference to the module. Service instances are shared across modules, so you can share data and pass messages between modules.

In the Stock Trader Reference Implementation (Stock Trader RI), the Market module provides an implementation of **IMarketFeedService**. The Position module consumes these services by using the shell application's dependency injection container, which provides service location and resolution. The **IMarketFeedService** is meant to be consumed by other modules, so it can be found in the **StockTraderRI.Infrastructure** common assembly, but the concrete implementation of this interface does not need to be shared, so it is defined directly in the Market module and can be updated independently of other modules.

To see how these services are exported into MEF, see the **MarketFeedService.cs** and **MarketHistoryService.cs** files, as shown in the following code example. The Position module's **ObservablePosition** receives the **IMarketFeedService** service through constructor dependency injection.

C#

```
// MarketFeedService.cs
[Export(typeof(IMarketFeedService))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class MarketFeedService : IMarketFeedService, IDisposable
{
    ...
}
```

This helps with cross-module communication because service consumers do not need a static reference to modules providing the service. This service can be used to send or receive data between modules.

Note: Some dependency injection containers allow the registration of dependencies using attributes, as shown in this example. Other containers may use explicit registration. In these cases, the registration typically occurs during module loading when Prism invokes the **IModule.Initialize** method. See [Modular Application Development](#) for more information.

Event Aggregation

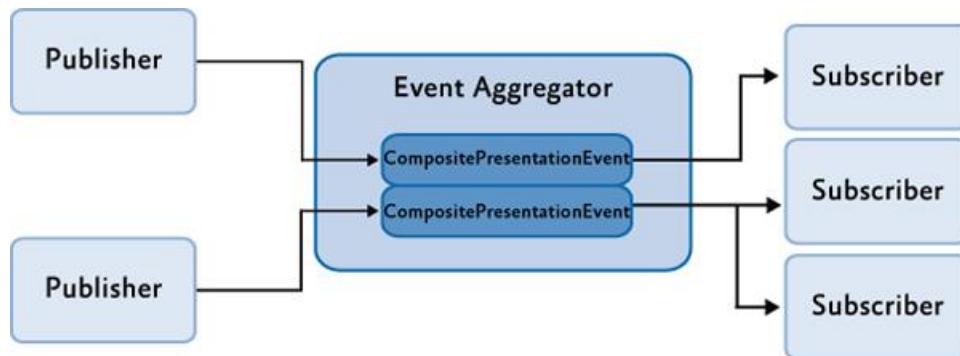
事件机制确保松耦合组件间的通信

The Prism Library provides an event mechanism that enables communications between loosely coupled components in the application. This mechanism, based on the event aggregator service, allows publishers and subscribers to communicate through events and still do not have a direct reference to each other.

The **EventAggregator** provides multicast publish/subscribe functionality. This means there can be multiple publishers that raise the same event and there can be multiple subscribers listening to the same event. Consider using the **EventAggregator** to publish an event across modules and when sending a message between business logic code, such as controllers and presenters.

One example of this, from the Stock Trader RI, is when the **Process Order** button is clicked and the order successfully processes; in this case, other modules need to know the order is successfully processed so they can update their views.

Events created with the Prism Library are typed events. This means you can take advantage of compile-time type checking to detect errors before you run the application. In the Prism Library, the **EventAggregator** allows subscribers or publishers to locate a specific **EventBase**. The event aggregator also allows for multiple publishers and multiple subscribers, as shown in the following illustration.



Event aggregator

About .NET Framework Events

Using .NET Framework events is the most simple and straightforward approach for communication between components if loose coupling is not a requirement. Events in the .NET Framework implement the Publish-Subscribe pattern, but to subscribe to an object, you need a direct reference to that object, which, in composite applications, typically resides in another module. This results in a tightly coupled design. Therefore, .NET Framework events are used for communication within modules instead of between modules.

If you use .NET Framework events, you have to be very careful of memory leaks, especially if you have a non-static or short-lived component that subscribes to an event on a static or longer-lived one. If you do not unsubscribe the subscriber, it will be kept alive by the publisher, and this will prevent the first one from being garbage-collected.

IEventAggregator

The **EventAggregator** class is offered as a service in the container and can be retrieved through the **IEventAggregator** interface. The event aggregator is responsible for locating or building events and for keeping a collection of the events in the system.

C#

```
public interface IEventAggregator
{
    TEventType GetEvent<TEventType>() where TEventType : EventBase;
}
```

The **EventAggregator** constructs the event on its first access if it has not already been constructed. This relieves the publisher or subscriber from needing to determine whether the event is available.

PubSubEvent

The real work of connecting publishers and subscribers is done by the **PubSubEvent** class. This is the only implementation of the **EventBase** class that is included in the Prism Library. This class maintains the list of subscribers and handles event dispatching to the subscribers.

The **PubSubEvent** class is a generic class that requires the payload type to be defined as the generic type. This helps enforce, at compile time, that publishers and subscribers provide the correct methods for successful event connection. The following code shows a partial definition of the **PubSubEvent** class.

Note: **PubSubEvent** can be found in the Microsoft.Practices.SubSubEvents namespace which is located in the Prism.PubSubEvents NuGet package.

C#

```
// PubSubEvent.cs
public class PubSubEvent<TPayload> : EventBase
{
    ...
    public SubscriptionToken Subscribe(Action<TPayload> action);
    public SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption
threadOption);
    public SubscriptionToken Subscribe(Action<TPayload> action, bool
keepSubscriberReferenceAlive)
    public SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption
threadOption, bool keepSubscriberReferenceAlive)

    public virtual SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption
threadOption, bool keepSubscriberReferenceAlive);
```

```

    public virtual SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption
threadOption, bool keepSubscriberReferenceAlive, Predicate<TPayload> filter);
    public virtual void Publish(TPayload payload);
    public virtual void Unsubscribe(Action<TPayload> subscriber);
    public virtual bool Contains(Action<TPayload> subscriber)
    ...
}

```

Creating and Publishing Events

The following sections describe how to create, publish, and subscribe to **PubSubEvent** using the **IEventAggregator** interface.

Creating an Event

The **PubSubEvent<TPayload>** is intended to be the base class for an application's or module's specific events. **TPayLoad** is the type of the event's payload. The payload is the argument that will be passed to subscribers when the event is published.

For example, the following code shows the **TickerSymbolSelectedEvent** in the Stock Trader Reference Implementation (Stock Trader RI). The payload is a string containing the company symbol. Notice how the implementation for this class is empty.

C#

```
public class TickerSymbolSelectedEvent : PubSubEvent<string>{}
```

Note: In a composite application, the events are frequently shared between multiple modules, so they are defined in a common place. In the Stock Trader RI, this is done in the **StockTraderRI.Infrastructure** project.

Publishing an Event

Publishers raise an event by retrieving the event from the **EventAggregator** and calling the **Publish** method. To access the **EventAggregator**, you can use dependency injection by adding a parameter of type **IEventAggregator** to the class constructor.

The following code demonstrates publishing the **TickerSymbolSelectedEvent**.

C#

```
this.eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Publish("STOCK0");
```

Subscribing to Events

Subscribers can enlist with an event using one of the **Subscribe** method overloads available on the **PubSubEvent** class. There are several ways to subscribe to **PubSubEvents**. Use the following criteria to help determine which option best suits your needs:

- If you need to be able to update UI elements when an event is received, subscribe to receive the event on the UI thread.
- If you need to filter an event, provide a filter delegate when subscribing.

- If you have performance concerns with events, consider using strongly referenced delegates when subscribing and then manually unsubscribe from the **PubSubEvent**.
- If none of the preceding is applicable, use a default subscription.

The following sections describe these options.

Subscribing on the UI Thread

Frequently, subscribers will need to update UI elements in response to events. In WPF, only a UI thread can update UI elements.

By default, the subscriber receives the event on the publisher's thread. If the publisher sends the event from the UI thread, the subscriber can update the UI. However, if the publisher's thread is a background thread, the subscriber may be unable to directly update UI elements. In this case, the subscriber would need to schedule the updates on the UI thread using the **Dispatcher** class.

The **PubSubEvent** provided with the Prism Library can assist by allowing the subscriber to automatically receive the event on the UI thread. The subscriber indicates this during subscription, as shown in the following code example.

C#

```
public void Run()
{
    ...
    this.eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews,
ThreadOption.UIThread);
}

public void ShowNews(string companySymbol)
{
    this.articlePresentationModel.SetTickerSymbol(companySymbol);
}
```

The following options are available for **ThreadOption**:

- **PublisherThread**. Use this setting to receive the event on the publishers' thread. This is the default setting.
- **BackgroundThread**. Use this setting to asynchronously receive the event on a .NET Framework thread-pool thread.
- **UIThread**. Use this setting to receive the event on the UI thread.

In order for **PubSubEvents** to publish to subscribers on the UI thread, the **EventAggregator** must initially be constructed on the UI thread.

Subscription Filtering

Subscribers may not need to handle every instance of a published event. In these cases, the subscriber can use the **filter** parameter. The **filter** parameter is of type **System.Predicate<TPayLoad>** and is a delegate that gets executed when the event is published to determine if the payload of the published event matches a set of criteria required to have the subscriber callback invoked. If the payload does not meet the specified criteria, the subscriber callback is not executed.

Frequently, this filter is supplied as a lambda expression, as shown in the following code example.

C#

```
FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>();

fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread, false,
fundOrder => fundOrder.CustomerId == this.customerId);
```

Note: The **Subscribe** method returns a subscription token of type

Microsoft.Practices.Prism.Events.SubscriptionToken that can be used to remove a subscription to the event later. This token is particularly useful when you are using anonymous delegates or lambda expressions as the callback delegate or when you are subscribing the same event handler with different filters.

Note: It is not recommended to modify the payload object from within a callback delegate because several threads could be accessing the payload object simultaneously. You could have the payload be immutable to avoid concurrency errors.

Subscribing Using Strong References

If you are raising multiple events in a short period of time and have noticed performance concerns with them, you may need to subscribe with strong delegate references. If you do that, you will then need to manually unsubscribe from the event when disposing the subscriber.

By default, **PubSubEvent** maintains a weak delegate reference to the subscriber's handler and filter on subscription. This means the reference that **PubSubEvent** holds on to will not prevent garbage collection of the subscriber. Using a weak delegate reference relieves the subscriber from the need to unsubscribe and allows for proper garbage collection.

However, maintaining this weak delegate reference is slower than a corresponding strong reference. For most applications, this performance will not be noticeable, but if your application publishes a large number of events in a short period of time, you may need to use strong references with **PubSubEvent**. If you do use strong delegate references, your subscriber should unsubscribe to enable proper garbage collection of your subscribing object when it is no longer used.

To subscribe with a strong reference, use the **keepSubscriberReferenceAlive** parameter on the **Subscribe** method, as shown in the following code example.

C#

```
FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>();
```

```
bool keepSubscriberReferenceAlive = true;

fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread,
keepSubscriberReferenceAlive, fundOrder => fundOrder.CustomerId == _customerId);
```

The **keepSubscriberReferenceAlive** parameter is of type **bool**:

- When set to **true**, the event instance keeps a strong reference to the subscriber instance, thereby not allowing it to get garbage collected. For information about how to unsubscribe, see the section [Unsubscribing from an Event](#) later in this topic.
- When set to **false** (the default value when this parameter omitted), the event maintains a weak reference to the subscriber instance, thereby allowing the garbage collector to dispose the subscriber instance when there are no other references to it. When the subscriber instance gets collected, the event is automatically unsubscribed.

Default Subscriptions

For a minimal or default subscription, the subscriber must provide a callback method with the appropriate signature that receives the event notification. For example, the handler for the **TickerSymbolSelectedEvent** requires the method to take a string parameter, as shown in the following code example.

C#

```
public TrendLineViewModel(IMarketHistoryService marketHistoryService,
IEventAggregator eventAggregator)
{
    ...
eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(this.TickerSymbolChanged);
}

public void TickerSymbolChanged(string newTickerSymbol)
{
    MarketHistoryCollection newHistoryCollection =
this.marketHistoryService.GetPriceHistory(newTickerSymbol);

    this.TickerSymbol = newTickerSymbol;
    this.HistoryCollection = newHistoryCollection;
}
```

Unsubscribing from an Event

If your subscriber no longer wants to receive events, you can unsubscribe by using your subscriber's handler or you can unsubscribe by using a subscription token.

The following code example shows how to directly unsubscribe to the handler.

C#

```
FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>();  
  
fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.PublisherThread);  
  
fundAddedEvent.Unsubscribe(FundAddedEventHandler);
```

The following code example shows how to unsubscribe with a subscription token. The token is supplied as a return value from the **Subscribe** method.

C#

```
FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>();  
  
subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler,  
ThreadOption.UIThread, false, fundOrder => fundOrder.CustomerId == this.customerId);  
  
fundAddedEvent.Unsubscribe(subscriptionToken);
```

More Information

For more information about weak references, see [Weak References](#) on MSDN.

10: Deploying Applications

To successfully move a Prism application into production, you need to plan for deployment as part of the design process of your application. This topic covers the considerations and actions you need to perform to prepare your composite or modular application for deployment and the actions you need to take to get the application in the user's hands.

Deploying WPF Prism Applications

A WPF Prism application can be composed of an executable and any number of additional DLLs. The main executable is the shell application project. Some of the additional DLLs will be the modules of the application. There may be some additional DLLs that are just shared assemblies used by the shell and modules of the application. In addition, you might have a set of resource or content files that get deployed along with the application.

To deploy a WPF Prism application, you have three choices:

- "XCopy deployment"
- ClickOnce deployment
- Windows Installer deployment

"XCopy deployment" is used as a general term for manual deployment through some sort of file copy operation, which may or may not include the use of the XCOPY command-line tool. If you choose to deploy the application in this way, it is up to you to manually package the files and move them to the target computer. The application should be ready to run as long as the expected folder structure and relative locations of the shell application executable, the module DLLs, and the content files are maintained.

Usually, a more automatic means of deployment is desired to ensure that things get placed in the right location and the user has easy access to run the application. To facilitate that, you can choose to use ClickOnce or Windows Installer (.msi files), depending on what additional installation requirements exist for the application.

The decision of whether to use ClickOnce or Windows Installer is often misunderstood. ClickOnce is not intended to be a one-size-fits-all deployment technology. It is intended for applications that need a low-impact install on a client computer. If your application needs to make computer-wide changes when it is installed—such as to install drivers, integrate with other applications, install services and other things that go outside the scope of just running your executable, ClickOnce is probably not an appropriate deployment choice. However, if you have a lightweight installation on the client computer and you want to benefit from network deployment and update of your WPF application, ClickOnce can be a great choice.

To create a Windows Installer deployment package (.msi file) for your application, you have a variety of choices, including Visual Studio Setup projects, Windows Installer XML (WiX) projects, or numerous third-party installer creation products.

Deploying WPF Prism Application with ClickOnce

ClickOnce is a Windows Presentation Foundation (WPF) or Windows Forms deployment mechanism that has been part of the .NET Framework since version 2.0. ClickOnce enables automatic deployment and update of WPF applications over the network from a deployment server. WPF Prism applications can use ClickOnce to get the shell, modules, and any other dependencies deployed to the client computer. The main challenge with Prism applications is that the Visual Studio publishing process for ClickOnce does not automatically include dynamically loaded modules in the published application.

Deploying a WPF application with ClickOnce is a two-step process. First, you have to publish the application from Visual Studio, and then you can deploy it to client computers. Publishing the application generates two manifests (a deployment manifest and an application manifest), and it copies the application files to a publish directory. That publish folder can then be moved to another server that may not be directly accessible from the developer computer to make the published application accessible to client computers from a known location and URL. Deploying an application to a client computer simply requires providing a URL or link that the user can navigate to. The URL points to the deployment manifest on the publishing deployment server. When that URL is loaded in the browser, ClickOnce on the client computer downloads the manifests and the application files specified by the manifests. After the files are downloaded and stored under the user profile, ClickOnce then launches the application. If subsequent updates are published to the deployment server, ClickOnce can automatically detect those updates, download, and apply them, or there are settings that allow you to detect and apply updates on demand or in the background after the application has launched.

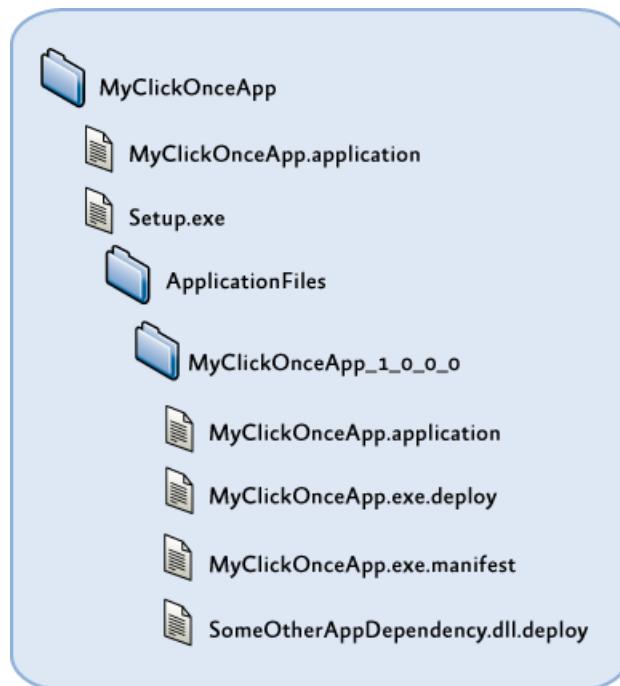
When you publish a WPF Prism application that has dynamically loaded modules, the shell project will typically not have project references to the dynamically loaded modules. As a result, the published ClickOnce application manifest also does not include those module files, and if you deploy the application using ClickOnce, the client computer will not get the module files. To address this, you must modify the application manifest to include the module files that are not referenced by the shell application project.

ClickOnce Publishing Process

You can publish ClickOnce applications from Visual Studio 2013 using the Windows Software Development Kit (SDK) tool named the Manifest Generating and Editing tool (Mage) or a custom tool that uses the ClickOnce publishing APIs. Visual Studio exposes most of the capabilities needed for ClickOnce publishing. However, Visual Studio may not be available or desired for IT administrators who manage ClickOnce deployments on the server. Mage is designed to address most common administrative tasks for ClickOnce; it is a lightweight .NET Framework Windows-based application that can be given to your administrators. However, Mage requires too many detailed steps, performed in the correct order, to successfully complete common tasks such as modifying the application files listed in the application manifest. To make these tasks simpler, a custom utility is needed.

The Manifest Manager Utility sample utility demonstrates how to use the ClickOnce publishing API to manage deployment and application manifests in a simpler way. This utility is used for updating application manifest file lists and deployment manifest settings in a single user interface (UI) and its use is described in later sections in this topic for initial deployment and update of a Prism application. The Manifest Manager Utility uses APIs exposed in the **Microsoft.Build.Tasks.Deployment** namespace to load, manipulate, and save modified manifest files for a ClickOnce deployment. You can download the [Manifest Manager Utility](#) from the Prism community site on Codeplex. To learn the specific steps involved in publishing and updating a WPF Prism application that uses dynamic module loading, see the [Publishing and Updating Applications Using the Prism Library Hands-on Lab](#).

The following illustration shows the typical structure for a ClickOnce application publication, based on the way Visual Studio generates the deployment folders when you publish an application with ClickOnce. It includes a root folder for the application, which contains the default deployment manifest (.application file). The default deployment manifest usually points to the most recently published version when generated by Visual Studio, but it can be changed to point to whichever version the administrator chooses. The root folder also contains the Setup.exe bootstrapper, which allows you to deploy prerequisites for your application that might require an installer or executable to run before deploying the application using ClickOnce. There is then a subfolder for the application-specific files, under which you get a separate subfolder for each version that you publish. The publish version is a separate project setting and entry in the deployment manifest file for versioning the deployment as a whole, as opposed to the individual assembly versions of the contained assemblies. The publish version is used by ClickOnce to determine when there is an update available from a client that has already installed a ClickOnce application.



ClickOnce publish folder structure

Under each publish version's application files folder, you have another copy of the deployment manifest (.application file) that can be used to deploy specific versions to a client computer, or it can be copied to the root folder to cause a server-side rollback to a previous version. The application executable, in addition to any dependent libraries (such as Prism module assemblies) and resource files, will also be in this folder and will be automatically suffixed by a .deploy file name extension when published by Visual Studio. This is done to simplify the file extension mappings on the publishing web server so that you don't have to allow downloads of .dll, .exe, and a myriad of other potential file types that the application is composed of.

The application manifest (.exe.manifest) file is also contained in this folder and is referenced by the deployment manifest. It contains the list of files the application is composed of with hash values per file to assist in change detection; it also contains a list of permissions required by the application to run because ClickOnce can launch applications in a partial trust AppDomain if desired.

If you manually generate or update a ClickOnce application publication using either Mage or a custom tool, you are not constrained to this folder and file structure. For any particular ClickOnce publication, the chain of dependencies includes the following:

- It includes a deployment manifest that points to the application manifest through an embedded code base URL.
- It includes an application manifest that contains relative paths to each of the application files. These files must reside in the same folder or a subfolder from where the application manifest resides.

It includes the application files themselves, usually with a .deploy file name extension appended to the file name to simplify mapping these files to MIME types on the deployment server. ClickOnce automatically strips off the .deploy file name extension on the client side after the file is downloaded.

ClickOnce Deployment and Update Process

The actual deployment of the application to a user via ClickOnce is almost always initiated by providing a URL or hyperlink to the deployment manifest of your published application on the deployment server. The user can click the hyperlink or enter the address in a browser, and the ClickOnce deployment process is invoked. After the manifest and application files are downloaded to the client computer, the application is launched. There are ClickOnce options that allow you to install the application during the initial deployment for offline use, or you can require the user to launch the application using the link or URL every time. When you publish a new version of the application to the deployment server, ClickOnce can automatically or manually check for updates and will download and apply the update for the next time the application launches.

More Information

You can download the [Manifest Manager Utility](#) from the Prism community site on Codeplex.

To learn the specific steps involved in publishing and updating a WPF Prism application that uses dynamic module loading, see the [Publishing and Updating Applications Using the Prism Library Hands-on Lab](#).

11: Glossary

This glossary includes definitions of important terms that appear in the Prism documentation.

bootstrapper. The class responsible for the initialization of an application built using the Prism Library.

command. A loosely coupled way for you to handle user interface (UI) actions. Commands bind a UI gesture to the logic that performs the action.

composite application. A composite application is composed of a number of discrete and independent modules. These components are integrated together in a host environment to form a single, seamless application.

composite command. A command that has multiple child commands.

container. Provides a layer of abstraction for the creation of objects. Dependency injection containers can reduce the dependency coupling between objects by providing the facility to instantiate instances of classes and manage their lifetime based on the configuration of the container.

DelegateCommand. Allows delegating the commanding handling logic to selected methods instead of requiring a handler in the code-behind. It uses .NET Framework delegates as the method of invoking a target handling method.

EventAggregator. A service that is primarily a container for events that allows publishers and subscribers to be decoupled so they can evolve independently. This decoupling is useful in modularized applications because new modules can be added that respond to events defined by the shell or other modules.

modularity. The ability to create complex applications from discrete functional units named *modules*. When you develop in a modularized fashion, you structure the application into separate modules that can be individually developed, tested, and deployed by different teams. It also helps you address separation of concerns by keeping a clean separation between the UI and business functionality.

model. Encapsulates the application's business logic and data.

Model-View-ViewModel (MVVM). The MVVM pattern helps to cleanly separate the business and presentation logic of your application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make the application much easier to test, maintain, and evolve.

module. A logical unit of separation in the application.

ModuleCatalog. Defines the modules that the end user needs to run the application. The module catalog knows where the modules are located and the module's dependencies.

ModuleManager. The main class that manages the process of validating the module catalog, retrieving modules if they are remote, loading the modules into the application domain, and invoking the module's **Initialize** method.

module management phases. The phases that lead to a module being initialized. These phases are module discovery, module loading, and module initialization.

navigation. The process by which the application coordinates changes to its UI as a result of the user's interaction with the application, or as a result of internal application state changes.

ViewModel-first composition. The composition approach where the view model is logically created first, followed by the view.

Notifications. Provide change notifications to any data-bound controls in the view when the underlying property value changes. This is required to implement the MVVM pattern and is implemented using the BindableBase class.

on-demand module. A module that is retrieved and initialized only when it is explicitly requested by the application.

region. A named location that you can use to define where a view will appear. Modules can locate and add content to a region in the layout without exact knowledge of how and where the region is visually displayed. This allows the appearance and layout to change without affecting the modules that add the content to the layout.

RegionContext. A technique that can be used to share context between a parent view and child views that are hosted in a region. The **RegionContext** can be set through code or by using data binding XAML.

RegionManager. The class responsible for maintaining a collection of regions and creating new regions for controls. The **RegionManager** finds an adapter mapped to a WPF control and associates a new region to that control. The **RegionManager** also supplies the attached property that can be used for simple region creation from XAML.

Separated Presentation pattern. Pattern used to implement views, which separates presentation and business logic from the UI. Using a separated presentation allows presentation and business logic to be tested independently of the UI, makes it easier to maintain code, and increases re-use opportunities.

shell. The main window of a WPF application where the primary UI content is contained.

scoped region. Regions that belong to a particular region scope. The region scope is delimited by a parent view and includes all the child views of the parent view.

service. A service provides functionality to other modules in a loosely coupled way through an interface and is often a singleton.

state-based navigation. Navigation accomplished via state changes to existing controls in the visual tree.

UI composition. The act of building an interface by composing it from discrete views at run time, likely from separate modules.

view. The main unit of UI construction within a composite UI application. The view encapsulates the UI and UI logic that you would like to keep as decoupled as possible from other parts of the application. You can define a view as a user control, data template, or even a custom control.

view-based navigation. Navigation accomplished via the addition or removal of elements from the visual tree.

view-first composition. The composition approach where the view is logically created first, followed by the view model or presenter on which it depends.

view discovery. A way to add, show, or remove views in a region by associating the type of a view with a region name. Whenever a region with that name displays, the registered views will be automatically created and added to the region.

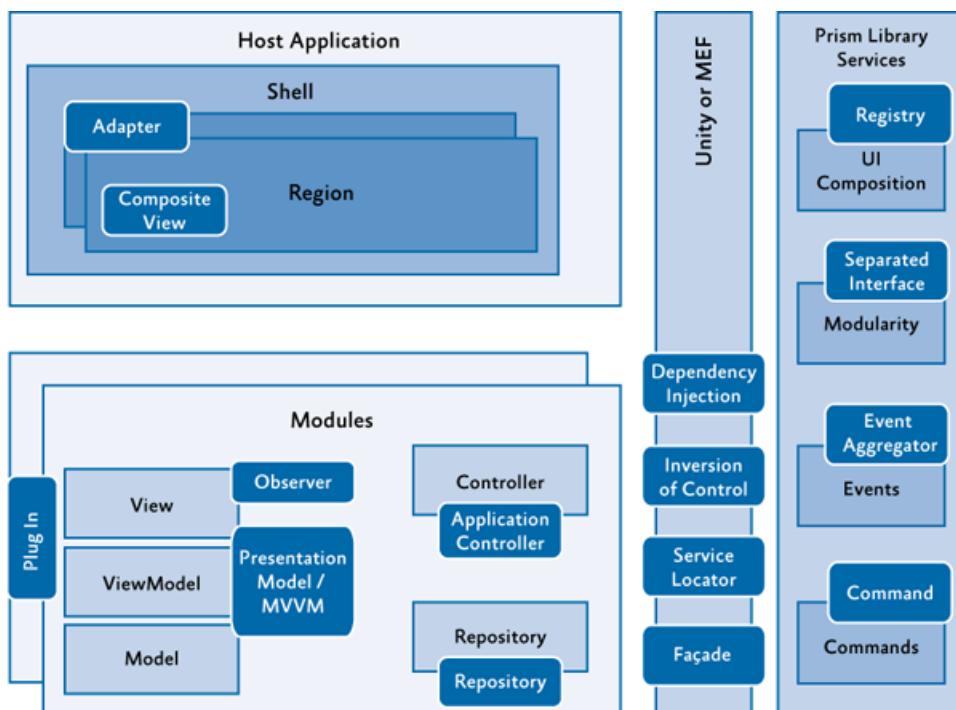
view injection. A way to add, show, or remove views in a region by adding or removing instances of a view to a region. The code interacting with the region does not have direct knowledge of how the region will handle displaying the view.

view model. Encapsulates the presentation logic and state for the view. It is responsible for coordinating the view's interaction with any model classes that are required.

view model location. Locates and instantiates view models and associating to their respective views typically by using a convention base approach.

12: Patterns in the Prism Library

When you build applications, you typically encounter or employ patterns. In the Prism Library and example reference implementation, the guidance demonstrates the Adapter, Application Controller, Command, Composite and Composite View, Dependency Injection, Event Aggregator, Façade, Inversion of Control, Observer, Model-View-ViewModel (MVVM), Registry, Repository, Separated Interface, Plug-In, and Service Locator patterns that are briefly discussed in this appendix. The following illustration shows a typical composite application architecture using the Prism Library and some of the common patterns. A simpler application would likely encounter some of these patterns while using Prism, but not necessarily all of them.



Sample composite application architecture with common patterns

This section provides a brief overview of the patterns in alphabetical order and pointers to where you can see an example of each pattern in the Prism code.

Adapter

The Adapter pattern, as the name implies, adapts the interface of one class to match the interface expected by another class. In the Prism Library, the Adapter pattern is used to adapt regions to the Windows Presentation Foundation (WPF) **ItemsControl**, **ContentControl**, and **Selector**. To see the Adapters pattern applied, see the file `ItemsControlRegionAdapter.cs` in the Prism Library.

Application Controller Pattern

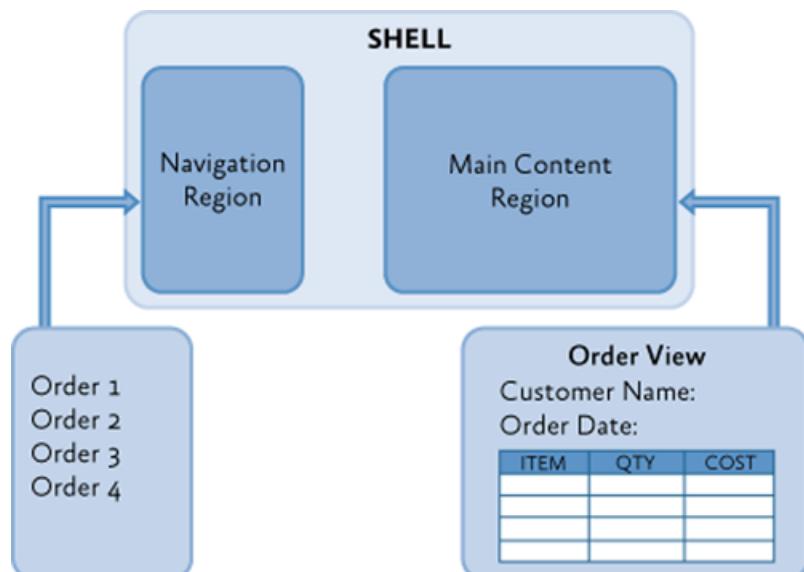
The Application Controller pattern allows you to separate the responsibility of creating and displaying views into a controller class. This kind of controller is a little different than the controller in an MVC application. The application controller's responsibility is to encapsulate the control of view presentation. It can take care of instantiating views; it does this by placing them in the appropriate container in the user interface (UI), switching between views that share the same container, and sometimes coordinates communication between views or view models. Even though the name of the pattern is Application Controller, controllers are often scoped to a subset of an application, such as a module controller in a Prism application or a controller that spans a set of related views. As a result, you will often have more than one controller in a Prism application. For an example implementation of this pattern, see the **OrdersController** class in the Stock Trader Reference Implementation (Stock Trader RI).

Command Pattern

The Command pattern is a design pattern in which objects are used to represent actions. A command object encapsulates an action and its parameters. This allows a decoupling of the invoker of the command and the handlers of the command. The Prism.Mvvm Library provides a **CompositeCommand** that allows combining of multiple **ICommand** items and a **DelegateCommand** that allows a ViewModel or controller to provide an **ICommand** that connects to local methods for execution and notification of ability to execute. To see the usage of the **CompositeCommand** and the **DelegateCommand** in the Stock Trader RI, see the files StockTraderRCommands.cs and OrderDetailsViewModel.cs.

Composite and Composite View

At the heart of a composite application is the ability to combine individual views into a composite view. Frequently, the composing view defines a layout for the child views. For example, the shell of the application may define a navigation area and content area to host child views at run time, as shown in the following illustration.



Composition example

In the Stock Trader RI, this can be seen with the use of regions in the shell. The shell defines regions that modules locate and add views to during the initialization process. For examples of defining regions, see the `Shell.xaml` file.

Composite views do not have to be dynamically composed, as is the case when using Prism's regions. A composite view can also just be a view that is built up of several other child views that are statically composed through the UI definition. An example of this is child user controls that are declared in the Extensible Application Markup Language (XAML).

Dependency Injection Pattern

The Dependency Injection pattern is a specialized version of the Inversion of Control pattern (described later in this appendix) where the concern being inverted is the process of obtaining the needed dependency. Dependency Injection is used throughout the Stock Trader RI and the Prism Library. When using a container, the responsibility of construction is put on the container instead of the consuming class. During object construction, the dependency injection container resolves any external dependencies. Because of this, the concrete implementation of the dependencies can be changed more readily as the system evolves. This better supports testability and growth of a system over time due to looser coupling. The Stock Trader RI uses the Managed Extensibility Framework (MEF) to help manage dependencies between components. However, the Prism Library itself is not tied to a specific dependency injection container; you are free to choose whichever dependency injection container you want, but you must provide an adapter that implements the **IServiceLocator** interface. The Prism Library provides adapters for both the MEF and Unity Application Block (Unity). To see an example of a component with its dependencies resolved by injection in the Stock Trader RI, see the constructor in the `NewsController.cs` file. For examples using Unity, see the **ModuleInit** class in the UI Composition QuickStart.

Event Aggregator Pattern

The Event Aggregator pattern channels events from multiple objects through a single object to simplify registration for clients. In the Prism Library, a variation of the Event Aggregator pattern allows multiple objects to locate and publish or subscribe to events. To see the **EventAggregator** and the events it manages, see **EventAggregator** and the **PubSubEvent** in the `Prism.PubSubEvents` Library. To see the usage of the **EventAggregator** in the Stock Trader RI, see the file `WatchListViewModel.cs`.

Façade Pattern

The Façade pattern simplifies a more complex interface, or set of interfaces, to ease their use or to isolate access to those interfaces. The Prism Library provides façades for the container and the logging services to help isolate the library from changes in those services. This allows the consumer of the library to provide its own services that will work with the Prism Library. The **IServiceLocator** and **ILoggerFacade** interfaces define the façade interfaces the Prism Library expects when it communicates with a container or logging service.

Inversion of Control Pattern

Frequently, the Inversion of Control (IoC) pattern is used to enable extensibility in a class or framework. For example, a class designed with an eventing model at certain points of execution inverts control by allowing event listeners to take action when the event is invoked.

Two forms of the IoC pattern demonstrated in the Prism Library and Stock Trader RI include dependency injection and the Template Method pattern. Dependency injection is described earlier. In the Template Method pattern, a base class provides a recipe, or process, that calls virtual or abstract methods. Because of this, an inherited class can override appropriate methods to enable the behavior required. In the Prism Library, this is shown in the **UnityServiceLocatorAdapter** class. To see another example of using the Template pattern, see the file StockTraderRIBootstrapper.cs in the Stock Trader RI.

Observer Pattern

The Observer pattern seeks to decouple those interested in an object's state change from the changing object. In the .NET Framework, this is often seen through events. Prism demonstrates a variation of the Observer pattern to separate the request for interaction with the user from the actual chosen interaction. This is done through an **InteractionRequest** object that is often offered by a view model in the Model-View-ViewModel (MVVM) pattern.

This **InteractionRequest** is an object that encapsulates an event monitored by the view. When the view receives an interaction request, it can choose how to handle the interaction. A view may decide to display a modal window to provide feedback to the user, or it may display an unobtrusive notification without interrupting the user's workflow. Offering this request as an object provides a way to data-bind in WPF to the request and to specify the response without requiring code-behind in the view.

Model-View-ViewModel Pattern

Presentation Model is one of several UI patterns that focus on keeping the logic for the presentation separate from the visual representation. This is done to separate the concerns of visual presentation from that of visual logic, which helps improve maintainability and testability. Related UI patterns include Model-View-Controller (MVC) and Model-View-Presenter (MVP). The Model-View-ViewModel (MVVM) approach, demonstrated in the Prism's Stock Trader RI, is a specific implementation variant of the Presentation Model pattern.

The Prism Library itself is intended to be neutral with respect to choice of separated UI patterns. You can be successful with any of the patterns, although considering the facilities in WPF for data binding, commands, and behaviors, the MVVM pattern is the recommended approach and the Prism guidance provides documentation and samples to get you started using MVVM. To see examples of MVVM in the Basic MVVM QuickStart, see the files QuestionnaireView.xaml, QuestionnaireView.xaml.cs, and QuestionnaireViewModel.cs.

Registry Pattern

The Registry pattern specifies an approach to locating one or more objects from a well-known object. The Prism Library applies the Registry pattern when associating view types to a region. The **IRegionViewRegistry** interface and **RegionViewRegistry** class define a registry used to associate region names to the view types created when those regions are loaded. This registry is used in the `ModuleInit.cs` file in the UI Composition QuickStart.

Repository Pattern

A repository allows you to separate how you acquire data for an application from the code that needs the data. The repository represents a collection of domain objects that the application code can consume without needing to be coupled to the specific mechanism that retrieves those objects. The domain objects are part of the model of the application, and by obtaining those objects through a repository, the repository retrieval and update strategy can be changed without affecting the rest of the application. Additionally, the repository interface becomes an easy dependency to substitute for the purposes of unit testing.

Separed Interface and Plug-In

The ability to locate and load modules at run time opens greater opportunities for parallel development, expands module deployment choices, and encourages a more loosely coupled architecture. The following patterns enable this ability:

- **Separated Interface.** This pattern reduces coupling by placing the interface definition in a separate package from the implementation. When using Prism with Unity, each module implements the **IModule** interface. For an example of implementing a module in the UI Composition Quickstart, see the file `ModuleInit.cs`.
- **Plug-In.** This pattern allows the concrete implementation of a class to be determined at run time to avoid requiring recompilation when changing which concrete implementation is used or because of changes in the concrete implementation. In the Prism Library, this is handled through the **DirectoryModuleCatalog**, **ConfigurationModuleCatalog**, and the **ModuleInitializer**, which work together to locate and initialize **IModule** plug-ins. For examples of supporting plug-ins, see the files `DirectoryModuleCatalog.cs`, `ConfigurationModuleCatalog.cs`, and `ModuleInitializer.cs` in the Prism Library.

Note: MEF was designed to support the plug-in model, allowing components to declaratively export and import concrete implementations.

Service Locator Pattern

The Service Locator pattern solves the same problems that the Dependency Injection pattern solves, but it uses a different approach. It allows classes to locate specific services they are interested in without needing to know who implements the service. Frequently, this is used as an alternative to dependency

injection, but there are times when a class will need to use service location instead of dependency injection, such as when it needs to resolve multiple implementers of a service. In the Prism Library, this can be seen when the **ModuleInitializer** service resolves individual **IModules**. For an example of using the **UnityContainer** to locate a service in the UI Composition Quickstart, see the file `ModuleInit.cs`.

More Information

The following are references and links to the patterns found in the Stock Trader RI and in the Prism Library:

- Composite pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Adapter pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Façade pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Template Method pattern in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Observer pattern in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- [Exploring the Observer Design Pattern](#) on MSDN.
- [Repository pattern](#) in *Patterns of Enterprise Application Architecture* by Martin Fowler or the abbreviated version on his website.
- Inversion of Control containers and the [Dependency Injection](#) pattern on Martin Fowler's website.
- [Plugin pattern](#) on Martin Fowler's website.
- [Registry pattern](#) on Martin Fowler's website.
- [Presentation Model pattern](#) on Martin Fowler's website.
- [Event Aggregator pattern](#) on Martin Fowler's website.
- [Separated Interface pattern](#) on Martin Fowler's website.
- [MVC and MVP variants](#) on Martin Fowler's website.
- [Design Patterns: Dependency Injection](#) by Griffin Caprio on MSDN.
- [Model-View-ViewModel pattern](#) on John Gossman's blog.

For more information about the Unity Application Block, see [Unity Application Block](#) on MSDN.

- (1) Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

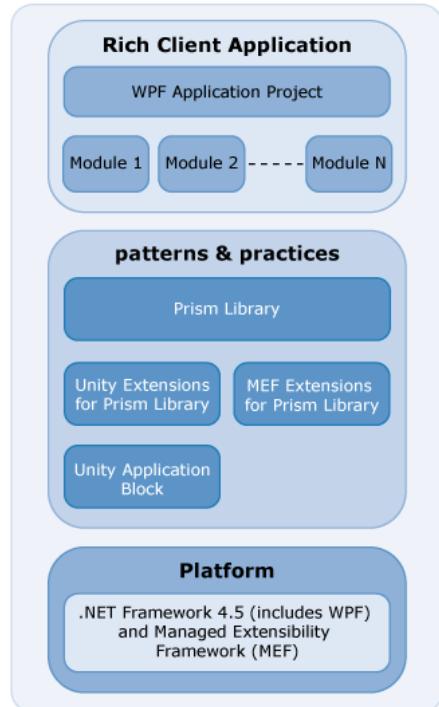
13: Prism Library

The Prism Library helps architects and developers create composite applications for Windows Presentation Foundation (WPF) using the Model-View-ViewModel pattern. The Prism Library can support those wanting to build a number of application styles with WPF, but it was primarily constructed for applications composed of discrete, functionally complete pieces that work together to create a single, integrated user interface (UI), often referred to as a composite application. The Prism Library accelerates the development of applications using proven design patterns.

The Prism Library is primarily designed to help architects and developers create applications that need to accomplish the following:

- Build clients composed of independent, yet cooperating, modules or pieces.
- Separate the concerns of module builders from the concerns of the shell developer; by doing this, business units can concentrate on developing domain-specific modules instead of the WPF architecture.
- Separate the concerns of presentation, presentation logic, and application model through support for presentation model patterns such as Model-View-ViewModel (MVVM).
- Use an architectural infrastructure to produce a consistent and high quality integrated application.

When building your application with the Prism Library, you may use the Unity Extensions for the Prism Library and the Unity Application Block (Unity) or the Managed Extensibility Framework (MEF) Extensions for the Prism Library and MEF. These are built on the .NET Framework 4.5 for WPF, as shown in the following illustration.



Composite application package

The Prism Library addresses common requirements for building both composite and non-composite applications on the WPF platform. As a whole, the Prism Library accelerates development by providing the services and components to address these needs.

The Prism Library ships signed binaries through NuGet packages to allow you to take advantage of Prism immediately without the need to compile and as source in case you want to make modifications or just see how it works.

Add Reference using NuGet and Accessing the Library Source Code

Add references to the Prism binaries in your code by searching NuGet for Prism. The [Prism](#) NuGet package is dependent on the [Prism.Composition](#), [Prism.Interactivity](#), [Prism.Mvvm](#), and [Prism.PubSubEvents](#) NuGet packages.

The [Prism NuGet package](#) will download the Prism.Composition, Prism.PubSubEvents, Prism.Mvvm, and Prism.Interactivity NuGet packages.

The source for the Prism library can be downloaded from <http://aka.ms/prism-wpf-code>.

Organization of the Prism Library

The Prism Library targeted for desktop applications consists of eight assemblies:

- **Microsoft.Practices.Prism.Composition.** This assembly contains interfaces and components to help build composite applications. These components include the **ModuleManager**,

ModuleCatalog, and **Bootstrapper**. Additionally, this assembly contains the **RegionManager** component that helps compose the user interface from multiple parts..

- **Microsoft.Practices.Prism.Interactivity**. This assembly contains behaviors and actions for interactions with the UI based on Blend for Visual Studio 2013 Behaviors (available in the Blend SDK), largely in support of the MVVM pattern. This includes **InteractionRequest**, **InteractionRequestTrigger**, **Confirmation**, and **Notification**. Additionally the **PopupWindowAction** responds to the **InteractionRequestTrigger**.
- **Microsoft.Practices.Prism.UnityExtensions**. This assembly provides components to use the Unity Application Block (Unity) with the Prism Library. These components include **UnityBootstrapper** and **UnityServiceLocatorAdapter**.
- **Microsoft.Practices.Prism.MefExtensions**. This assembly provides components to use Managed Extensibility Framework (MEF) with the Prism Library. These components include **MefBootstrapper** and **MefServiceLocatorAdapter**.
- **Microsoft.Practices.Prism.PubSubEvents (Event Aggregator)**. This assembly contains interfaces and components to help send loosely coupled messages between modules. The components include the **PubSubEvents** and **EventAggregator**.
- **Microsoft.Practices.Prism.Mvvm**. This assembly contains interfaces and components to help implement the MVVM pattern. These components include **BindableBase**, **PropertySupport**, **ViewModelLocationProvider**, **DelegateCommand**, and **CompositeCommand**.
- **Microsoft.Practices.Prism.Mvvm.Desktop**. This assembly contains the components specific to WPF, which includes the platform specific dependency property called **AutoWireViewModelProperty**.
- **Microsoft.Practices.Prism.SharedInterfaces**. This assembly contains the shared Prism interfaces **IActiveAware**.

The Prism Library Source

The source for Prism.Composition, Prism.Interactivity, Prism.UnityExtensions, Prism.MefExtensions, Prism.SharedInterfaces, Prism.PubSubEvents, and Prism.Mvvm assemblies can be found in the PrismLibrary folder where Prism is installed. These assemblies target WPF applications.

Modifying the Library

If you want to modify the Prism Library, you can replace the NuGet referenced assemblies with your own version of the binaries.

Running the Tests

If you modify the Prism Library and want to verify that existing functionality is not broken, execute the unit tests for the projects. To run all the desktop unit tests in the solution file PrismLibrary/Desktop.sln, on the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

More Information

Prism's community sites are:

- Prism: <http://www.codeplex.com/Prism>.
 - PubSubEvents (Event Aggregator): <http://pnppubsub.codeplex.com>.
 - MVVM: <http://pnpmvvm.codeplex.com>.
-

For more information about Unity, see the following:

- "Unity Application Block" on MSDN: <http://www.msdn.com/unity>.
 - Unity community site on CodePlex: <http://www.codeplex.com/unity>.
-

For more information about MEF, see the following:

- "[Managed Extensibility Framework Overview](#)" on MSDN.
 - MEF community site on CodePlex: <http://mef.codeplex.com/>.
-

For more information about service locator, see the Common Service Locator on CodePlex:

<http://commonservicelocator.codeplex.com/>.

14: Upgrading from Prism Library 4.1

This topic describes how to upgrade a solution from version 4.1 to version 5.0 of the Prism Library and the major changes that you should be aware of if you are considering upgrading to the 5.0 version.

ViewModel Namespace and API Changes

The classes in the **Microsoft.Practices.Prism.ViewModel** namespace were made obsolete but still exist in Prism 5.0. You should use the classes from the **Microsoft.Practices.Prism.Mvvm** portable class library with **Microsoft.Practices.Prism.Mvvm** namespace. The **BindableBase** class replaces the **NotificationObject** class. If you need to implement **INotifyPropertyChanged** event, you should now use **BindableBase**, and use the **SetProperty** method in the property setter, which verifies if the value actually changed and if so, sets the backing field and raises the **PropertyChanged** event.

The 4.1 code was as follows:

C#

```
this.RaisePropertyChanged(() => this.WatchListItems);

. . .

if (value != this.timeInForce)
{
    this.timeInForce = value;
    this.RaisePropertyChanged(() => this.TimeInForce);
}
```

The 5.0 code is as follows:

C#

```
OnPropertyChanged(() => this.WatchListItems);

. . .

SetProperty(ref this.timeInForce, value);
```

The Prism NuGet package will manage the changes to the new Prism assemblies.

If you decide to manually update your references, the **Microsoft.Practices.Prism.ViewModel** namespace now requires you to add the following references:

- Microsoft.Practices.Prism.Mvvm
- Microsoft.Practices.Prism.Mvvm.Desktop
- Microsoft.Practices.Prism.ShareInterfaces

Alternatively you can add a NuGet reference to the **Prism.Mvvm** package if you only want the **Prism.Mvvm** APIs.

EventAggregator Namespace and API Changes

The classes in the Events namespace were made obsolete but still exist in Prism 5.0. You should use the classes from the **Prism.PubSubEvents** portable class library with the **Prism.PubSubEvents** namespace. The **PubSubEvent** class replaces the **CompositePresentationEvent** class.

The Prism NuGet package will manage the changes to the new Prism assemblies.

If you decide to manually update your references, you will now need to add a reference to the following:

- **Microsoft.Practices.Prism.PubSubEvents**
-

Alternatively you can insert a NuGet reference to the Prism.PubSubEvents NuGet package if you only want the Prism.PubSubEvents APIs.

Regions Namespace API Changes

The **UriQuery** class was replaced with the **NavigationParameters** class and moved to the **Regions** namespace. Previous functionality remains the same, and support for object parameters was added. The **RequestNavigate** method defined in the interface **INavigateAsync** was updated to allow the passing of **NavigationParameters**.

Commands Assembly Changes

The following classes from the **Commands** namespace were moved from the Prism library to the Prism.Mvvm portable class library:

- **CompositeCommand**
 - **DelegateCommand**
 - **DelegateCommandBase**
 - **WeakEventHandlerManager**
-

For these classes you will only need to change your references to the Prism.Mvvm assembly.

The **CommandBehaviorBase** class was moved to the **Prism.Interactivity** namespace from the **Commands** namespace. The **ExecuteCommand** method now takes an object as a parameter.

The **ButtonBaseClickCommandBehavior** and **Click** classes were removed as they were obsolete last release.

Prism NuGet Packages

The following signed Prism assemblies can now be referenced from NuGet:

- [Prism 5.0](#)
- [Prism.Composition 5.0](#)
- [Prism.Interactivity 5.0](#)
- [Prism.PubSubEvents 1.0](#)

- [Prism.Mvvm 1.0](#)
 - [Prism.UnityExtensions 5.0](#)
 - [Prism.MefExtensions 5.0](#)
-

The [Prism NuGet package](#) will download the Prism.Composition, Prism.PubSubEvents, Prism.Mvvm, and Prism.Interactivity NuGet packages.

15: Extending the Prism Library

Prism contains assets that represent recommended practices for Windows Presentation Foundation (WPF) client development. Developers can use an unmodified version of the guidance to create composite applications using the Model-View-ViewModel (MVVM) pattern. However, because each application is unique, you should analyze whether Prism is suitable for your particular needs. In some cases, you will want to customize the guidance to incorporate your enterprise's best practices and frequently repeated developer tasks.

The Prism Library can serve as the foundation for your WPF client applications. The Prism Library was designed so that significant pieces can be customized or replaced to fit your specific scenario. You can modify the source code for the existing library to incorporate new functionality. Developers can replace key components in the architecture with ones of their own design because of the reliance on a container to locate and construct key components in the architecture. In the library, you can even replace the container itself if you want. Other common areas to customize include creating or customizing the bootstrapper to select a module discovery strategy for module loading, calling your own logger, using your own container, and creating your own region adapters.

This topic describes several key extensibility points in the Prism Library. These tend to be more advanced topics and are not expected to be performed for most developers using the Prism Library. A solid understanding of the goals and design decisions in the Prism Library will help to ensure any extensions to Prism functionality don't create side effects or degrade the architecture. It is recommended that the main topics of the Prism documentation are read before extending the Prism Library. Most of the techniques described in this document rely on replacing or modifying Prism Library default configuration during the bootstrapping sequence when the application starts, so reading the section [Prism Key Concepts](#) in [Introduction](#) is a prerequisite.

The following are the key extensibility points in the Prism Library covered in this topic:

- **Application Bootstrapper.** This demonstrates the key extensibility point of the Prism Library.
- **Modularity.** This demonstrates extensibility points when building a modular application.
- **Region Management.** This demonstrates extending how regions behave, how they are hosted, and how they interact with their views.
- **Region Navigation.** This demonstrates how to change your logical navigation structure.
- **View Model Locator.** This demonstrates how to modify the conventions when using the View Model Locator.

Guidelines for Extensibility

Use these guidelines when you extend the Prism Library. You can extend the library by adding or replacing services, modifying the source code, or adding new application capabilities.

Exposing Functionality

A library should provide a public API to expose its functionality. The interface of the API should be independent of the internal implementation. Developers should not be required to understand the library design or implementation to effectively use its default functionality. Whenever possible, the API should apply to common scenarios for a specific functionality.

Extending Libraries

The Prism Library provides extensibility points that developers can use to tailor the library to suit their needs. For example, when using the Prism Library, you can replace the provided logging service with your own logging service.

You can extend the library without modifying its source code. To accomplish this, you should use extensibility points, such as public base classes or interfaces. Developers can extend the base classes or implement the interfaces and then add their extensions to the library. When defining the set of extensibility points, consider the effect on usability. A large number of extensibility points can make the library complicated to use and difficult to configure.

Some developers may be interested in customizing the code, which means they will modify the source code instead of using the extension points. To support this effort, the library design should provide the following:

- It should follow object-oriented design principles whenever practical.
 - It should use appropriate patterns.
 - It should efficiently use resources.
 - It should adhere to security principles (for example, distrust of user input and principle of least privilege).
-

Recommendations for Modifying the Prism Library

When modifying the source code, follow these best practices:

- Make sure you understand how the library works by reading the topics that describe its design. Consider changing the library's namespace if you significantly alter the code or if you want to use your customized version of the library together with the original version.
 - Consider authoring your own assemblies that use the Prism Library's built in extensibility points first before altering or replacing the Prism Library binaries.
 - Use strong naming. A strong name allows the assembly to be uniquely identified, versioned, and checked for integrity. You will need to generate your own key pair to sign your modified version of the application block. For more information, see [Strong-Named Assemblies](#) on MSDN. Alternatively, you can choose to not sign your custom version. This is referred to as weak naming.
-

Extensibility Points in the Prism Library

This section outlines the extension points, by functional area, and associated information for extending the library.

Container and Bootstrapper

The Prism Library directly supports both the Unity Application Block (Unity) and Managed Extensibility Framework (MEF) as dependency injection containers; however, because the container is accessed through the **IServiceLocator** interface, the container can be replaced.

Each Prism application configures the Prism Library through a bootstrapper class. Each stage in the bootstrapping process is replaceable, as well as the sequence itself. The bootstrapper provides a key extensibility point to replace default implementations with custom implementations or register additional types and services.

Logging

Some Prism Library components log information, warning messages, or error messages. To avoid a dependency on a particular logging approach, it logs these messages to the **ILoggerFacade** interface. A common extension is to provide a custom logger for specific applications.

Modules

The Prism Library provides various ways to populate the module catalog and load modules; however, your scenario may have needs that the library does not provide.

Module loading includes the following three phases, which can be customized:

- **Module discovery.** This is the process of populating a module catalog. Frequently, this is done directly or by sweeping a directory, but your application may need to do this some other way, such as from a database. In these cases, you can create a custom catalog that populates itself from an appropriate source.
- **Module retrieval and loading.** This is the process of acquiring the module binaries locally and loading the module into the current application domain. The library provides the **FileModuleTypeLoader**, but you may want to implement your own retrieval strategy.
- **Module initialization.** This is the process of initializing a module. In the library, this is done by the **ModuleInitializer**, but it can be replaced by providing a new object that implements **IModuleInitializer**.

Regions

The Prism Library provides default control adapters for enabling a control as a region. Extensions around regions may involve providing custom region adapters, custom regions, or replacing the region manager. If you have a custom WPF control or a third-party control that does not work with the provided region adapters, you may want to create custom region adapters that will. It is also possible to replace the default **RegionManager** by supplying a new **IRegionManager** in the container.

Region Navigation

The region feature of the Prism Library also supports navigation, including back/forward journaling support. Views within a region can extend and participate in navigation through the **INavigationAware** interface. Developers familiar with Silverlight navigation features will find **Region** analogous to the **Frame** class. Region navigation supports several extensibility points that make it possible to change the logical navigation structure of the application in addition to replacement of navigation services.

The **RegionNavigationContentLoader** class provides the ability to load content into a region based on the **NavigationContext**. If the content being navigated to is already in the region, the **RegionNavigationContentLoader** will locate that content and make it active instead of creating new content to add to the region. The **RegionNavigationContentLoader.GetCandidatesFromRegion** method searches the region's views matching them by type. However, it is possible to have a view whose type does not match the type used to resolve it. For example, you could register your view with a dependency injection container using a "friendly" name that does not match the name of your view type.

C#

```
[Export("FriendlyName")]
public class MyViewType
```

The Prism Library ships with **UnityRegionNavigationContentLoader** and **MefRegionNavigationContentLoader** that override the base **GetCandidatesFromRegion** method providing special handling necessary to find view types based on possible friendly name registration. If you are not using either **UnityRegionNavigationContentLoader** or **MefRegionNavigationContentLoader**, then make sure to add handling to a subclass of **RegionNavigationContentLoader** specific to the dependency injection container you are using.

Container and Bootstrapper

The Prism Library contains the **Bootstrapper** base class. The Unity and MEF components derive from this class as **UnityBootstrapper** and **MefBootstrapper**, respectively. The **Bootstrapper** base class defines an abstract **Run** method that leaves the exact sequencing of the process up to the derived classes. Almost every method is marked as virtual, allowing you to override individual methods to customize and extend the bootstrapping process.

For most type instantiation, a bootstrapper will use the dependency injection container. However, there are some parts of the bootstrapping process that cannot use the container:

- **Creating the logger.** Generally, the logger is created first (before the container) because the bootstrapper needs to log information about creating the container. For more information about changing the logging implementation, see the section, "Logging."
- **Creating and configuring catalogs.** Catalogs (for example, **ModuleCatalog** and **AggregateCatalog**) are created before the container because they are used during construction of the container.

- **Creating the shell.** Because the shell may already exist before the bootstrapping sequence runs, the **CreateShell** method is left as abstract for the application developer to implement. The application developer can use the container to instantiate or locate the shell because the container has been created and initialized.

Replacing Default Prism Library Types

There may be times when you need to change or extend the underlying implementation of a Prism Library type for an application. Because the Prism Library relies on dependency injection, you can replace the type during the bootstrapping sequence and both your application and the Prism Library will use the new type.

Replacing Default Types Using Unity

Any replacement types registered in the container before the **UnityBootstrapper.ConfigureContainer** method is called will replace the type. The **ConfigureContainer** default implementation uses the **RegisterTypeIfMissing** method to only add a Prism Library type if that associated interface is not already registered.

To replace Prism Library types in Unity, first derive your new type from the interface or class you want to replace. The following code example shows a replacement for the **IEventAggregator** interface.

C#

```
// when using Unity
public class ReplacementEventAggregator : IEventAggregator
{
    // ...
}
```

Now that you have the replacement type, override the **ConfigureContainer** method in the bootstrapper and register interface and type before calling the base class. The following code example shows how to register the replacement for the **IEventAggregator**.

C#

```
// when using Unity
protected override void ConfigureContainer()
{
    this.RegisterTypeIfMissing(typeof(IEventAggregator),
    typeof(ReplacementEventAggregator), true);

    base.ConfigureContainer();
}
```

Replacing Default Types Using MEF

Any replacement types registered in the container before the **MefBootstrapper.ConfigureContainer** method is called will replace the type. The **ConfigureContainer** default implementation only adds a Prism Library type if that associated interface is not already registered.

To replace Prism Library types in MEF, first derive your new type from the interface you want to replace and apply the appropriate MEF Export attributes to it. The following code example shows a replacement for the **IEventAggregator** interface.

C#

```
// when using MEF
[Export(typeof(IEventAggregator))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class ReplacementEventAggregator : IEventAggregator
{
    // ...
}
```

Now that you have the replacement type, override the **ConfigureAggregateCatalog** method in the bootstrapper and add a catalog that contains the type to the **AggregateCatalog**. The following code example shows how to use a **TypeCatalog** to add the replacement type. An **AssemblyCatalog** could also have been used.

C#

```
// when using MEF
protected override void ConfigureAggregateCatalog()
{
    this.AggregateCatalog.Catalogs.Add(new
TypeCatalog(typeof(ReplacementEventAggregator)));

    base.ConfigureAggregateCatalog();
}
```

Registering Non-MEF Attributed Types with the MEF Container

Registering types with MEF is simple if you own the code and can take a direct dependency on MEF, because all you need to do is add an **Export** attribute to the types. However, in some situations, you may need to register types with MEF when you cannot take a direct dependency on the MEF assemblies. This problem was encountered while the developers added MEF support to Prism because one of the design goals was to ensure that the core Prism libraries were not container-specific. This meant that the **Microsoft.Practices.Prism** assembly could not reference **System.ComponentModel.Composition** and use the **Export** attribute. Instead, the team created derived classes in the **Microsoft.Practices.Prism.MefExtensions** assembly that derived from the types the team wanted to expose and exported the appropriate type. The following code example from the **MefRegionManager** class shows an example of this approach by deriving from **RegionManager** and exporting the new type as an **IRegionManager**.

C#

```
[Export(typeof(IRegionManager))]
public class MefRegionManager : RegionManager
{}
```

Creating a Minimal Bootstrapper

Some applications do not use many of the features in the Prism Library. In some cases, application developers may want the absolute minimum level of services—only dependency injection and service location. To do this, override the **ConfigureContainer** method in the bootstrapper and implement the following.

C#

```
// when using UnityBootstrapper
protected override void ConfigureContainer()
{
    // Base class implementation deliberately not called
    // base.ConfigureContainer();

    this.Container.AddNewExtension<UnityBootstrapperExtension>();
    Container.RegisterInstance<ILoggerFacade>(Logger);
    this.Container.RegisterInstance(this.ModuleCatalog);
    RegisterTypeIfMissing(typeof(IServiceLocator),
    typeof(UnityServiceLocatorAdapter), true);
}

protected override RegionAdapterMappings ConfigureRegionAdapterMappings()
{
    return null;
}

protected override IRegionBehaviorFactory ConfigureDefaultRegionBehaviors()
{
    return null;
}
```

Note: The overrides of the region adapters and mappings are required because Unity cannot determine the appropriate concrete type to return when an implementation of an interface is requested. These calls associate the concrete type to return for each interface. When concrete types are requested Unity is able to directly resolve them by instantiating that type.

C#

```
// when using MEFBootstrapper
protected override void ConfigureContainer()
{
    // Base class implementation deliberately not called
    // base.ConfigureContainer();

    this.Container.ComposeExportedValue<ILoggerFacade>(this.Logger);
    this.Container.ComposeExportedValue<IServiceLocator>(new
MefServiceLocatorAdapter(this.Container));
    this.Container.ComposeExportedValue<AggregateCatalog>(this.AggregateCatalog);
}
```

Changing Dependency Injection Containers

If you want to use Prism with a container other than Unity or MEF in your application, there are several things you need to do. First, you need to write a Service Locator adapter for your container. You can use the **MefServiceLocatorAdapter** and the **UnityServiceLocatorAdapter** as examples of how this can be done. You will also need to write a container-specific bootstrapper class. Next, you need to create a new container-specific bootstrapper, derived from the **Bootstrapper** class, and implement the necessary methods, using the **MefBootstrapper** and **UnityBootstrapper** as examples.

Logging

The Prism Library is designed to log messages throughout the library. To do this logging in a way that is not tied to a specific logging library, the Prism Library uses a logging façade, **ILoggerFacade**, to log its messages. This interface contains a single method named **Log** that logs messages. By default, the **UnityBootstrapper** and **MefBootstrapper** create a **TextLogger** as the designated logger.

There are three steps for creating and integrating a custom logger:

1. Create a class that implements the **ILoggerFacade** interface.
2. Implement the **Log** method.
3. In your application bootstrapper class, override the **CreateLogger** method to return a new instance of your logging class.

The **Log** method in the **ILoggerFacade** interface takes three parameters:

- **Message**. This is the message to be logged.
- **Category**. This is the category of the event to be logged. The valid options are **Debug**, **Exception**, **Info**, and **Warn**.
- **Priority**. This is the priority of the event to be logged. The valid options are **None**, **High**, **Medium**, and **Low**.

The following code example shows a custom logger that wraps some other logging framework that takes only a string.

C#

```
// CustomLogger
using Microsoft.Practices.Prism.Logging;
...

public class CustomLogger : ILoggerFacade
{
    public void Log(string message, Category category, Priority priority)
    {
        string messageToLog =
            String.Format(System.Globalization.CultureInfo.InvariantCulture,
```

```

    "{1}: {2}. Priority: {3}. Timestamp:{0:u}." ,
    DateTime.Now,
    category.ToString().ToUpperInvariant(),
    message,
    priority.ToString());

    MyOtherLoggingFramework.Log(messageToLog);
}
}

```

C#

```

// ApplicationBootstrapper
using Microsoft.Practices.Prism.Logging;
...

public class ApplicationBootstrapper : UnityBootstrapper
{
    ...
    protected override ILoggerFacade CreateLogger()
    {
        return new CustomLogger();
    }
}

```

Modules

The following sections describe how the modularity features can be extended during registration, assembly discovery, type discovery, and module initialization.

Adding Features to the Module Catalog

The Prism Library provides **ModuleCatalog** as both a class you can populate directly through the **AddModule** methods, or you can derive from add methods to populate the **Items** property.

The **ModuleCatalog** class in the Prism Library provides a lot of additional capabilities beyond the **IModule** interface. There are many different overloads of the **AddModule** method, module group dependency checking, and sorting. There are several ways to extend the functionality of the **ModuleCatalog**:

- **Derive from ModuleCatalog.** If you need to change the behavior of **ModuleCatalog**, derive a new class and override one of the virtual methods.
- **Write extension methods on IModuleCatalog.** If you need additional functionality in your application where you use **IModuleCatalog**, write an extension method on the interface.
- **Write extension methods on ModuleCatalog.** If you need additional functionality, but only in places where you use **ModuleCatalog**, write an extension method on the concrete type.

Discovering Modules from a Custom Source

The Prism Library supports populating the module catalog from application configuration and from a XAML file. You can extend Prism in your application to support loading from other data sources, such as a web service, database, or other external files.

The following describes several ways to populate the catalog.

- **Use the static `CreateFromXaml` method.** If your data is already in the **Modularity:ModuleCatalog** XAML schema, or if it can easily be converted, you can use this method to directly populate a **ModuleCatalog**.
- **Replace the `IConfigurationStore` in the `ConfigurationModuleCatalog`.** If you are running a WPF desktop application, you can implement an `IConfigurationStore` to return the module section for the `ConfigurationModuleCatalog`.
- **Derive from `ModuleCatalog`.** You can also follow the example of the `ConfigurationModuleCatalog` to derive from `ModuleCatalog`, acquire your data, and then call the `AddModule` method to populate the catalog.

The follow code examples show how to load a custom configuration module file from disk.

C#

```
// Bootstrapper
protected override Microsoft.Practices.Prism.Modularity.IModuleCatalog
CreateModuleCatalog()
{
    ConfigurationModuleCatalog catalog = new ConfigurationModuleCatalog();
    catalog.Store = new MyModuleCatalogStore();
    return catalog;
}
```

C#

```
// MyModuleCatalogStore
public class MyModuleCatalogStore : IConfigurationStore
{
    public ModulesConfigurationSection RetrieveModuleConfigurationSection()
    {
        ExeConfigurationFileMap fileMap = new ExeConfigurationFileMap()
        {
            ExeConfigFilename = "MyModuleCatalog.config"
        };

        Configuration configuration =
ConfigurationManager.OpenMappedExeConfiguration(fileMap,
ConfigurationUserLevel.None);
        return configuration.GetSection("modules") as
ModulesConfigurationSection;
    }
}
```

Retrieving and Loading Modules from a Custom Assembly Source

If your application has a packaging or distribution mechanism other than assemblies, you can implement your own **IModuleTypeLoader** to download and access types.

The Prism 4.1 Library **MefXapModuleTypeLoader** class is an example of this. It uses the MEF **DeploymentCatalog** to download XAP files, locate the assemblies, and register them with the MEF catalog.

Each **IModuleTypeLoader** implements the **CanLoadModuleType** method to allow the **ModuleManager** to determine the appropriate type loader to use for obtaining a module. The following code example shows the **MefXapModuleTypeLoader** implementation.

C#

```
// MefXapModuleTypeLoader.cs
public bool CanLoadModuleType(ModuleInfo moduleInfo)
{
    if (moduleInfo == null)
    {
        throw new ArgumentNullException("moduleInfo");
    }

    if (!string.IsNullOrEmpty(moduleInfo.Ref))
    {
        Uri uriRef;
        return Uri.TryCreate(moduleInfo.Ref, UriKind.RelativeOrAbsolute, out uriRef);
    }

    return false;
}
```

After you have your module type loader, you need to ensure it is in the **ModuleManager**'s collection of type loaders. The following code example is from the **Prism.MefExtensions.Silverlight** project.

C#

```
// MefModuleManager.Silverlight.cs
public override IEnumerable<IModuleTypeLoader> ModuleTypeLoaders
{
    get
    {
        if (this.mefTypeLoaders == null)
        {
            this.mefTypeLoaders = new List<IModuleTypeLoader>()
                { this.MefXapModuleTypeLoader };
        }
        return this.mefTypeLoaders;
    }
    set
    {
        this.mefTypeLoaders = value;
    }
}
```

```

    }
}

```

Changing How Modules Are Initialized

In addition to the **ModuleCatalog**, the **ModuleManager** provides many virtual functions that can be overridden to change how modules are loaded and initialized. Integrating with the MEF required the **MefModuleManager** to override several methods in the **ModuleManager**. There are several ways to change the behavior:

- **Derive from ModuleManager.** If you need to change the fundamental behavior of the module loading and initialization sequence, derive from a new class and override virtual methods.
- **Replace IModuleInitializer.** If you need to change how module types are instantiated and initialized, replace **IModuleInitializer**.
- **Write a custom IModuleTypeLoader.** If you need to change how assemblies are loaded and module types discovered within assemblies, write a custom **IModuleTypeLoader**. For more information, see the section, [Retrieving and Loading Modules from a Custom Assembly Source](#).

Regions

The following sections describe how the region management features of the Prism Library can be extended when regions are attached to controls, how regions behave, and how a region discovers its views.

Region Adapters

Region adapters control how items placed in a region interact with the host control. The following sections describe how to extend this behavior by creating a custom region adapter and controlling the registration of the adapters.

Creating a Custom Region Adapter

To expose a UI control as a region, a region adapter is used. Region adapters are responsible for creating a region and associating it to the control. By doing this, developers can manage the UI control's contents in a consistent way through the **IRegion** interface. Each region adapter adapts a particular type of UI control. The Prism Library provides three region adapters out-of-the-box:

- **ContentControlRegionAdapter.** This adapter adapts controls of type **System.Windows.Controls.ContentControl** and derived classes.
- **SelectorRegionAdapter.** This adapter adapts controls derived from the class **System.Windows.Controls.Primitives.Selector**, such as the **System.Windows.Controls.TabControl** control.

- **ItemsControlRegionAdapter.** This adapter adapts controls of type `System.Windows.Controls.ItemsControl` and derived classes.

There are some scenarios in which none of the preceding region adapters suit the developer needs. In those cases, custom region adapters can be created to adapt controls not supported by the Prism Library out-of-the-box.

Region adapters implement the `Microsoft.Practices.Prism.Regions.IRegionAdapter` interface. This interface defines a single method named `Initialize` that takes the object to adapt and returns a new region associated with the adapted control. The interface definition is shown in the following code.

C#

```
public interface IRegionAdapter
{
    IRegion Initialize(object regionTarget, string regionName);
}
```

To create a region adapter, you derive your class from `RegionAdapterBase<T>` and implement the `CreateRegion` and `Adapt` methods. Optionally, override the `AttachBehaviors` method to attach special logic to customize the region behavior. If you want to interact with the control that hosts the region, you should also implement `IHostAwareRegionBehavior`.

The `CreateRegion` method is an abstract method defined in the `RegionAdapterBase` class. It returns a region instance (an object that implements the `IRegion` interface) to be associated with the adapted control. The Prism Library provides the following region implementations out-of-the-box:

- **Region.** This region allows multiple active views. This is the region used for controls derived from the `Selector` class.
- **SingleActiveRegion.** This region allows a maximum of one active view at a time. This is the region used for `ContentControl` controls.
- **AllActiveRegion.** This region keeps all the views in it active. Deactivation of views is not allowed. This is the region used for `ItemsControl` controls.

The `Adapt` method is also an abstract method defined in the `RegionAdapterBase` class. It adapts the control to the region created earlier. The `Adapt` method takes two parameters: the region with which the adapted control has to be associated and the control to adapt. The following code example shows the `ContentControlRegionAdapter`.

C#

```
public class ContentControlRegionAdapter : RegionAdapterBase<ContentControl>
{
    public ContentControlRegionAdapter(IRegionBehaviorFactory regionBehaviorFactory)
        : base(regionBehaviorFactory)
    {
    }
```

```

protected override void Adapt(IRegion region, ContentControl regionTarget)
{
    if (regionTarget == null) throw new ArgumentNullException("regionTarget");
    bool contentIsSet = regionTarget.Content != null;
    contentIsSet = contentIsSet || (BindingOperations.GetBinding(regionTarget,
ContentControl.ContentProperty) != null);

    if (contentIsSet)
    {
        throw new
InvalidOperationException(Resources.ContentControlHasContentException);
    }

    region.ActiveViews.CollectionChanged += delegate
    {
        regionTarget.Content = region.ActiveViews.FirstOrDefault();
    };

    region.Views.CollectionChanged +=
        (sender, e) =>
    {
        if (e.Action == NotifyCollectionChangedEventArgs.Add &&
region.ActiveViews.Count() == 0)
        {
            region.Activate(e.NewItems[0]);
        }
    };
}

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

}

```

Note: The region adapter will be registered as a singleton service and will be kept alive throughout the application's lifetime, so make sure you do not keep references to possibly shorter lived objects, such as UI controls or region instances.

Region adapter mappings are used by the region manager service to associate the correct region adapters for XAML-defined regions. The following section describes how to customize the registration of region adapter mappings.

Customizing the Region Adapter Mappings

One phase of the bootstrapping process is to register the default region adapter mappings. These mappings are used by the region manager to associate the correct adapters for XAML-defined regions.

By default, an **ItemsControlRegionAdapter**, a **ContentControlRegionAdapter**, and a **SelectorRegionAdapter** are registered. For more information about these adapters, see [Composing the User Interface](#).

The following code example shows the default implementation of the **ConfigureRegionAdapterMappings** method. To customize the registration of region adapters, override this method in your applications bootstrapper.

C#

```
// Bootstrapper.cs
protected virtual RegionAdapterMappings ConfigureRegionAdapterMappings()
{
    RegionAdapterMappings regionAdapterMappings =
ServiceLocator.Current.GetInstance<RegionAdapterMappings>();
    if (regionAdapterMappings != null)
    {
        regionAdapterMappings.RegisterMapping(typeof(Selector),
ServiceLocator.Current.GetInstance<SelectorRegionAdapter>());
        regionAdapterMappings.RegisterMapping(typeof(ItemsControl),
ServiceLocator.Current.GetInstance<ItemsControlRegionAdapter>());
        regionAdapterMappings.RegisterMapping(typeof(ContentControl),
ServiceLocator.Current.GetInstance<ContentControlRegionAdapter>());
    }

    return regionAdapterMappings;
}
```

Region Behaviors

Region behaviors are used by the Prism Library to provide most of the functionality for a region. During the bootstrapping process, the bootstrapper registers the region behaviors that are attached to each region by default. Additionally, adapters may add behaviors only when a region is associated with a specific control type.

Adding a Region Behavior for All Regions

After you create a behavior, or extend an existing one, you can register it so it will be added to all new regions. You can do this by overriding the **ConfigureDefaultRegionBehaviors** in the bootstrapper. The following code example shows how to add a custom behavior for all regions.

C#

```
protected override IRegionBehaviorFactory ConfigureDefaultRegionBehaviors()
{
    IRegionBehaviorFactory factory = base.ConfigureDefaultRegionBehaviors();
    factory.AddIfMissing("MyBehavior", typeof(MyCustomBehavior));
}
```

[Adding a Region Behavior for a Single Region](#)

The following code example shows how to add a region behavior to a single region.

C#

```
IRegion region = regionManager.Region["Region1"];
region.Behaviors.Add("MyBehavior", new MyRegion());
```

[Replacing an Existing Region Behavior](#)

If you want to replace a default behavior with a different behavior, you can add it by overriding the **ConfigureDefaultRegionBehaviors** method in your application-specific bootstrapper and registering your behavior with the same key value as the default behavior. The Prism Library adds a default region behavior only if a behavior with that key has not already been added.

Occasionally, you may want to add or a replace a region behavior to regions on a particular view. If those regions are defined in XAML, like most regions are, the region may not be initially available for attaching your custom behavior. You will need to monitor the availability of the region and attach your behavior when the region becomes available. The following code example shows how to replace the **AutoPopulateBehavior** with your custom version when the region becomes available.

C#

```
public class MyView : UserControl
{
    public MyView()
    {
        InitializeComponent();

        ObservableObject<IRegion> observableRegion =
RegionManager.GetObservableRegion(this.MyRegionHostControl);

        observableRegion.PropertyChanged += (sender, args) =>
        {
            IRegion region = ((ObservableObject<IRegion>)sender).Value;
            region.Behaviors.Add(AutoPopulateBehavior.BehaviorKey,
                new CustomAutoPopulateBehavior());
        };
    }
}
```

[Removing a Region Behavior](#)

Although there is no way to remove an existing behavior after it is added, you can prevent a behavior from being added by overriding the **ConfigureDefaultRegionBehaviors** method in your application-specific bootstrapper.

Changing How Views Are Discovered

You may want to control how views are registered or created when using view discovery. The following are approaches to extending view discovery:

- **Custom RegionViewRegistry.** If you want to have extra control over registration of types (for example, scoping the registry) or control over the creation of your types, you should derive from this class.
- **Custom AutoPopulateBehavior.** If you want to change where the region discovers its registered views (if you do not want to use the **RegionViewRegistry**) or if you want to change which views are actually added to the region (for example, if you want to provide the ability to filter), you can create a custom **AutoPopulateBehavior** for a single region or change the default for all regions.

Region Navigation

The following sections describe how to extend the region navigation features of the Prism Library.

Changing Your Logical Navigation Structure

The region navigation features in the Prism Library use the type name of each view as the navigation Uniform Resource Identifier (URI). Your application may want to expose a URI navigation scheme independent of the view type names.

WPF applications can replace the **IRegionNavigationContentLoader** implementation to achieve the same result. Multi-targeted applications may also want to use this approach to maintain a single place where the URI structure for the application is defined.

To change the logical navigation structure, derive a new class from **RegionNavigationContentLoader** and override the **GetContractFromNavigationContext** method. In the method, translate the incoming contract name to the view type name to load. It is recommended to call the base class because it conveniently parses the URI into a contract string to inspect. The following code example shows a custom region content loader that maps "Home" to the Home view and "About" to the About view.

Note: This example uses MEF, so export attributes are applied at the top of the class to make it available in the MEF container.

C#

```
[Export(typeof(IRegionNavigationContentLoader))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class CustomRegionNavigationContentLoader : RegionNavigationContentLoader
{
    [ImportingConstructor]
    public CustomRegionNavigationContentLoader(IServiceLocator serviceLocator)
        : base(serviceLocator)
    {
```

```

    }

    protected override string GetContractFromNavigationContext(NavigationContext
navigationContext)
{
    string contract = base.GetContractFromNavigationContext(navigationContext);

    if (contract.Equals("Home", StringComparison.OrdinalIgnoreCase))
    {
        return typeof(HomeView).Name;
    }

    if (contract.Equals("About", StringComparison.OrdinalIgnoreCase))
    {
        return typeof(AboutView).Name;
    }

    return contract;
}
}
}

```

After you have your custom navigation content loader, replace it as the implementation of the **IRegionNavigationContentLoader** in the container. For more information about replacing types in the container, see the section, [Container and Bootstrapper](#), earlier in this topic.

Advanced Navigation Replacements

The following sections describe replacing major portions of the region navigation infrastructure provided by the Prism Library. Most developers will not have scenarios that require this level of customization.

RegionNavigationContentLoader

The **RegionNavigationContentLoader** type implements the **IRegionNavigationContentLoader** interface. You may either derive from **RegionNavigationContentLoader** and override methods, or replace the implementation of the interface entirely.

In addition to the **LoadContent** method, the **RegionNavigationContentLoader** type has two other possible methods to override, but they should be required only in uncommon navigation scenarios:

- **GetCandidatesFromRegion**. This method uses a filter to determine the views in a region that are candidates for handling the navigation request. Applications that need to do special filtering or ordering of candidate views will need to override this method.
- **CreateNewRegionItem**. This method is called to create a view if a candidate is not found that can handle the navigation request. The default implementation uses the **IServiceLocator** to create an instance of the view. Applications that need special logic outside of the container to return instances or singletons of views will need to override this method.

[IRegionNavigationJournal/IRegionNavigationJournalEntry](#)

The region navigation service contains a journal that records the navigation history and provides back and forward navigation. The default implementation is a standard stack implementation. Applications that want to implement more advanced journal and history features (such as the Internet Explorer **Back** button drop-down menu) may need to replace the **RegionNavigationJournal** to change behavior and may need to replace the **RegionNavigationJournalEntry** to provide additional data, such as a Title field and an Icon field, with each entry.

[IRegionNavigationService](#)

The region navigation service provides the core functionality of coordinating the navigation from one view to another. Applications that want to delegate to another navigation system or want to wholesale replace the navigation system in the Prism Library will need to replace the **RegionNavigationService**.

[View Model Locator](#)

The View Model Locator is used in the MVVM Basic QuickStart to wire the view and the view model using its standard convention. This section describes how to change the conventions for naming and locating views, naming, locating and associating view models with views.

For guidance on determining whether to use the View Model Locator or to wire your view and view model together using MEF, see [Implementing the MVVM Pattern](#). As background, the Stock Trader reference implementation uses MEF to wire the view and the view model.

[Changing the View Model Locator Conventions](#)

The **ViewModelLocationProvider** provides a static method called **SetDefault viewTypeToViewModelTypeResolver** that can be used to provide your own convention for associating views to view models.

C#

```
ViewModelLocationProvider.SetDefault viewTypeToViewModelTypeResolver((viewType) =>
{
    ...
    return viewModelType;
});
```

By default, if located in the **View** namespace, the view will update the namespace to **ViewModel** and append the "ViewModel" suffix to the view name. Prism will look for this view model in the same assembly.

[Configuring the ViewModelLocationProvider to Use a Container](#)

The following example shows how to configure the **ViewModelLocationProvider** to construct a view model using a container.

When bootstrapping your application use the **SetDefaultViewModelFactory** method to use your container to resolve view model types. The following is an example using Microsoft's Unity dependency injection container.

C#

```
IUnityContainer _container = new UnityContainer()  
...  
ViewModelLocationProvider.SetDefaultViewModelFactory((t)=> _container.Resolve(t));
```

The default strategy for creating the view models is using the **Activator.CreateInstance** method, which is a valid approach if you have a default constructor in the view model and there are no dependencies to be injected.

16: Code Samples

The code samples for the Prism Library for WPF are focused applications that illustrate specific Prism-related concepts. The QuickStarts and Reference Implementation are an ideal starting point if you want to gain an understanding of a key concept such as Modularity, MVVM, Commands, UI Composition, Navigation, Event Aggregations, User Interactivity, and Composite Application. The Stock Trader Reference Implementation demonstrates proven practices for implementing composite applications. The samples include both source code and documentation.

In order to build and run the samples select the appropriate shortcut file and press F5 to build and run.

Installing Prism

This section describes how to install Prism. It involves the following three steps:

1. Install system requirements.
2. Extract the Prism source code and documentation.
3. Compile and run QuickStarts, Reference Implementation or Prism Library source code.

Step 1: Install System Requirements

Prism was designed to run on the Microsoft Windows 8 desktop, Microsoft Windows 7, Windows Vista, or Windows Server 2008 operating system. WPF applications built using this guidance require the .NET Framework 4.5.

Before you can use the Prism Library, the following must be installed:

- Microsoft .NET Framework 4.5 (installed with Visual Studio 2012) or Microsoft .NET Framework 4.51.
- Microsoft Visual Studio 2012 Professional, Premium, or Ultimate editions or Microsoft Visual Studio 2013 Professional, Premium, or Ultimate editions.

Note: Visual Studio 2013 Express Edition can be used to develop Prism applications using the Prism Library.

Optionally, you should consider also installing the following:

- [Microsoft Blend for Visual Studio 2013](#). A professional design tool for creating compelling user experiences and applications for WPF.

Step 2: Extract the Prism Source Code, and Documentation

To install the Prism assets, right-click the downloaded file, and then click **Run as administrator**. This will extract the source code and documentation into the folder of your choice. You may also need to right-click the file and unblock before you can extract the contents.

Step 3: Compile and run QuickStarts, Reference Implementation or Prism Library source code.

In order to build and run the code sample, select the appropriate shortcut file and press F5 to build and run.

Name	Code sample download from Code Gallery	Category	Summary
<u>Stock Trader Reference Implementation</u>	<u>Download Stock Trader RI code</u>	Prism	<p>The Stock Trader RI application is a reference implementation that illustrates the baseline architecture. Within the application, you will see solutions for common, and recurrent, challenges that developers face when creating composite WPF applications.</p> <p>The Stock Trader RI illustrates a fictitious, but realistic financial investments scenario. Contoso Financial Investments (CFI) is a fictional financial organization that is modeled after real financial organizations. CFI is building a new composite application to be used by their stock traders.</p>
<u>Getting Started Using the Prism Library Hands-on Lab</u>	<u>Download Hello World QuickStart code</u>	Get Started	<p>The Hello World QuickStarts are the ending solution for the Getting Started Using the Prism Library Hands-on Lab. In this lab, you will learn the basic concepts of Prism and apply them to create a Prism Library solution that you can use as the starting point for building a composite WPF.</p>

Name	Code sample download from Code Gallery	Category	Summary
<u>Modularity QuickStarts</u>	<ul style="list-style-type: none"> • Download Modularity QuickStart code for Unity • Download Modularity QuickStart code for MEF 	Modularity	The Modularity QuickStarts demonstrate how to code, discover, and initialize modules using Prism. These QuickStarts represent an application composed of several modules that are discovered and loaded in the different ways supported by the Prism Library using MEF and Unity as the composition containers.
<u>Interactivity QuickStart</u>	Download Interactivity QuickStart code	Interactivity	This QuickStart demonstrates how to create a view and view model that work together when the view model needs to interact with the user or user gesture needs to raise an event that invokes a command. In each of these scenarios the view model should not need to know about the view. The first scenario is handled by using InteractionRequests and InteractionRequestTriggers . The second scenario is handled by InvokeCommandAction .
<u>MVVM QuickStart</u>	Download MVVM QuickStart code	MVVM	The MVVM QuickStart demonstrates how to build a very simple application that implements the MVVM pattern.
<u>Commanding QuickStart</u>	Download Command QuickStart code	Commanding	The Commanding QuickStart demonstrates how to build a WPF UI that uses commands provided by the Prism Library to handle UI actions in a decoupled way.

Name	Code sample download from Code Gallery	Category	Summary
UI Composition QuickStart	Download UI Composition QuickStart code	UI Composition	This QuickStart demonstrates how to build WPF UIs composed of different views that are dynamically loaded into regions and that interact with each other in a decoupled way. It illustrates how to use both the view discovery and view injection approaches for UI composition.
State-Based Navigation QuickStart	State-Based Navigation QuickStart	Navigation	This QuickStart demonstrates an approach to define the navigation of a simple application. The approach used in this QuickStart uses the WPF Visual State Manager (VSM) to define the different states that the application has and defines animations for both the states and the transitions between states.
View-Switching Navigation QuickStart	Download View-Switching Navigation QuickStart code	Navigation	This QuickStart demonstrates how to use the Prism Region Navigation API. The QuickStart shows multiple navigation scenarios, including navigating to a view in a region, navigating to a view in a region contained in another view (nested navigation), navigation journal support, just-in-time view creation, passing contextual information when navigating to a view, views and view models participating in navigation, and using navigation as part of an application built through modularity and UI composition.
Event Aggregation QuickStart	Download Event Aggregation QuickStart code	Event Aggregation	This QuickStart demonstrates how to build a WPF application that uses the Event Aggregator service. This service enables you to establish loosely coupled communications between components in your application.

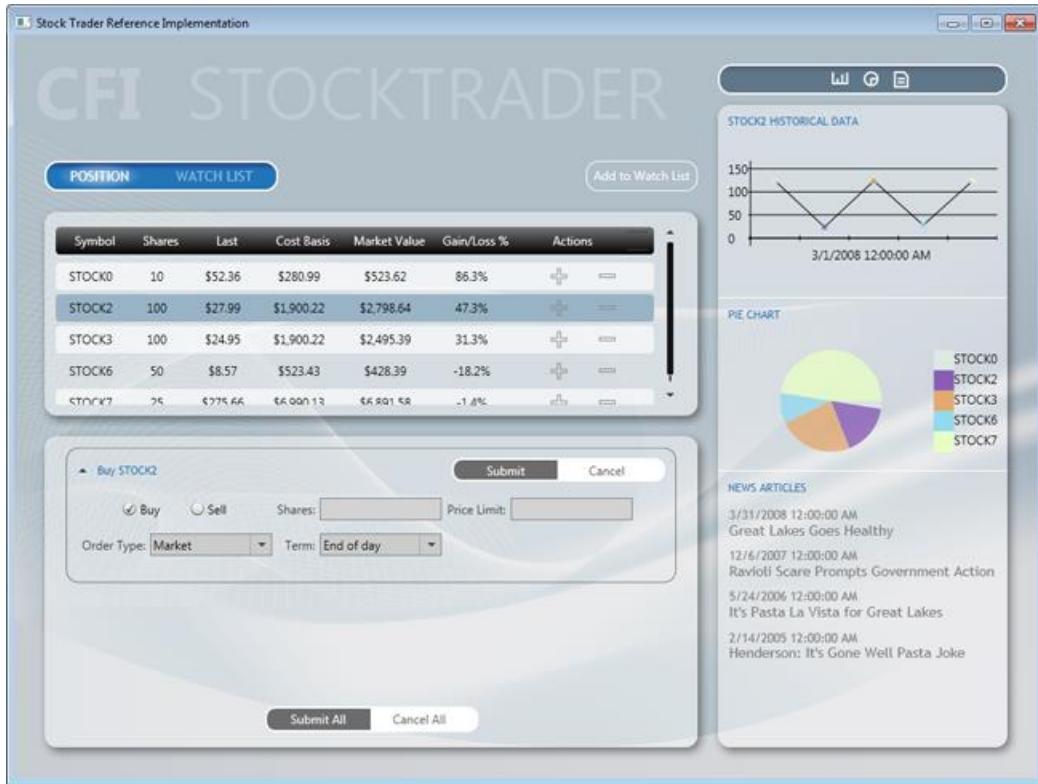
Stock Trader Reference Implementation

Prism includes a sample called a reference implementation, which is a composite application that is based on a real-world scenario. This intentionally incomplete application illustrates the composite application baseline architecture. Within the application, you will see solutions for common, and recurrent, challenges that developers face when creating composite applications. We solve many of the challenges using design patterns such as Model-View-ViewModel (MVVM), Composite View, Event Aggregator, Plug-In, and Dependency Injection that embody important architectural design principles such as separation of concerns and loose coupling. Prism helps you to create a modular application design and build applications using loosely coupled components that can evolve independently but that can be easily and seamlessly integrated into the overall application.

The reference implementation is not a real-world application; however, it is based on real-world challenges customers are facing. When you look at this application, do not look at it as a reference point for building a stock trader application—instead, look at it as a reference for building a composite application.

Note: When looking at this application, it may seem inappropriate to implement it in the way it was implemented. For example, you might question why there are so many modules, and it may seem overly complex. The focus of Prism is to address challenges around building composite applications. For this reason, certain scenarios are used in the reference implementation to emphasize those challenges.

The following illustration shows the desktop version of the Stock Trader Reference Implementation (Stock Trader RI).



Stock Trader RI

You can use the reference implementation in different ways. You can step through a running example that demonstrates application-specific code built on reusable guidance. You can also copy sections of the source code that implement any particular guidance into your own applications.

The reference implementation was developed using a "test driven" approach and includes automated (unit) tests for most of its components. You can modify the reference implementation and use the unit tests to verify its functionality. The reference implementation for the Prism 5.0 release demonstrates several key features of the updated Prism library:

- [Managed Extensibility Framework](#) (MEF) as the dependency injection container
- Modularity and user interface (UI) composition through custom attributes
- Model-View-ViewModel pattern (MVVM)
- Region-based navigation

Building and Running the Reference Implementation

The Stock Trader RI requires Visual Studio 2012 or later and the .NET Framework 4.5.1. The reference implementation is compatible with [Blend for Visual Studio 2013](#).

To run the Stock Trader RI In Windows Explorer, double-click the following shortcut file to open the solution in Visual Studio:

Open RI - StockTrader Reference Implementation.lnk

1. Press F5.
-

Interacting with the Reference Implementation

The features of the Stock Trader reference implementation are covered in greater detail later in the Scenarios section. The following steps provide a quick introduction to the basic features.

To see the pie chart and line chart for each stock

1. Click the **Position** tab.
 2. In the **Position** table, click the row that corresponds to the stock whose pie chart and line chart you want to see.
-

To see a news item corresponding to a stock

1. Click the **Position** tab.
 2. In the **Position** table, click a stock in that corresponds to the stock you want to learn more about.
 3. Click a news article. If you click the control in the upper-right corner, a **News Reader** dialog box opens.
-

To add a stock to the watch list

1. In the **Add to Watch List** box, type the stock symbol for the stock you want to add to the watch list. Valid values include STOCK0 through STOCK9 as the stock symbol.
 2. Press ENTER.
-

To remove a stock from the watch list

1. Click the **Watch List** button.
 2. In the watch list, click the X symbol next to the stock that you want to remove.
-

To buy or sell shares from a stock

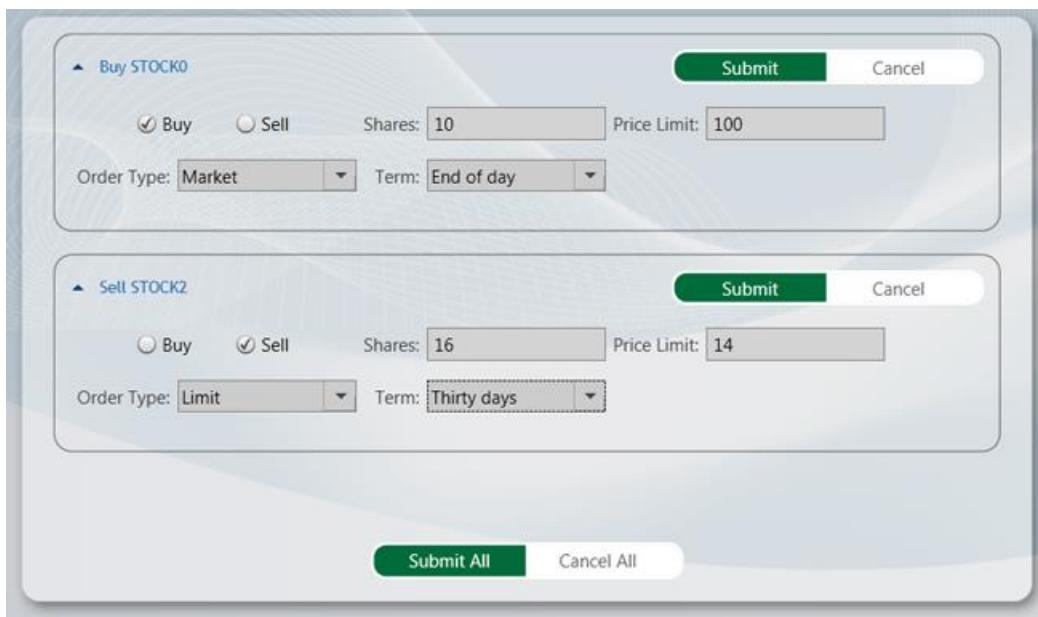
1. In the **Position** area, click the + or – symbol next to the stock that you want to buy or sell.
2. In the **Buy & Sell** area, enter the following data:
 - a. In the **Shares** box, type the number of shares you want to buy or sell.
 - b. In the **Price Limit** box, type the appropriate price.
 - c. In the **Order Type** drop-down box, click **Limit**, **Market**, or **Stop**.

- d. In the **Term** box, click **End of day** or **Thirty days**. Term is the length of time an order will be active before it is carried out or it expires.
3. To submit the order, click the **Submit** button. To cancel the order, click the **Cancel** button.

To submit or cancel all your buy and sell orders

- If you have multiple orders that are ready to be bought or sold, the **Submit All** and **Cancel All** buttons are enabled on the **Buy & Sell** area and on the main task bar. The **Submit All** button will be enabled only if all individual orders are able to be submitted.

The following illustration shows the Stock Trader RI **Buy & Sell** tab.



Buy & Sell area in the Stock Trader RI

Acceptance Tests

The Stock Trader RI includes a separate solution that includes acceptance tests. The acceptance tests describe how the reference implementation should perform when you follow a series of steps. You can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

Outcome

You should see the Stock Trader RI shell window and the tests automatically interact with the application. At the end of the test pass, you should see that all tests have passed.

The Scenario

The Stock Trader RI illustrates a fictitious, but realistic financial investments scenario. Contoso Financial Investments (CFI) is a fictional financial organization that is modeled after real financial organizations. CFI is building a new composite application to be used by their stock traders. This topic contains a

summary of the scenario and demonstrates the business drivers that led to a series of technical decisions that ultimately result in the use of Prism.

Contoso Financial Investments Scenario

Contoso Financial Investments (CFI) is a global investment firm with one hundred traders. Core to doing business in CFI, there is a 15-year-old legacy trader application developed in Visual C++ with the Microsoft Foundational Class Library that, over time, has become increasingly difficult to maintain.

Operating Environment

For the last several years, CFI's lack of maintainability has brought new development on the application to a standstill—this has left the application in maintenance mode. To meet new customer requirements, CFI adopted the Microsoft .NET Framework development platform and branched out, creating additional applications that were each maintained by separate teams in a silo. The idea was that having separately developed applications would actually result in the development effort being more efficient. Each team developing in their own silo meant that CFI could remove any contention that might arise, and it would pave the way for easily creating new teams. This would allow CFI to scale out their development teams into several locations, including setting up several offshore teams.

The harsh reality is that the approach proved to be extremely inefficient on several levels. Because each application was developed in a silo, the trader is now required to maintain multiple copies of the same data throughout a growing suite of applications, including StockPortfolio, MarketView, and StockHist. The data is not identical, but there are elements of the data that are duplicated. To do their jobs, traders constantly jump back and forth between these various applications. To assist with this, CFI employed a "launcher" that quickly launched all the applications from a central place. The launcher also passed the user's logon credentials to the application to skip the logon screen for each application. The launcher is more of a bandage than anything else. It did not greatly improve the overall workflow of the traders in that the applications cannot integrate with one another, nor do they support a consistent UI.

Operational Challenges

Because of the lack of integration, getting a consolidated view of all the related data is not an easy task. There is a customer-facing reporting site that can pull from each of the back-end systems to create this "one" view, but it is littered with problems, the least of which is that if the data has not been properly duplicated, the reports do not work. In addition, entering the duplicate data is extremely time consuming and significantly impacts the number of orders that the trader is processing. Manually entering the data caused many errors in the system. Attempts to automatically synchronize the different systems have been too costly, because the schemas are very different and change frequently. With all these problems, CFI, like many other businesses, has managed to continue to operate as a profitable business. As customer demand has increased, CFI has invested the necessary funds to expand its services. It has also consistently grown its trading force whose jobs have become more and more difficult because of the inefficient operating conditions. Recently, however, this inefficiency has increased to the point that the business is starting to lose money:

- The interaction time per transaction has greatly increased because of the time it takes to navigate the suite of applications.
 - The cost of employee training and in-house support has greatly increased because of the high complexity and lack of consistency of the applications.
 - Maintenance costs of the various applications are extremely prohibitive. For example, in a recent instance, a logic bug that was detected required changes in seven different systems. This critical bug took three weeks to fix because other parts of the system heavily depended on the code where the bug resided. This greatly increased the cost of fixing it, testing it, and deploying it—it brought the total price to \$150,000. This included the effort to fix three additional bugs that were created as part of the original fix.
 - CFI has been unable to keep up with emerging technologies that can offer it a competitive edge and reduced development costs.
-

Emerging Requirements

Currently, CFI is faced with a new challenge around Service-Oriented Architecture (SOA). Fabrikam Web Traders, one of CFI's chief competitors, has offered its customers a rich client desktop experience for managing their portfolios remotely and on-site. The client is able to access Fabrikam's back-end systems through web services. Several large CFI customers are now requesting the same capabilities.

Although there is no immediate threat, in the long term, the business impact can be crippling. If CFI continues with the current strategy and does not both improve its efficiency and adapt to changing market conditions, it will lose business to its competition.

Meeting the Business and IT Objectives

The Chief Executive Officer (CEO) is an opportunist who sees this challenge as an opportunity for CFI to rise to the occasion. Working with the Chief Information Officer (CIO) and Chief Technology Officer (CTO), they devise a three-point strategy for moving CFI forward. The strategy is as follows:

- Reduce the cost of development. To do this, the new system should do the following:
 - It should provide structure for teams to collaborate through a well-defined architecture.
 - It should support distributed teams, including using some offshore developers.
 - It should provide a shorter development life cycle—this improves the time to market.
 - It should present data in ways that were previously prohibitive and time consuming to implement.
 - It should support Test-Driven Development (TDD).
 - It should support automated acceptance tests.
 - It should support integration with third-party systems.

- Improve trader efficiency. To accomplish this, the system should do the following:
 - It should support better multitasking.
 - It should provide a UI that is better adapted to the trader workflow.
 - It should consolidate existing applications.
 - It should provide shorter interaction time per transaction (data visualizations).
 - It should provide better information flow (contextual UI queues).
 - It should provide better use of screen area (also known as screen real estate).
 - It should provide integration among the different components of the system and with external components (services).
 - It should present reduced training time.
 - It should support users whether they are located remotely or are on-site.
 - It should support corporate branding and UI styling.
 - It should minimize the cost of adding new functionality to the system.
 - It should support adding custom extensions provided by either the customer or third-party companies.
- Create a new customer-facing product offering. This offering should do the following:
 - It should include a rich client desktop experience for portfolio management.
 - It should provide UI customization and corporate branding to beat out the competition.
 - It should provide extensibility for third-party vendors.

The CTO has delivered these requirements to the senior architect, who is investigating various options for delivering them.

Development Challenges

For the architect, this project represents one of the most significant changes in the technology environment of CFI. Work will be spread across several software development teams, with additional development being outsourced. In the past, cooperation between the development teams has been limited, and development tended to occur on an ad-hoc basis. This was because he identified the following problems that are a result of current development methodology:

- **Inconsistency.** Similar applications are developed in different ways. This results in higher maintenance and training costs.

- **Varying quality.** Developers with varying levels of experience lack guidance on implementing proven practices. This situation results in inconsistent quality among the applications they produce.
 - **Poor productivity.** In many cases, developers across the company repeatedly solve the same problems in different applications, with little or no reuse of code. Because there was no central design, it was very difficult to get the applications to communicate with one another.
-

The Solution: Prism

The senior architect needs a strategy to realize the architectural vision set forth and to resolve the development challenges identified in the previous section. After significant research, he decides that the best solution can be found in Prism offered by the Microsoft patterns & practices group.

Prism is a set of assets for building complex WPF applications. Prism enables designing a composite application in the following ways:

- It provides infrastructure and support for developing and maintaining WPF composite applications through non-invasive and lightweight APIs.
 - It dynamically composes UI components.
 - It supports application modules that are developed, tested, and deployed by separate teams.
 - It allows incremental adoption.
 - It provides an integrated and consistent user experience.
 - It can be integrated with existing WPF applications.
 - It supports a multi-targeted scenario.
-

Prism from Microsoft patterns & practices meets the requirements of CFI and should allow them to achieve their goals by making development significantly more efficient and predictable. Support for integrating with existing WPF applications is of particular interest to the architect because CFI recently developed several WPF applications to address recent customer needs. He is confident that the guidance will assist him in delivering an effective solution that is robust, reliable, based on proven practices, and that can best use WPF . After presenting his findings to the CTO, the CTO agrees that Prism will help to deliver an effective solution efficiently and cost-effectively. He gives approval for the project to proceed.

Stock Trader RI Features

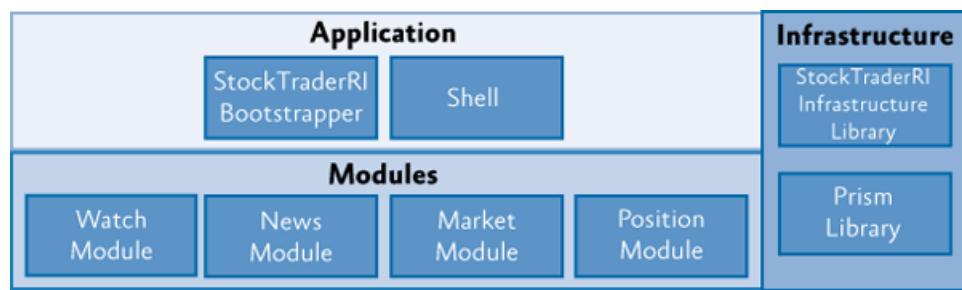
The CFI stock trader application is used for managing a trader's portfolio of investments. Using the stock trader application, traders can see their portfolios, view trend data, buy and sell shares, manage items in their watch lists, and view related news.

The Stock Trader RI supports the following actions:

- See the pie chart and line chart for each stock.
- See a news item that corresponds to a stock.
- Add a stock to the watch list.
- View the watch list.
- Remove a stock from the watch list.
- Buy or sell shares from a stock.
- Submit or cancel your entire buy and sell orders.

Logical Architecture

The following illustration shows a high-level logical architecture view of the Stock Trader RI.



Architectural view of the Stock Trader RI

The Stock Trader RI uses Prism Library for WPF.

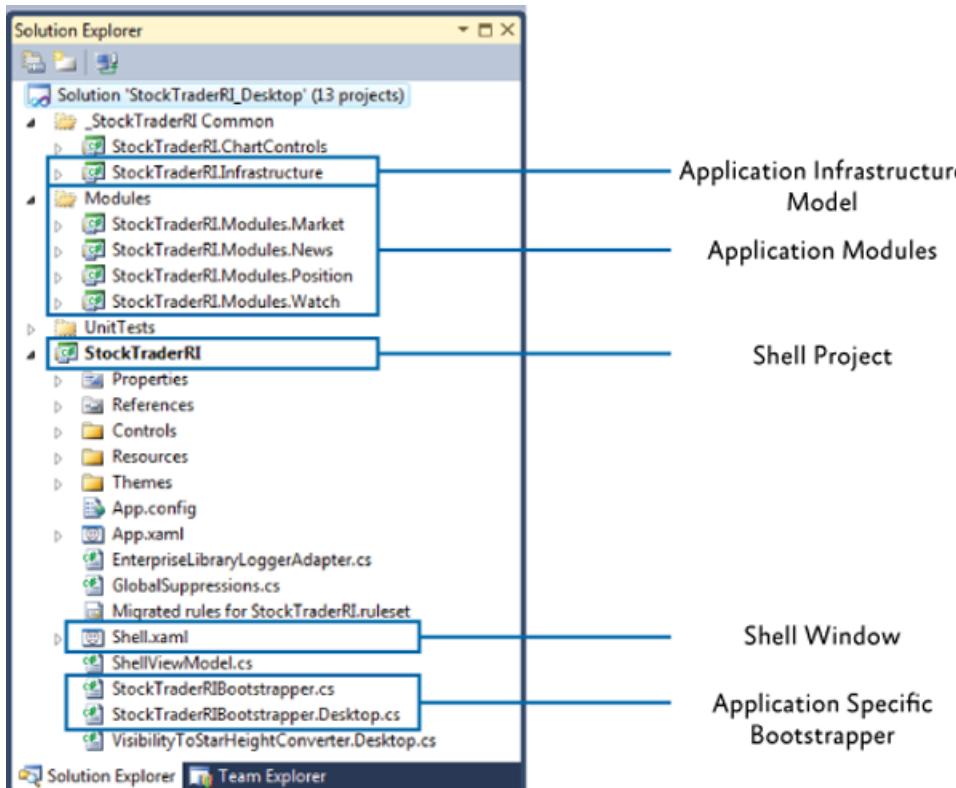
The following describes the main elements of the Stock Trader RI architecture:

- **Application.** The application is lightweight and contains the shell that hosts each of the different UI components within the reference implementation. It also contains the **StockTraderRIBootstrapper**, which sets up the container and initializes module loading.
- **Modules.** The solution is divided into the following four modules, which are each maintained by separate teams in different locations:
 - **Watch module.** The Watch module contains the **Watch List** and **Add To Watch List** functionality.
 - **News module.** The News module contains the **NewsFeedService**, which handles retrieving stock news items.
 - **Market module.** The Market module handles retrieval of market trend data for the trader's positions and notifies the UI when those positions change. It also handles populating the Trend line for the selected position.
 - **Position module.** The Position module handles populating the list of positions in the trader's portfolio. It also contains the Buy/Sell order functionality.

- **Infrastructure.** The infrastructure contains functionality for both the Stock Trader RI and the Prism core:
 - **Prism Library.** This contains the core composition services and service interfaces for handling regions, commanding, and module loading. It also contains the container façade for the Unity Application Block (Unity) and MEF. The **StockTraderRIBootstrapper** inherits from the **MefBootstrapper**.
 - **Stock Trader RI Infrastructure Library.** This contains service interfaces specific to the Stock Trader RI, shared models, and shared commands.

Implementation View

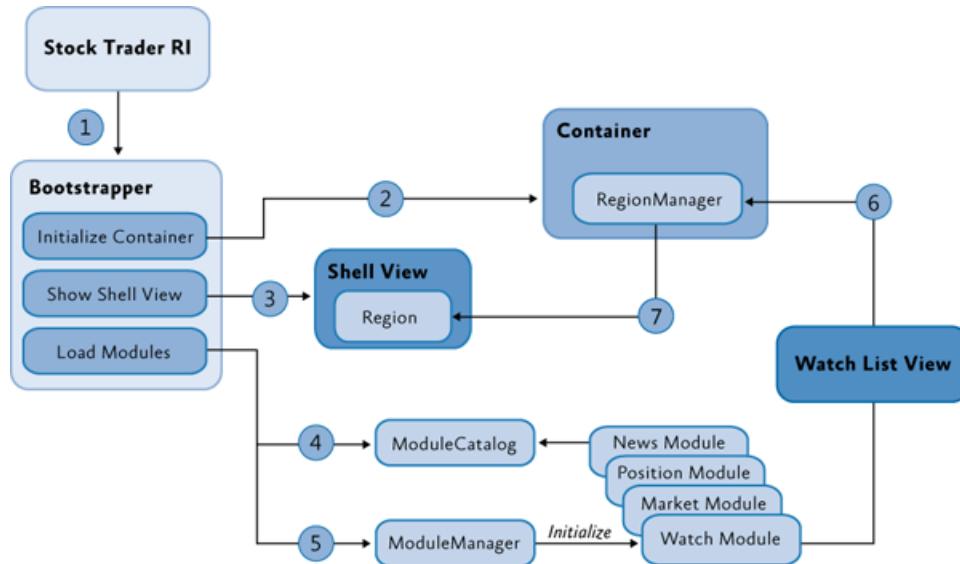
The Stock Trader RI is based on the Prism Library. The following illustration shows the Stock Trader RI (Desktop version) Solution Explorer.



Stock Trader RI solution view

How the Stock Trader RI Works

The Stock Trader RI is a composite application, which is composed of a set of modules that are initialized at run time. The following illustration shows the application's startup process, which includes the initialization of modules. The next sections provide details about each of these steps.



Stock Trader RI startup process

The Stock Trader RI startup process is the following:

1. The application uses the **StockTraderRIBootstrapper**, which inherits from the Prism Library's **MefBootstrapper** for its initialization.
2. The application initializes the Prism Library's **MefServiceLocatorAdapter** for use in the modules.
3. The **StockTraderRIBootstrapper** creates and shows the shell view.
4. The Prism Library's **ModuleCatalog** finds all the modules the application needs to load.
5. The Prism Library's **ModuleManager** loads and initializes each module.
6. Modules use the Prism Library's **RegionManager** service to add a view to a region.
7. The Prism Library's **Region** displays the view.

Modules

A module is a logical unit of separation in the application. In the Stock Trader RI, each module exists in a separate assembly, but this is not an absolute requirement. The advantage of having this separation is that it makes the application more maintainable and enables distributed teams to work on different modules with minimal overlap on the files being updated in the source control system.

The application does not directly insert views from each module into the shell; instead, each module contributes content to the shell view and interacts with other modules. The final system is composed of the aggregation of the modules' contributions. By using composition, you can create applications with emergent behaviors—this refers to the application being able to scale up in complexity and requirements as it grows.

The modules are loosely coupled. This means they do not directly reference each other, which promotes separation of concerns and allows modules to be individually developed, tested, and deployed by different teams.

Services and Containers

This is possible through a set of application services that the modules have access to. Modules do not directly reference one another to access these services. In the Stock Trader RI, a dependency injection container (referred to as the container) injects these services into modules during their initialization (the Stock Trader RI uses the MEF container).

Note: For an introduction to dependency injection and Inversion of Control, see the article, [Loosen Up - Tame Your Software Dependencies for More Flexible Apps](#), by James Kovacs in *MSDN Magazine*.

Bootstrapping the Application

Modules get initialized during a bootstrapping process by a class named **MefBootstrapper**. The **MefBootstrapper** is responsible for starting the core composition services used in an application created with the Prism Library. The following code from the **MefBootstrapper** class shows how the Module Manager is located from the container.

C#

```
// MefBootstrapper.cs
protected override void InitializeModules()
{
    IModuleManager manager = this.Container.GetExportedValue<IModuleManager>();

    manager.Run();
}
```

The Module Manager manages the process of validating the module catalog, retrieving modules if they are remote, loading the modules into the application domain, and calling the **IModule.Initialize** method.

Configuring the Aggregate Catalog

The **StockTraderRIBootstrapper** class configures the **AggregateCatalog** in code. In this case, the shell has direct references to all the modules, so the **StockTraderRIBootstrapper** can directly add them to the **AggregateCatalog**. The **StockTraderRIBootstrapper** also adds its own assembly to the catalog so that types exported within the application are available in the container.

C#

```
// StockTraderRIBootstrapper.cs
protected override void ConfigureAggregateCatalog()
{
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(StockTraderRIBootstrapper).Assembly));
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(StockTraderRICommands).Assembly));
```

```

    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(MarketModule).Assembly));
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(PositionModule).Assembly));
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(WatchModule).Assembly));
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(NewsModule).Assembly));
}

```

Module Loading

After the container is populated, the types contained in each module assembly are available.

Note: Each module class (for example, **NewsModule**) in the reference implementation is empty. The use of MEF allows for discovery of types using declarative attributes, so there is not any work to be done during module initialization. If a module needed to do additional work when it is loaded, the module class should then implement **IModule** and perform this initialization in the **Initialize** method. The **ModuleManager** would then discover, load, and initialize that module.

During this initialization process, the container will inject instances into types to resolve their dependencies. The following code shows how the news feed service, region manager, and event aggregator services are injected into the **ArticleViewModel** constructor.

C#

```

// ArticleViewModel.cs
[Export(typeof(ArticleViewModel))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class ArticleViewModel : BindableBase
{
    [ImportingConstructor]
    public ArticleViewModel(INewsFeedService newsFeedService, IRegionManager
regionManager, IEventAggregator eventAggregator)
    {
        ...
    }
}

```

In addition, other types, such as services, are available so they can be accessed either by the same module or other modules in a loosely coupled fashion.

Views

A view is any content that a module contributes to the UI. In the Stock Trader RI, views are discovered at run time and added to regions. Regions are classes associated with a control container, such as **ContentControl** or **TabControl**.

Note: In the Stock Trader RI, views are usually user controls. However, data templates in WPF are an alternative approach to rendering a view.

View Registration in the Container

Views can be registered through declarative attributes, directly in code, or through configuration. The Stock Trader RI uses MEF and the MVVM pattern to demonstrate the use of declarative attributes.

Views associate themselves with a region through a custom export attribute, as shown in the following code example.

C#

```
// ArticleViewModel.cs
[ViewExport(RegionName = RegionNames.ResearchRegion)]
[PartCreationPolicy(CreationPolicy.Shared)]
public partial class ArticleView : UserControl
```

The **AutoPopulateExportedViewsBehavior** in the Stock Trader RI infrastructure discovers the views in the container and automatically populates them into the associated region, as shown in the following code example.

C#

```
// AutoPopulateExportedViewsBehavior.cs
[ImportMany(AllowRecomposition = true)]
public Lazy<object, IViewRegionRegistration>[] RegisteredViews { get; set; }

public void OnImportsSatisfied()
{
    AddRegisteredViews();
}

private void AddRegisteredViews()
{
    if (this.Region != null)
    {
        foreach (var viewEntry in this.RegisteredViews)
        {
            if (viewEntry.Metadata.RegionName == this.Region.Name)
            {
                var view = viewEntry.Value;

                if (!this.Region.Views.Contains(view))
                {
                    this.Region.Add(view);
                }
            }
        }
    }
}
```

Model-View-ViewModel

The Stock Trader RI uses the MVVM pattern to separate UI, presentation logic, and the data model. Using MVVM allows the view model to be unit tested because it has no direct knowledge of the view.

The Prism Library provides the **BindableBase** class that the view models in the Stock Trader RI use to notify the user interface of property changes. **BindableBase** makes implementing **INotifyPropertyChanged** much easier.

In the Stock Trader RI, the view and view model are connected through view discovery. The view is discovered by the **AutoPopulateExportedViewsBehavior** and instantiated through the container. Because the view declares an import of the view model, the container then instantiates the view model and injects it into the view, as shown in the following code example.

C#

```
// ArticleView.xaml.cs
[Import]
ArticleViewModel ViewModel
{
    set
    {
        this.DataContext = value;
    }
}
```

For more information about view discovery, see [Composing the User Interface](#).

Commands

Views can communicate with presenters and services in a loosely coupled fashion by using commands. The **Add To Watch List** control, as shown in the following illustration, uses the **AddWatchCommand**, which is a **DelegateCommand**, to notify the **WatchListService** whenever a new watch item is added.

Note: The **DelegateCommand** is one kind of command that the Prism Library provides. For more information about commands in Prism, see "[Commands](#)" in [Implementing the MVVM Pattern](#).



Add To Watch List control

Using a **DelegateCommand** allows the service to delegate the command's **Execute** method to the service's **AddWatch** method, as shown in the following code example.

C#

```
// WatchListService.cs
```

```

public WatchListService(IMarketFeedService marketFeedService)
{
    ...
    AddWatchCommand = new DelegateCommand<string>(AddWatch);
    ...
}

private void AddWatch(string tickerSymbol)
{
    ...
}

```

The **WatchListService** is also injected into the **AddWatchViewModel**, which exposes the command to the view.

C#

```

// AddWatchViewModel.cs
public class AddWatchViewModel : BindableBase
{
    private string stockSymbol;
    private IWatchListService watchListService;

    [ImportingConstructor]
    public AddWatchViewModel(IWatchListService watchListService)
    {
        if (watchListService == null)
        {
            throw new ArgumentNullException("service");
        }
        this.watchListService = watchListService;
    }

    public string StockSymbol
    {
        get { return stockSymbol; }
        set
        {
            SetProperty(ref stockSymbol, value);
        }
    }

    public ICommand AddWatchCommand { get { return
this.watchListService.AddWatchCommand; } }
}

```

The **AddWatchButton** in the view then binds to the **AddWatchViewModel** command through the **DataContext**.

XAML

```
<!--AddWatchView.xaml -->
<StackPanel Orientation="Horizontal">
    <TextBox Name="AddWatchTextBox" MinWidth="100" Style="{StaticResource CustomTextBoxStyle}"
        Infrastructure:ReturnKey.Command="{Binding Path=AddWatchCommand}"
        Infrastructure:ReturnKey.DefaultTextAfterCommandExecution="Add to Watch List"
        Text="Add to Watch List"
        AutomationProperties.AutomationId="TextBoxBlock" Margin="5,0,0,0"/>
</StackPanel>
```

This is using an attached behavior on the **Add To Watch List** text box, so when the user enters a stock symbol and then presses ENTER, the **AddWatchCommand** will be invoked, thereby passing the stock symbol to the **WatchListService**.

Event Aggregator

The Event Aggregator pattern channels events from multiple objects through a single object to simplify registration for clients. In the Prism Library, a variation of the Event Aggregator pattern allows multiple objects to locate and publish or subscribe to events.

In the Stock Trader RI, the event aggregator is used to communicate between modules. The subscriber tells the event aggregator to receive notifications on the UI thread. For example, when the user selects a symbol on the **Position** tab, the **PositionSummaryViewModel** in the Position module raises an event that specifies the symbol that was selected, as shown in the following code example.

C#

```
// PositionSummaryViewModel.cs
eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Publish(CurrentPositionSummaryItem.TickerSymbol);
```

The **ArticleViewModel** in the News module listens to the event to display the news related to the selected symbol, as shown in the following code example.

C#

```
// ArticleViewModel.cs
eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(OnTickerSymbolSelected, ThreadOption.UIThread);
```

Note: The notification of the event is on the UI thread to safely update the UI and avoid a WPF exception.

Technical Challenges

The Stock Trader Reference Implementation (Stock Trader RI) demonstrates how you can address common technical challenges that you face when you build composite applications in WPF. The following table describes the technical challenges that the Stock Trader RI addresses.

Technical challenge	Feature in the Stock Trader RI	Example of where feature is demonstrated
Views and composite UI		
Regions: The use of regions for placing the views without having to know how the layout is implemented.	Regions defined in the shell and position module's orders view.	StockTraderRI\Shell.xaml StockTraderRI.Modules.Position\Orders\OrdersView.xaml
Composite view: Shows how a composite view communicates with its child view.	Order screen	StockTraderRI.Modules.Position\Orders\OrderCompositeViewModel.cs StockTraderRI.Modules.Position\Orders\OrderDetailsViewModel.cs StockTraderRI.Modules.Position\Orders\OrderCommandsView.xaml.cs
Compose UI across modules: Shows how a module can have views in different parts of the shell that interact with each other.	The Watch module has a view and also is a part of the toolbar.	StockTraderRI.Modules.Watch\AddWatch\AddWatchView.xaml StockTraderRI.Modules.Watch\WatchList\WatchListView.xaml
	The News module has an article list view and a popup article reader view that shows the same articles.	StockTraderRI.Modules.News\Article\ArticleView.xaml StockTraderRI.Modules.News\Article\NewsReader.xaml
Decoupled communication		
Commands: Shows the Command pattern. The command to buy or sell a stock is a delegate command. Each row in the list uses the same command instance but with a different parameter corresponding to the stock. This decouples the invoker from the receiver and shows passing additional data with the command.	Buy and Sell command invokers in PositionSummaryView and handlers in OrdersController	StockTraderRI.Modules.Position\Controllers\OrdersController.cs StockTraderRI.Modules.Position\PositionSummary\PositionSummaryView.xaml
Composite commands: Use composite commands to broadcast all of the	Submit All and Cancel All buttons	StockTraderRI.Infrastructure\StockTraderRICommands.cs StockTraderRI.Modules.Position\Orders\OrderDetails.cs

commands. The Submit All or Cancel All commands execute all the individual instances of the Submit or Cancel commands.		IsViewModel.cs StockTraderRI.Modules.Position\Controllers\OrdersController.cs
Event Aggregator pattern: Publish and Subscribe to events across decoupled modules. Publisher and Subscriber have no contract other than the event type.	Show relevant news content: When the user selects a position in the position list, the communication to the news module uses the EventAggregator service.	StockTraderRI.Modules.Position\PositionSummary\PositionSummaryPresentationModel.cs StockTraderRI.Modules.News\Controllers\NewsController.cs
	Market feed updates: The consumers of the market feed service subscribe to an event to be notified when new feeds are available; the consumers then update the model behind the UI.	StockTraderRI.Modules.Market\Services\MarketFeedService.cs StockTraderRI.Modules.Position\PositionSummary\ObservablePosition.cs StockTraderRI.Modules.Watch\WatchList\WatchListViewModel.cs
Services: Services are also used to communicate between modules. Services are more contractual and flexible than commands.	Several service implementations in module assemblies	Services: StockTraderRI.Modules.Market\Services\MarketFeedService.cs StockTraderRI.Modules.Market\Services\MarketHistoryService.cs StockTraderRI.Modules.News\Services\NewsFeedService.cs StockTraderRI.Modules.Watch\Services\WatchListService.cs StockTraderRI.Modules.Position\Services\AccountPositionService.cs StockTraderRI.Modules.Position\Services\XmlOrdersService.cs
Other technical challenges		
WPF: Use WPF for the user interface	Shell and module views	The starting point for Stock Trader RI - Desktop version is in the StockTraderRI\App.xaml.cs
Bootstrapper: The use of a bootstrapper to initialize the application with global services.	Created bootstrapper with MEF and configuring global services, such as logging and defining the module catalog.	Bootstrapper: StockTraderRI\StockTraderRIBootstrapper.cs

Unit and Acceptance Tests

The Stock Trader RI includes unit tests within the solution. Unit tests verify whether individual units of source code work as expected.

To run the Stock Trader RI unit tests

- On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.
-

The Stock Trader RI includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the Stock Trader RI acceptance tests

1. In Visual Studio, open the solution file StockTraderRI\StockTraderRI.Tests.AcceptanceTest\StockTraderRI.Tests.AcceptanceTest.sln.
 2. Build the solution.
 3. Open Test Explorer.
 4. After building the solution, the test will be found. Click the **Run All** button to run the acceptance tests.
-

Outcome

You should see the reference implementation window and the tests automatically interact with the application. At the end of the test run, you should see that all tests have passed.

Modularity QuickStarts

Modulatiry QuickStarts source code:

- Download [Unity version](#)
 - Download [MEF version](#)
-

The QuickStarts included in this topic provide code samples that demonstrate how to create a modular WPF application using the Prism library. The samples demonstrate how to code, discover, and initialize modules:

- **Creating modules.** Modules are classes that implement the **IModule** interface. Declarative attributes can be used to name modules, control initialization, and define dependencies.
- **Registering modules.** Modules can be registered in the following ways:
 - **Directly in code.** Modules can be directly registered in the module catalog in the application code. Using this approach, you can use conditional logic to determine which module should be included in your application. Modules added in code are referenced by the application instead of being loaded at run time.
 - **Using configuration.** Prism can register modules with the module catalog by loading a configuration file. Declaring the modules in configuration allows the modules to be loaded and initialized independent of the application.
 - **Using directory inspection.** A directory can be specified and inspected to load assemblies in the directory and discover modules.
- **Registering module dependencies.** Modules can have dependencies on other modules. Prism provides dependencies management, including cyclic dependencies and duplicate module detection.
- **Initializing modules.** Prism supports the following two initialization modes:
 - **When available.** Modules can be initialized as soon as they are available. Modules downloaded with the application are initialized during startup. Modules set to download in the background are initialized immediately after downloading completes.
 - **On-demand.** Modules can be initialized when the application code requests it. Modules downloaded in the background start downloading when the application requests the module, and then they initialize immediately after downloading completes.

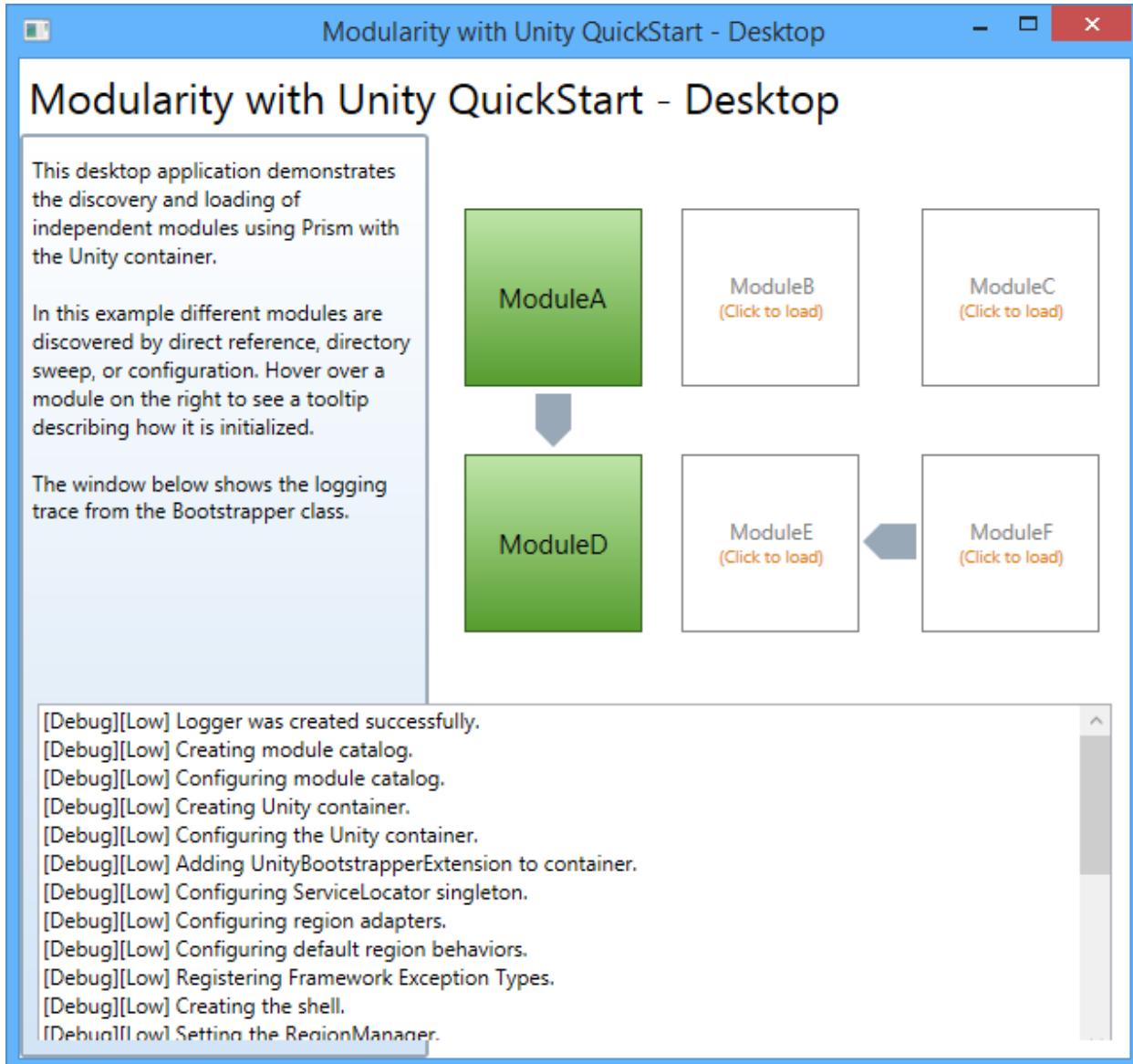
- **Downloading modules in the background.** Although downloading in the background is most useful to Silverlight applications, desktop applications can now take advantage of the same progress and completion events as assemblies are discovered and loaded:
 - **Displaying progress.** An application can subscribe to a progress-changed event to display byte count and percentage progress as modules are downloaded.
 - **Action on download complete.** An application can subscribe to a load module-completed event to take additional action after a module loads and initializes.
- **Leveraging different dependency injection containers.** Traditionally, the QuickStarts have demonstrated using the Unity container, while the core library code remained container-agnostic. With the addition of Managed Extensibility Framework (MEF) to the .NET Framework 4, there are two QuickStarts, each of which uses a different container:
 - **ModularityWithMef.** This QuickStart demonstrates modularity when using the MEF as the dependency injection container. Prism leverages MEF's declarative attribute model to integrate the **ModuleCatalog** and MEF's **ComposablePartsCatalog**.
 - **ModularityWithUnity.** This QuickStart demonstrates modularity when using Unity as the dependency injection container.

Scenarios

This section describes the scenarios included in both modularity QuickStarts. Each QuickStart is composed of six modules: ModuleA, ModuleB, ModuleC, ModuleD, ModuleE, and ModuleF. Each module demonstrates an aspect of how modules are discovered, downloaded, and initialized.

Module	Defined in	Initialized	Downloaded	Depends on
A	Code	When available	With application	D
B	Directory	On demand	In background	
C	Code	On demand	With application	
D	Directory	When available	In background	
E	Configuration	On demand	In background	
F	Configuration	On demand	In background	E

Each QuickStart displays each module as a control. The module control indicates whether it has been initialized, displays downloading progress, and on-demand modules can be clicked to request initialization. The control also provides a tooltip that shows its current initialization state and discovery information. At the bottom of each QuickStart page is a text box that displays the log entries from the bootstrapping sequence and module-loading details. The following illustration shows the main page of the Modularity with MEF QuickStart.



Modularity QuickStart user interface

Building and Running the QuickStarts

This QuickStart requires Microsoft Visual Studio 2012 or later with .NET Framework 4.5.1.

To build and run the ModularityWithMef QuickStart

1. In Visual Studio, open the solution file
Quickstarts\Modularity\Desktop\ModularityWithMef\ModularityWithMef.Desktop.sln.
2. In the **Build** menu, click **Rebuild Solution**.
3. Press F5 to run the QuickStart.

To build and run the ModularityWithUnity QuickStart

1. In Visual Studio, open the +solution file
Quickstarts\Modularity\Desktop\ModularityWithUnity\ModularityWithUnity.Desktop.sln.
2. In the **Build** menu, click **Rebuild Solution**.
3. Press F5 to run the QuickStart.

Note: Both QuickStarts have post-build events configured on each module project to automatically store the modules' assemblies in a folder after a successful build. Modules B and D are copied into a DirectoryModules folder and Modules E and F are copied into the same location as the application executable.

To see the post-build events configuration, right-click a module project, and then click **Properties**. In the **Properties** dialog box, click the **Build Events** tab.

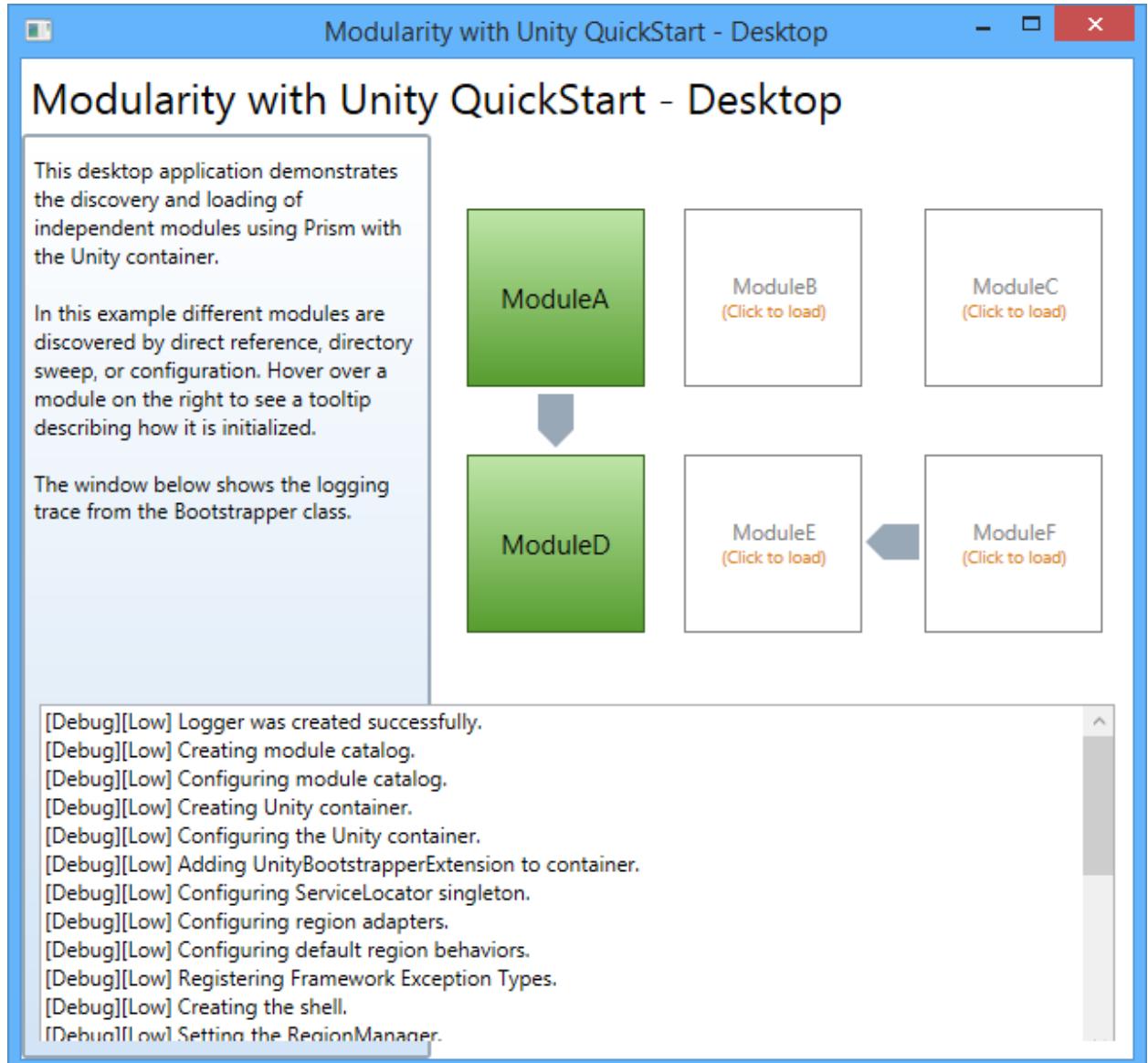
The following code shows the post-build event command in the **Post-build event command line** text box.

```
xcopy "$(TargetDir)*.*"  
"$(SolutionDir)ConfigurationModularity\bin\$(ConfigurationName)\DirectoryModules\" /Y
```

Walkthrough

To explore the scenario, perform the steps to build and run the QuickStart:

1. The main window shows a set of modules, each of which displays the module's initialization state, as shown in the following illustration. As the application starts, Module D and Module A are discovered and initialized.

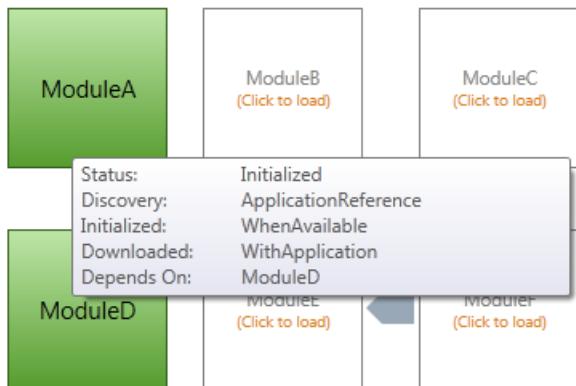


Main page of Modularity with MEF QuickStart

Module D is discovered by directory inspection at application startup. Module A is initialized when it is available and depends on Module D. After Module D loads, Module A is initialized. The trace window at the bottom shows messages as the application is initialized.

Note: If no dependencies are specified, the module load order is non-deterministic.

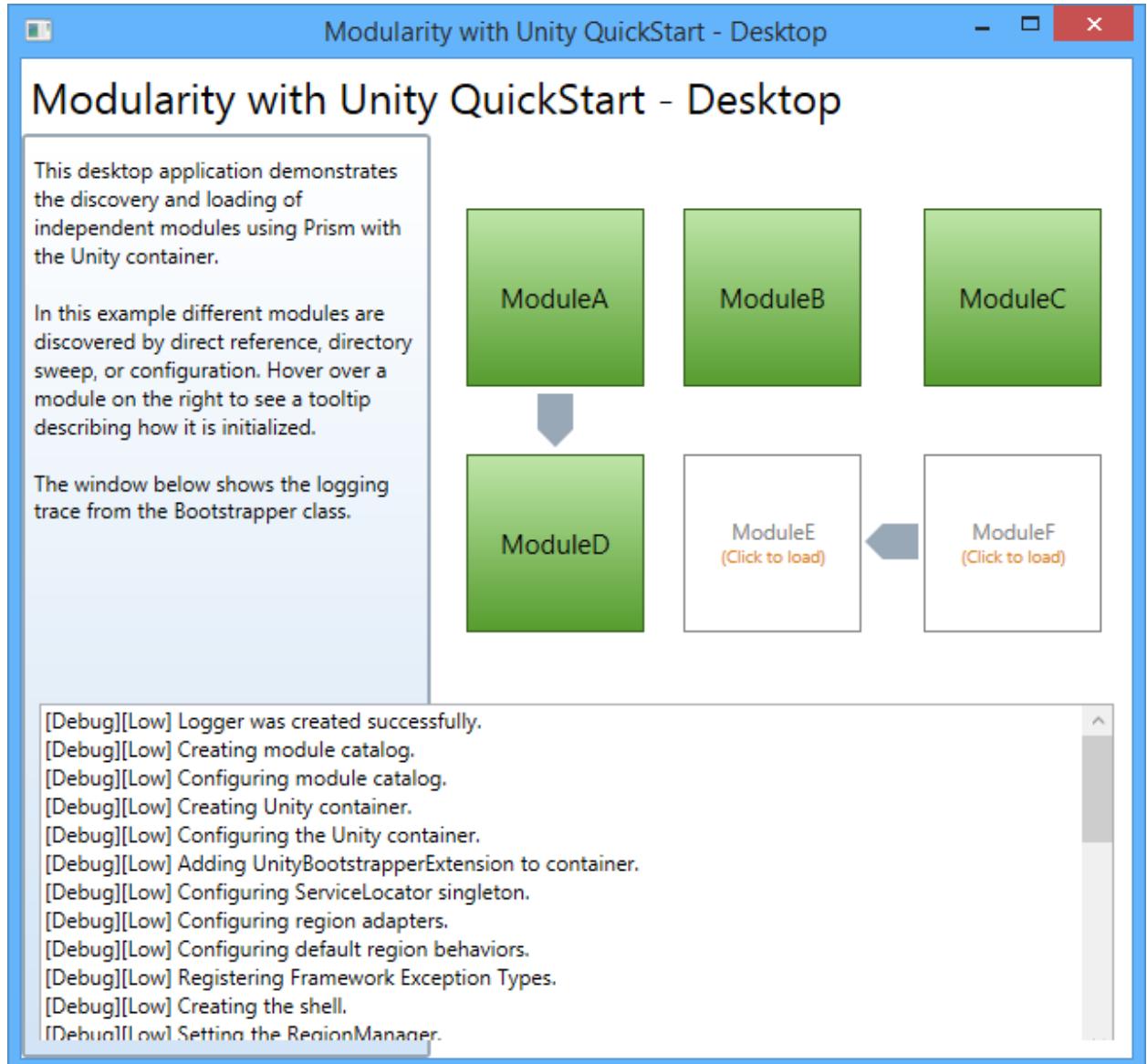
2. Hover over the **Module A** control. When the mouse hovers over the **Module A** control, a descriptive tooltip is displayed, as shown in the following illustration.



Module Information tooltip

As you hover the pointer over a module, a tooltip displays that shows information about its status, discovery, initialization, download timing, and dependencies.

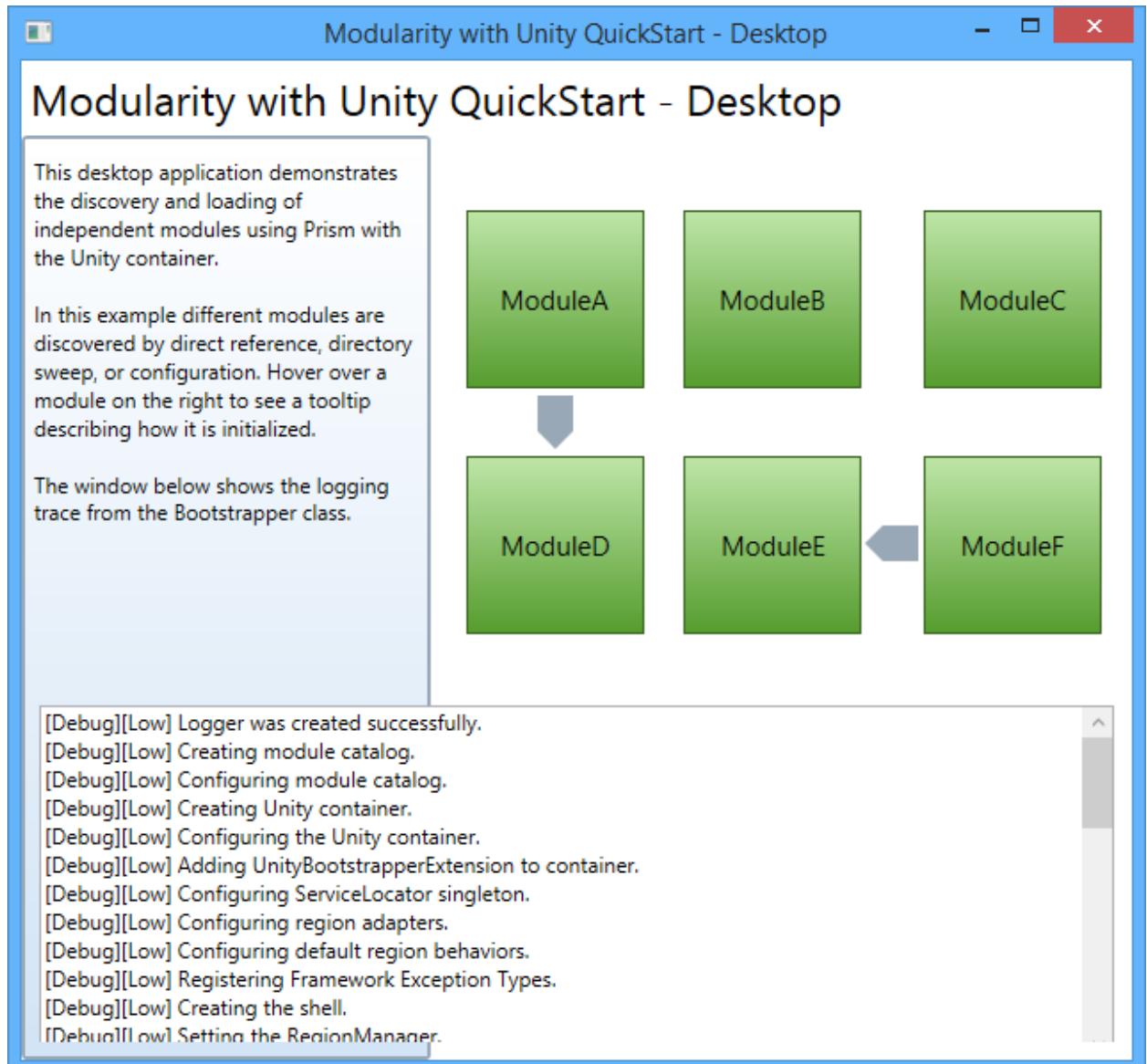
3. Click the **Module B** and **Module C** controls. As each module initialization state changes, the visual control is updated. When either the **Module B** control or the **Module C** control is clicked, that module gets loaded, as shown in the following illustration.



Screen shot of module loaded

Module B is discovered by directory inspection, and Module C is referenced by the application. Both modules are loaded on demand.

4. Click the **Module F** control. When the **Module F** control is clicked, Module E and Module F get loaded, as shown in the following illustration.



Screen shot of Module E getting loaded to load Module F

Notice that Module F completes its initialization first, but is not initialized until after Module E initializes because of the dependency.

Implementation Details

The QuickStarts highlight the key components in modularity. The following sections describe the key artifacts the QuickStarts.

The **Bootstrapper** overrides several methods from either the **MefBootstrapper** or the **UnityBootstrapper** to support the specifics of the application. These sections describe important differences between dependency injection containers.

Application Startup

The application uses the **QuickStartBootstrapper** to start the application and initialize the main window.

C#

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);

    // The bootstrapper will create the Shell instance, so the App.xaml
    // does not have a StartupUri.
    QuickStartBootstrapper bootstrapper = new QuickStartBootstrapper();
    bootstrapper.Run();
}
```

The **Bootstrapper** overrides **CreateShell** and **InitializeShell** methods to create and show the main window.

C#

```
protected override DependencyObject CreateShell()
{
    return ServiceLocator.Current.GetInstance<Shell>();
}

protected override void InitializeShell()
{
    base.InitializeShell();

    Application.Current.MainWindow = (Window)this.Shell;
    Application.Current.MainWindow.Show();
}
```

Creating Modules

In this QuickStart, six modules are created by implementing the **IModule** interface. Attributes are added, depending on the dependency injection container chosen (that is, Unity or MEF).

C#

```
// when using Unity
[Module(ModuleName = WellKnownModuleNames.MyModule)]
[ModuleDependency(WellKnownModuleNames.DependentModule)]
public class MyModule: IModule
{
    ...
}
```

When using Unity, attributes can be used to name the module and specify dependencies.

C#

```
// when using MEF
[ModuleExport(typeof(ModuleA), DependsOnModuleNames = new string[] { "ModuleD" })]
public class ModuleA : IModule
{
    ...
}
```

When using MEF, the **ModuleExport** attribute allows MEF to discover the appropriate type deriving from the **IModule** interface; in addition, it provides the ability to specify additional module metadata.

Registering Modules

In this QuickStart, some modules are directly referenced by the application, discovered by inspecting a directory, or registered by loading a configuration file.

The **QuickStartBootstrapper** overrides **CreateModuleCatalog** and **ConfigureModuleCatalog** methods to register modules.

C#

```
// when using Unity
protected override IModuleCatalog CreateModuleCatalog()
{
    return new AggregateModuleCatalog();
}
```

C#

```
// when using Unity
protected override void ConfigureModuleCatalog()
{
    // Module A is defined in the code.
    Type moduleAType = typeof(ModuleA);
    ModuleCatalog.AddModule(new ModuleInfo(moduleAType.Name,
moduleAType.AssemblyQualifiedName));

    // Module C is defined in the code.
    Type moduleCType = typeof(ModuleC);
    ModuleCatalog.AddModule(new ModuleInfo()
    {
        ModuleName = moduleCType.Name,
        ModuleType = moduleCType.AssemblyQualifiedName,
        InitializationMode = InitializationMode.OnDemand
    });

    // Module B and Module D are copied to a directory as part of a post-build step.
    // These modules are not referenced in the project and are discovered by
    // inspecting a directory.
    // Both projects have a post-build step to copy themselves into that directory.
```

```

    DirectoryModuleCatalog directoryCatalog = new DirectoryModuleCatalog() {
ModulePath = @".\DirectoryModules" };
    ((AggregateModuleCatalog)ModuleCatalog).AddCatalog(directoryCatalog);
    // Module E and Module F are defined in configuration.
    ConfigurationModuleCatalog configurationCatalog = new
ConfigurationModuleCatalog();
    ((AggregateModuleCatalog)ModuleCatalog).AddCatalog(configurationCatalog);

}

```

Note: To demonstrate multiple ways of using the **ModuleCatalog**, the QuickStart using Unity implements an **AggregateModuleCatalog** that derives from **IModuleCatalog**. This class is not intended to be used in a shipping application.

When using MEF, the **AggregateCatalog** provides module and type discovery. In this case, the **QuickStartBootstrapper** overrides the **ConfigureAggregateCatalog** template method and registers assemblies with MEF. The **ModuleCatalog** is still used for registering modules by loading a configuration file.

C#

```

// when using MEF
protected override IModuleCatalog CreateModuleCatalog()
{
    // When using MEF, the existing Prism ModuleCatalog is still the place to
    // configure modules via configuration files.
    return new ConfigurationModuleCatalog();
}

```

C#

```

// when using MEF
protected override void ConfigureAggregateCatalog()
{
    base.ConfigureAggregateCatalog();

    // Add this assembly to export ModuleTracker.
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(QuickStartBootstrapper).Assembly));
    // Module A is referenced in in the project and directly in code.
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(ModuleA).Assembly));
    this.AggregateCatalog.Catalogs.Add(new
AssemblyCatalog(typeof(ModuleC).Assembly));
    // Module B and Module D are copied to a directory as part of a post-build step.
    // These modules are not referenced in the project and are discovered by
    // inspecting a directory.
    // Both projects have a post-build step to copy themselves into that directory.
    DirectoryCatalog catalog = new DirectoryCatalog("DirectoryModules");
}

```

```

    this.AggregateCatalog.Catalogs.Add(catalog);
}

```

Loading Modules

This QuickStart demonstrates both loading modules at startup and on demand, displaying progress, and handling dependencies between modules.

Note: This QuickStart has additional classes that help to track module initialization state. These classes are for demonstration purposes only and are not intended for shipping applications.

The Shell user interface contains a **ModuleControl** for each module. The Shell also has the **ModuleTracker** class as its **DataContext**.

The **ModuleTracker** contains a **ModuleTrackingState** for each module. **ModuleControl** data binds to **ModuleTrackingState** and uses a custom style to visually display the downloading and initialized state of the module.

The Shell responds to a request from the user interface (UI) to load a module and call the **ModuleManager.LoadModule** method.

C#

```

private void ModuleC_RequestModuleLoad(object sender, EventArgs e)
{
    // The ModuleManager uses the Async Events pattern.
    this.moduleManager.LoadModule(WellKnownModuleNames.ModuleC);
}

```

The Shell is notified of download progress by subscribing to the **ModuleManager.ModuleDownloadProgressChanged** event.

C#

```

this.moduleManager.ModuleDownloadProgressChanged +=
this.ModuleManager_ModuleDownloadProgressChanged;

```

C#

```

void ModuleManager_ModuleDownloadProgressChanged(object sender,
    ModuleDownloadProgressChangedEventArgs e)
{
    this.moduleTracker.RecordModuleLoading(e.ModuleInfo.ModuleName, e.BytesReceived,
        e.TotalBytesToReceive);
}

```

The Shell is notified when the module is downloaded and initialized by subscribing to the **ModuleManager.LoadModuleCompleted** event.

C#

```
this.moduleManager.LoadModuleCompleted += this.ModuleManager_LoadModuleCompleted;
```

C#

```
void ModuleManager_LoadModuleCompleted(object sender, LoadModuleCompletedEventArgs e)
{
    this.moduleTracker.RecordModuleLoaded(e.ModuleInfo.ModuleName);
}
```

Key Modularity Classes

The following are some key classes used in the modularity QuickStarts:

- **ModuleCatalog**. This class is responsible for cataloging the metadata for modules and module groups in the application.
- **ModuleManager**. This class coordinates the initialization of the modules. It manages the retrieval and the subsequent initialization of the modules.
- **ModuleInitializer**. This class assists the **ModuleManager** in creating instances of modules.
- **IModuleTypeLoader**. This is the interface for derived types (for example, the **XapModuleTypeLoader** class) to retrieve modules from the file system or a remote server.
- **Bootstrapper/MefBootstrapper/UnityBootstrapper**. This class assists applications in starting and initializing a modular Prism application.

Acceptance Tests

The Modularity QuickStarts include a separate solution with acceptance tests for both Unity and MEF QuickStarts. Acceptance tests describe how an application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the applications in a variety of scenarios.

To run the Modularity QuickStarts acceptance tests

1. In Visual Studio, open one of the following solution files:
 - QuickStarts\Modularity\ModularityWithUnity.Tests.AcceptanceTest\ModularityWithUnity.Tests.AcceptanceTest.sln
 - QuickStarts\Modularity\ModularityWithMEF.Tests.AcceptanceTest\ModularityWithMEF.Tests.AcceptanceTest.sln
2. Build the solution.
3. Open **Test Explorer**.

4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.
-

Outcome

When you run the acceptance tests, you should see the QuickStart windows and the tests automatically interact with the user interface. At the end of the test pass, you should see that all tests have passed.

More Information

To learn more about modularity, see [Modular Application Development](#).

Interactivity QuickStart

The Interactivity QuickStart demonstrates how views and view models can interact with the user. This includes interactions triggered from the view model and interactions fired by controls located in the view. To handle these different interactions the Prism library provides **InteractionRequests** and **InteractionRequestTriggers**, along with the custom **InvokeCommandAction** action.

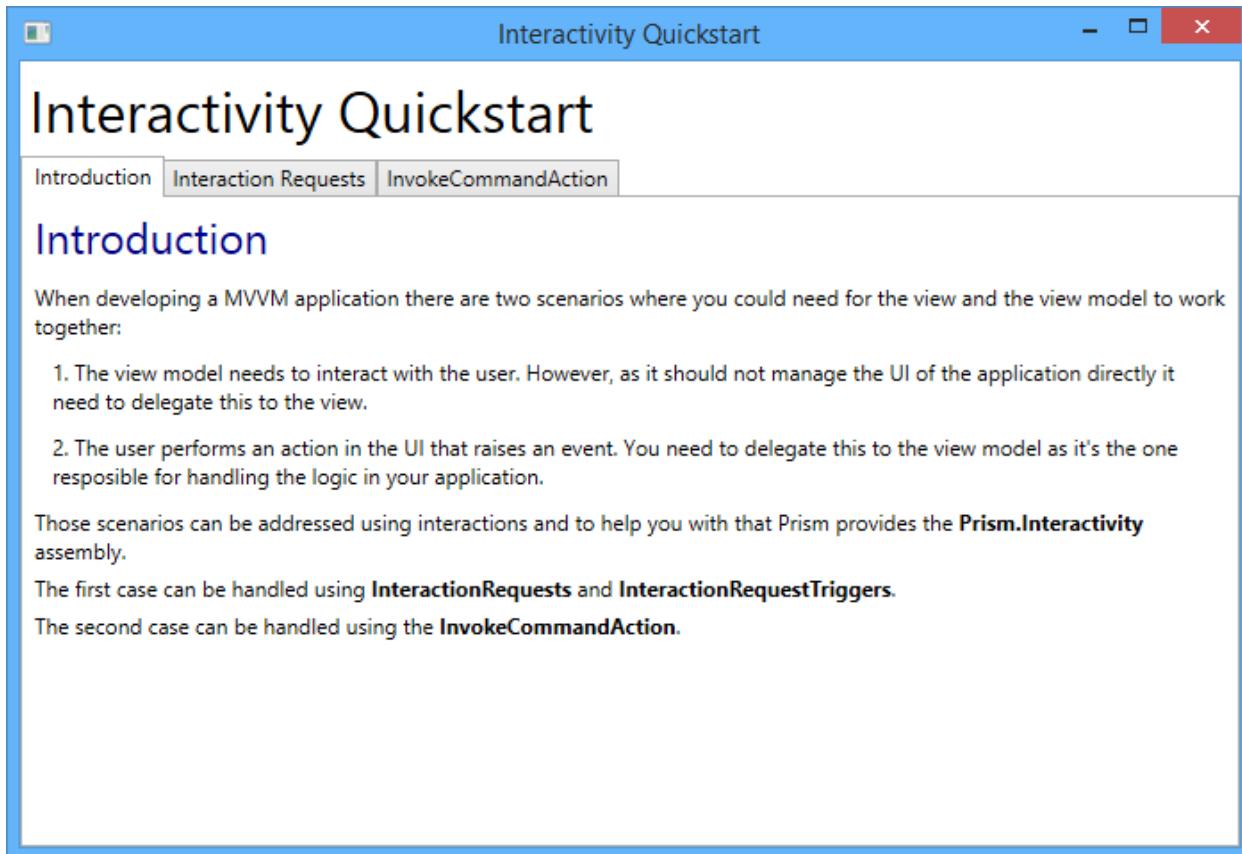
InvokeCommandAction is used to connect a trigger including events to a WPF command.

Scenarios

This section describes the scenarios included in the interactivity QuickStart. The QuickStart is composed of three tabs: Introduction, Interaction Requests, and InvokeCommandAction.

The **Introduction** tab, contains information about the purposes of the QuickStart. The **Interaction Requests** tab, which includes four different scenarios, describes the usage of **InteractionRequests** and **InteractionRequestTriggers** when a view model needs to interact with the user. Lastly, the **InvokeCommandAction** tab, demonstrates how a command can be invoked in response to an event raised by a control in the view.

The following illustration shows the main page of the Interactivity QuickStart.



Interactivity QuickStart user interface

Building and Running the QuickStart

This QuickStart requires the .Net Framework 4.5.1, Microsoft Visual Studio 2012 or later, and Blend for Visual Studio 2013.

To build and run the Interactivity QuickStart:

1. In Visual Studio, open the solution file QuickStart\Interactivity\InteractivityQuickstart.sln.
2. In the **Build** menu, click **Rebuild Solution**.
3. Press F5 to run the QuickStart.

Implementation Details

The QuickStart highlights the key components in user interactions. The following sections describe the key artifacts of the QuickStart.

In MVVM applications, there are scenarios where the view or the view model need to cooperate to interact with the user.

When the view model needs to interact with the user, it needs to delegate this interaction to the view, since the view model should not manipulate the UI. Prism provides **Interaction Requests** and **Interaction Request Triggers** to cover these scenarios.

When the user performs an action in the UI that raises an event that should trigger business logic, this business logic should be delegated to the view model, which is in charge of handling the application logic. Prism provides **InvokeCommandAction** to help with this scenario.

Interaction Requests

Prism provides Interaction Requests as a method to handle interactions initiated by a view model where the user should respond to.

Notification Interactions

A view model should define a property that holds the **InteractionRequest** instance; in this example it is called **NotificationRequest** and is typically initialized in the view model's constructor. Notice that the **InteractionRequest** is of type **INotification**, which represents an interaction request used for notifications.

C#

```
public InteractionRequest<INotification> NotificationRequest { get; private set; }
```

The view model will trigger the interaction when the **RaiseNotificationCommand** invokes the **RaiseNotification** method. The interaction is raised using the **Raise** method of the **InteractionRequest** instance, which receives an implementation of the **INotification** interface as well as a callback that will be executed when the interaction finishes. In this example, the default implementation of the **INotification** interface—the **Notification** class—is used.

C#

```
private void RaiseNotification()
{
    this.NotificationRequest.Raise(
        new Notification { Content = "Notification Message", Title = "Notification" },
        n => { InteractionResultMessage = "The user was notified." });
}
```

When creating the instance of the **Notification** class, the **Content** and **Title** properties are specified. **Content** is the message of the notification, and **Title** is the popup window caption.

To use interaction requests you need to define the corresponding **InteractionRequestTrigger** in the view's XAML code, as shown in the following code.

XAML

```
<prism:InteractionRequestTrigger SourceObject="{Binding NotificationRequest,
Mode=OneWay}">
    <prism:PopupWindowAction IsModal="True" CenterOverAssociatedObject="True"/>
</prism:InteractionRequestTrigger>
```

Note that the **InteractionRequestTrigger** has a **SourceObject** property that is bound to the **InteractionRequest** property in the view model.

The **InteractionRequestTrigger** has an associated **PopupWindowAction** provided by Prism that will execute when the view model raises the interaction request. This action will display a pop-up window using some of the out-of-the-box views, or you can specify custom pop-up views. The **IsModal** property will display this pop-up window as a modal when set to **true**, and the **CenterOverAssociatedObject** property will display the pop-up window in the center of the parent view if set to **true**.

As no custom window was specified in the example, the out-of-the-box popup window for Notifications will be used, as shown in the following figure.



Default Notification View

Confirmation Interactions

Confirmation Interactions display a message to the user, showing two buttons to either accept or cancel the interaction.

To send a Confirmation Interaction, declare an **InteractionRequest** property of type **IConfirmation** when you declare the property that will hold the **InteractionRequest** instance, as seen in the following code.

C#

```
public InteractionRequest<IConfirmation> ConfirmationRequest { get; private set; }
```

To raise the confirmation, the code is similar to notification Interactions, but this time you will pass a **Confirmation** class instance to the **Raise** method. The **Confirmation** class is the default implementation of the **IConfirmation** interface.

C#

```
private void RaiseConfirmation()
{
    this.ConfirmationRequest.Raise(
        new Confirmation { Content = "Confirmation Message", Title = "Confirmation" },
        c => { InteractionResultMessage = c.Confirmed ? "The user accepted." : "The user cancelled." });
}
```

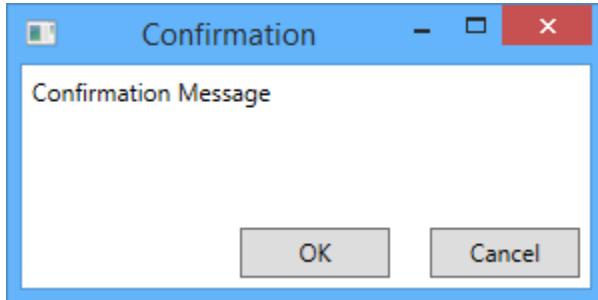
Notice that in the callback, different actions can be performed depending on the user's choice.

As this example uses the default popup window, the XAML definition is similar to the Notification Interaction, as you can see in the following code.

XAML

```
<prism:InteractionRequestTrigger SourceObject="{Binding ConfirmationRequest,
Mode=OneWay}">
    <prism:PopupWindowAction IsModal="True" CenterOverAssociatedObject="True"/>
</prism:InteractionRequestTrigger>
```

The following figure shows the default Confirmation popup window.



Default Confirmation view

Custom pop-up windows

To use custom popup windows instead of those provided out-of-the-box, use the **WindowContent** property of the **PopupWindowAction** action in the XAML definition of the Interaction, setting it to an instance of your custom popup window. This is demonstrated in the following code snippet.

XAML

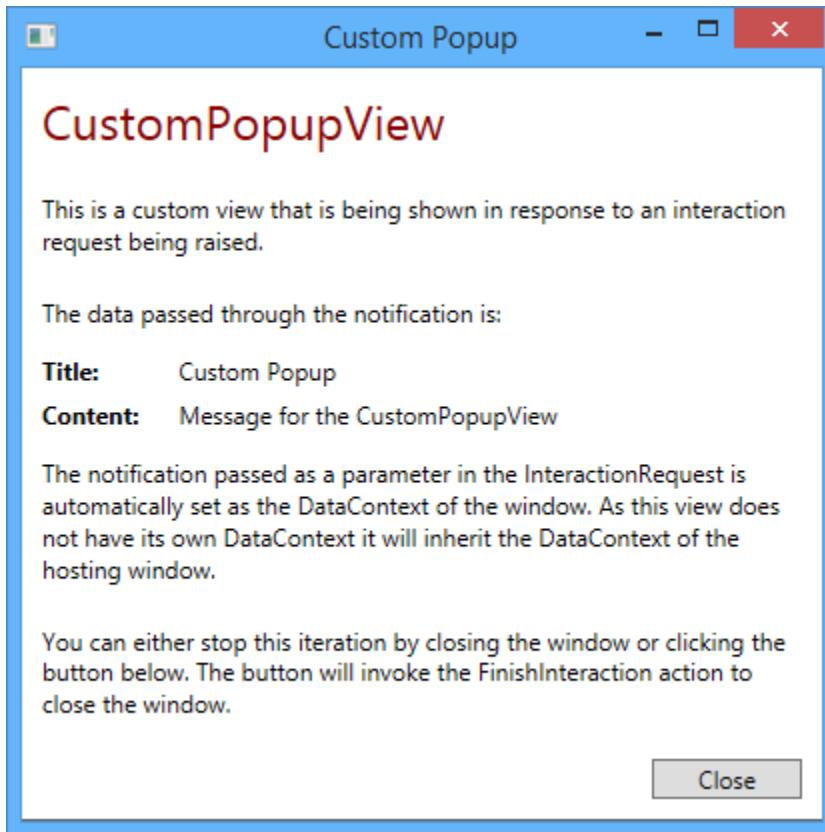
```
<prism:InteractionRequestTrigger SourceObject="{Binding CustomPopupViewRequest,
Mode=OneWay}">
    <prism:PopupWindowAction>
        <prism:PopupWindowAction.WindowContent>
            <views:CustomPopupView />
        </prism:PopupWindowAction.WindowContent>
    </prism:PopupWindowAction>
</prism:InteractionRequestTrigger>
```

The call to the **Raise** method of the interaction request instance is the same as a regular notification. In this case, we are passing a simple notification as a parameter. The custom popup view does not have a **DataContext** of its own; therefore, it will inherit the notification object passed as the **DataContext** of the window shown in the following code.

C#

```
private void RaiseCustomPopupView()
{
    this.InteractionResultMessage = "";
    this.CustomPopupViewRequest.Raise(
        new Notification { Content = "Message for the CustomPopupView", Title =
"Custom Popup" });
}
```

The following figures shows the custom pop-up window in action, which uses the **Notification** instance properties.



A custom popup view

Complex Custom Popup Windows

If you want to show a custom popup window that contains a more complex functionality, you can set a custom view model to your popup view.

In the following example, the **PopupWindowAction** action defines a custom view. When this action is executed the view will be shown inside a new window.

XAML

```
<prism:InteractionRequestTrigger SourceObject="{Binding ItemSelectionRequest,
Mode=OneWay}">
    <prism:PopupWindowAction>
        <prism:PopupWindowAction.WindowContent>
            <views:ItemSelectionView />
        </prism:PopupWindowAction.WindowContent>
    </prism:PopupWindowAction>
</prism:InteractionRequestTrigger>
```

Note: Take into account that the view and its view model are created only once and will be reused each time the action is executed.

Your custom popup view model needs to implement the **IInteractionRequestAware** interface in order to get the notification object from the interaction request as well as an action that can be invoked to finish the interaction. You can see the interface members in the following code.

C#

```
public interface IInteractionRequestAware
{
    INotification Notification { get; set; }

    Action FinishInteraction { get; set; }
}
```

The custom view model class will implement this interface, as shown in the code snippet.

C#

```
public class ItemSelectionViewModel : BindableBase, IInteractionRequestAware
{
    private ItemSelectionNotification notification;

    public ItemSelectionViewModel()
    {
        this.SelectItemCommand = new DelegateCommand(this.AcceptSelectedItem);
        this.CancelCommand = new DelegateCommand(this.CancelInteraction);
    }

    public Action FinishInteraction { get; set; }

    public INotification Notification
    {
        get
        {
            return this.notification;
        }
        set
        {
```

```

        if (value is ItemSelectionNotification)
    {
        this.notification = value as ItemSelectionNotification;
        this.OnPropertyChanged(() => this.Notification);
    }
}

public string SelectedItem { get; set; }

public ICommand SelectItemCommand { get; private set; }

public ICommand CancelCommand { get; private set; }

public void AcceptSelectedItem()
{
    if (this.notification != null)
    {
        this.notification.SelectedItem = this.SelectedItem;
        this.notification.Confirmed = true;
    }

    this.FinishInteraction();
}

public void CancelInteraction()
{
    if (this.notification != null)
    {
        this.notification.SelectedItem = null;
        this.notification.Confirmed = false;
    }

    this.FinishInteraction();
}
}

```

In the preceding code, note that the **Notification** property raises the **OnPropertyChanged** event when its value is updated.

To pass information to the custom popup view model, a custom **Confirmation** class is created. The **Confirmation** class is used instead of the **Notification** class, to take advantage of the **Confirmed** property to be able to determine if the user selected an item or closed the dialog. Think of this class as a Data Transfer Object (DTO). It will contain the properties that the popup view needs.

C#

```
public class ItemSelectionNotification : Confirmation
{
    public ItemSelectionNotification()
    {
        this.Items = new List<string>();
        this.SelectedItem = null;
    }

    public ItemSelectionNotification(IEnumerable<string> items)
        : this()
    {
        foreach(string item in items)
        {
            this.Items.Add(item);
        }
    }

    public IList<string> Items { get; private set; }

    public string SelectedItem { get; set; }
}
```

When you define the interaction request property in your view model, you will define it as the **ItemSelectionNotification** type, or whichever custom notification type you need, as seen in the following code.

C#

```
public InteractionRequest<ItemSelectionNotification> ItemSelectionRequest { get;
private set; }
```

Lastly, when you raise the interaction request, you will create an instance of your custom Notification (the **ItemSelectionNotification** class in this example) and add the required data for it to work. In this case, note that the items that populate the list are added to the corresponding **Items** property.

C#

```
private void RaiseItemSelection()
{
    ItemSelectionNotification notification = new ItemSelectionNotification();
    notification.Items.Add("Item1");
    notification.Items.Add("Item2");
    notification.Items.Add("Item3");
    notification.Items.Add("Item4");
    notification.Items.Add("Item5");
    notification.Items.Add("Item6");

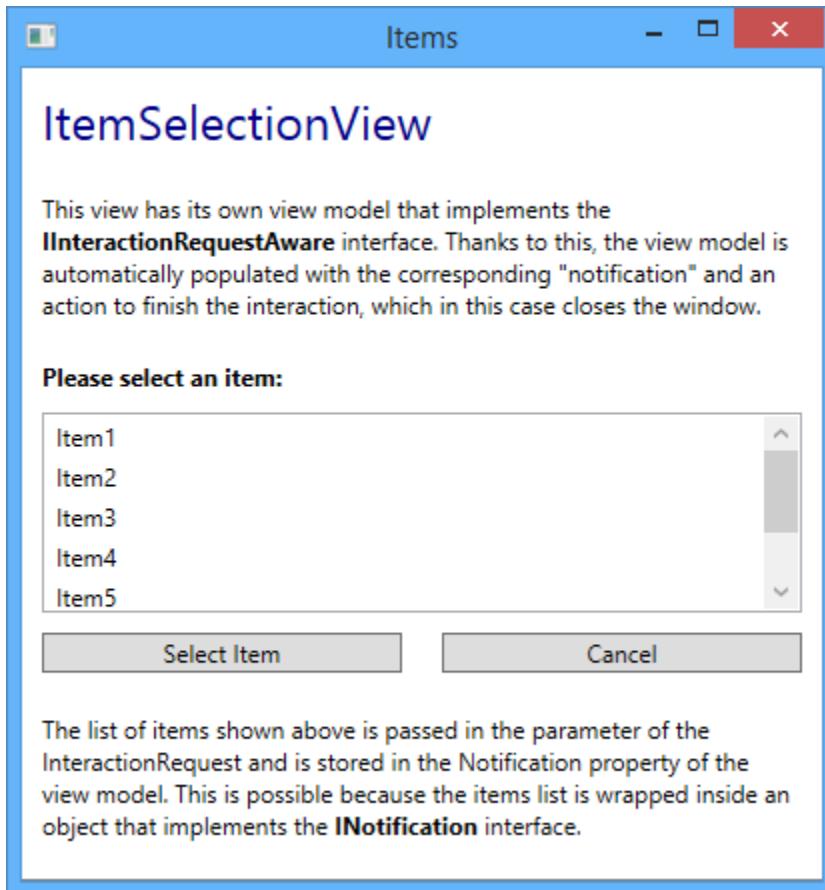
    notification.Title = "Items";
```

```
this.InteractionResultMessage = "";
this.ItemSelectionRequest.Raise(notification,
    returned =>
{
    if (returned != null && returned.Confirmed && returned.SelectedItem != null)
    {
        this.InteractionResultMessage = "The user selected: " +
    returned.SelectedItem;
    }
    else
    {
        this.InteractionResultMessage = "The user cancelled the operation or
    didn't select an item.";
    }
});
}
```

The custom popup view has its own view model which implements the **IInteractionRequestAware** interface; therefore, its **Notification** property will be automatically populated with this notification by the **PopupWindowAction**.

This way the parent view's view model and the popup window's view model are able to exchange data without direct references to each other.

In the following figure, you can see a custom popup view that provides a more complex functionality.



A custom popup view with a more complex interaction

InvokeCommandAction

When you need to invoke a command in response to an event raised by a control located in the view, you can use Blend's **InvokeCommandAction**.

The **InvokeCommandAction** allows you to execute a command in response to a triggered event. However, you cannot pass all or part of the **EventArgs** as a command parameter. Prism provides a custom **InvokeCommandAction** action that can help you in this case. It has an additional property called **TriggerParameterPath**, which is used to specify the member of the **EventArgs** of the fired event that will be passed as the command parameter. The **InvokeCommandAction** also sets the **IsEnabled** property of the associated control based on the value returned from **CanExecute** of the command.

To use the **InvokeCommandAction** action, in the view, you will register a trigger, such as **EventTrigger**, that will execute the **InvokeCommandAction** when the event is raised by the control. This action will execute a command passing the specified parameter of the event that triggered the action. You can see this in the following code.

XAML

```

<ListBox Grid.Row="1" Margin="5" ItemsSource="{Binding Items}"
SelectionMode="Single">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="SelectionChanged">
            <!-- This action will invoke the selected command in the view model and
pass the parameters of the event to it. -->
            <prism:InvokeCommandAction Command="{Binding SelectedCommand}"
TriggerParameterPath="AddedItems" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</ListBox>

```

In the preceding code the trigger will activate on the **SelectionChanged** event of the **Listbox** control, and the **InvokeCommandAction** will execute the **SelectedCommand** command, passing the **AddedItems** property of the **SelectionChangedEventArgs** as the command parameter. If the **TriggerParameterPath** property is not set, the **SelectionChangedEventArgs** instance will be directly passed to the command. If the **CommandParameter** property is set, the trigger parameter will be ignored.

Key Interactivity Classes

The following are some key classes used in the Interactivity QuickStart:

- **IInteractionRequest**. Interface that represents a request for user interaction. View models can expose interaction request objects through properties and raise them when user interaction is required so that views associated with the view models can materialize the user interaction using an appropriate mechanism.
- **InteractionRequest<T>**. Implementation of the **IInteractionRequest** interface.
- **INotification**. Interface that represents an interaction request used for notifications.
- **Notification**. Basic implementation of **INotification** containing the **Title** and **Content** properties.
- **IConfirmation**. Represents an interaction request used for confirmations. It contains the **Confirmed** property, which indicates if the confirmation was accepted or not.
- **Confirmation**. Basic implementation of **IConfirmation**. It also inherits from the **Notification** class.
- **IInteractionRequestAware**. Interface used by the **PopupWindowAction** class. If the **DataContext** object of a view that is shown with this action implements this interface, it will be populated with the **INotification** data of the interaction request as well as an Action to finish the request upon invocation.
- **InvokeCommandAction**. Trigger action that executes a command when invoked. It contains the **TriggerParameterPath** property that is parsed to identify the child property of the trigger parameter that will be used as the command parameter.

Acceptance Tests

The Interactivity QuickStart include a separate solution with acceptance tests. Acceptance tests describe how an application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the applications in a variety of scenarios.

To run the Interactivity QuickStart acceptance tests

1. In Visual Studio, open the QuickStarts\Interactivity\Interactivity.Tests.AcceptanceTest\Interactivity.Tests.AcceptanceTest.sln solution file.
 2. Build the solution.
 3. Open Test Explorer.
 4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.
-

Outcome

When you run the acceptance tests, you should see the QuickStart window and the tests automatically interact with the user interface. At the end of the test pass, you should see that all tests have passed.

More Information

To learn more about interactivity, see [Composing the User Interface](#).

MVVM QuickStart

The Model-View-ViewModel (MVVM) QuickStart provides sample code that demonstrates how to separate the state and logic that support a view into a separate class named **ViewModel** using the Prism Library. The view model sits on top of the application data model to provide the state or data needed to support the view, insulating the view from needing to know about the full complexity of the application. The view model also encapsulates the interaction logic for the view that does not directly depend on the view elements themselves. This QuickStart provides a tutorial on implementing the MVVM pattern.

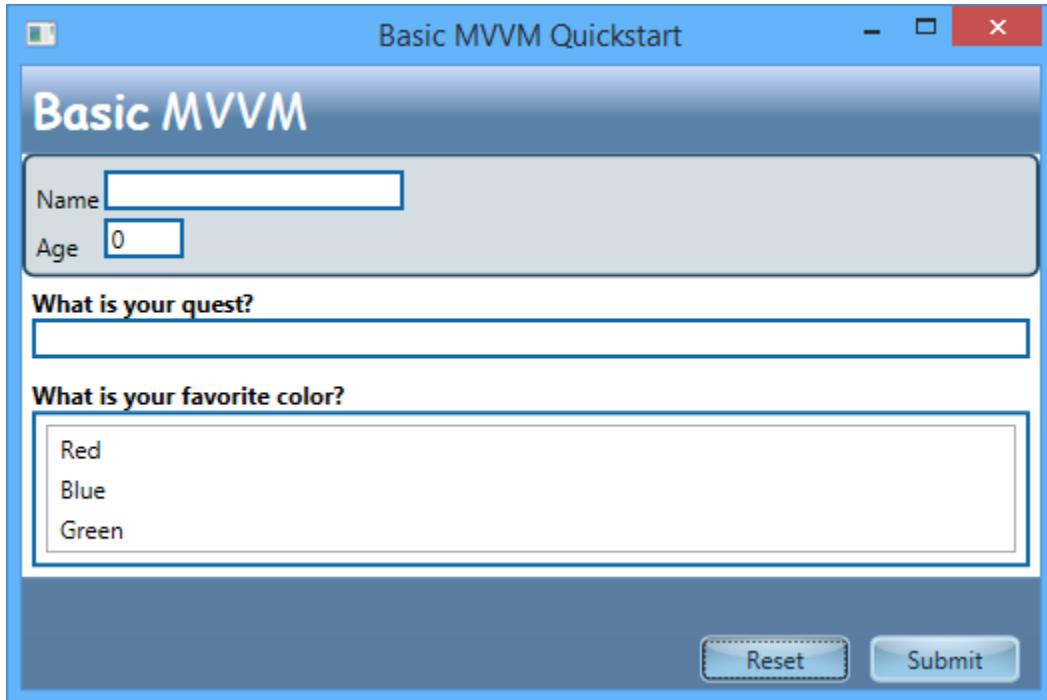
A common approach to designing the views and view models in an MVVM application is the first sketch out or storyboard for what a view looks like on the screen. Then you analyze that screen to identify what properties the view model needs to expose to support the view, without worrying about how that data will get into the view model. After you define what the view model needs to expose to the view and implement that, you can then dive into how to get the data into the view model. Often, this involves the view model calling to a service to retrieve the data, and sometimes data can be pushed into a view model from some other code such as an application controller.

This QuickStart leads you through the following steps:

- Analyzing the view to decide what state is needed from a view model to support it
 - Defining the view model class with the minimum implementation to support the view
 - Defining the bindings in the view that point to view model properties
 - Attaching the view to the view model
-

Business Scenario

The main window of the Basic MVVM Application QuickStart represents a subset of a survey application. In this window, an empty survey with different types of questions is shown; and there is a button to submit the questionnaires. The following illustration shows the QuickStart main window.



MVVM QuickStart user interface

Building and Running the QuickStart

This QuickStart requires Microsoft Visual Studio 2012 or later and the .NET Framework 4.5.1.

To build and run the MVVM QuickStart

1. In Visual Studio, open the solution file
Quickstarts\BasicMVVMQuickstart/Desktop\BasicMVVMQuickstart/Desktop.sln.
2. In the **Build** menu, click **Rebuild Solution**.
3. Press F5 to run the QuickStart.

Implementation Details

The QuickStart highlights the key elements and considerations to implement the MVVM pattern in a basic application.

Analyzing What Properties Are Needed on the View Model

Open the Views\MainWindow view in the designer. The list of color selections will be dynamically populated. When analyzing a view to define a view model, you want to identify each individual item of data and behavior that you need. In this case, assuming the questions are fixed and will not be dynamically driven, you need the following properties exposed from your view model:

- Name: string

- Age: int
- Quest: string
- FavoriteColor: string
- Submit: Command
- Reset: Command

Because the first four properties are related to questionnaires, a questionnaire class is created to store them. The questionnaire class will be the model of the application, and the view model will only expose a property of type **Questionnaire** to support them.

Note that even things like buttons represent something that needs support from the view model. You can either expose a command, as shown in this QuickStart, or you can expose a method. With the former, you will need a property exposed from the view model with an object that implements the **ICommand** interface; with the latter, you need a behavior that can target a method.

Note: For button clicks, you have the choice of commands or behaviors. For more information, see [Command-Enabled Controls vs. Behaviors](#) in [Advanced MVVM Scenarios](#). In this topic, you will use a command. To do that, you need a command implementation, which does not exist in a form compatible with view models in the .NET Framework. Prism provides the **DelegateCommand** class that is perfect for hooking up views to view models with commands.

As we want to demonstrate parent-child view model composition, the application UI is composed by two views: **MainWindow**, which contains the **Reset** and **Submit** buttons and an instance of the second view, which is the **QuestionnaireView** that includes the questionnaire's questions.

The **QuestionnaireView** is directly instantiated in the XAML code, as seen in the following code.

XAML

```
<Window x:Class="BasicMVVMQuickstart/Desktop.Views.MainWindow" ...>
    <Grid x:Name="LayoutRoot"
        Background="{StaticResource MainBackground}">
        <Grid MinWidth="300" MaxWidth="800">
            ...
            <views:QuestionnaireView Grid.Row="1"
                DataContext="{Binding QuestionnaireViewModel}"
                Height="246" VerticalAlignment="Top">
                </views:QuestionnaireView>
            ...
        </Grid>
    </Grid>
</Window>
```

In order to populate this child view with its corresponding view model, its **DataContext** is set to a property of the **MainWindow**'s view model that contains an instance of the child **QuestionnaireView**'s view model.

Implementing the View Model to Support the View

Open the **QuestionnaireViewModel.cs** file. The view model implements the **Questionnaire** and **AllColors** properties and derives from the **BindableBase** class.

C#

```
public class QuestionnaireViewModel : BindableBase
{
    private Questionnaire questionnaire;
    public QuestionnaireViewModel()
    {
        this.Questionnaire = new Questionnaire();
        this.AllCollors = new[] { "Red", "Blue", "Green" };
    }

    public Questionnaire Questionnaire
    {
        get { return this.questionnaire; }
        set { SetProperty(ref this.questionnaire, value); }
    }
    public IEnumerable<string> AllCollors { get; private set; }
}
```

The **INotifyPropertyChanged** interface is implemented on the **BindableBase** base class. The property changed notification is added to the whole **Questionnaire** property, using the **SetProperty** method of the **BindableBase** class as shown in the following code.

C#

```
public Questionnaire Questionnaire
{
    get { return this.questionnaire; }
    set { SetProperty(ref this.questionnaire, value); }
}
```

Note: The view model class typically derives from the **BindableBase** class. In some cases, the model can derive from **BindableBase**, when the property that needs to update the view when its value is changed is stored in the model.

To support **INotifyPropertyChanged**, your class needs to derive from the **BindableBase** class, and the property setter needs to call the **SetProperty** method of the **BindableBase** class.

The following code shows how the **BindableBase** class implements the **INotifyPropertyChanged** interface. Note that the **SetProperty** method updates the property's value and fires the **PropertyChanged** event when a property changes its value. Alternatively, you can use the

OnPropertyChanged method, passing a lambda expression that references the property, to fire the **PropertyChanged** event. This is useful for when one property update triggers another property update. And also provides backward compatibility with the previous version of Prism.

C#

```
public abstract class BindableBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual bool SetProperty<T>(ref T storage, T value, [CallerMemberName]
string propertyName = null)
    {
        if (object.Equals(storage, value)) return false;

        storage = value;
        this.OnPropertyChanged(propertyName);

        return true;
    }

    protected void OnPropertyChanged(string propertyName)
    {
        var eventHandler = this.PropertyChanged;
        if (eventHandler != null)
        {
            eventHandler(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    protected void OnPropertyChanged<T>(Expression<Func<T>> propertyExpression)
    {
        var propertyName = PropertySupport.ExtractPropertyName(propertyExpression);
        this.OnPropertyChanged(propertyName);
    }
}
```

The method is used so that when the questionnaire property changes its value and updates the user interface. In any view where the data might change in the view model and you want those changes to be reflected on the screen, you need to implement this pattern for all the properties in the view model.

The questionnaire property, the collection property, and the command property are initialized in the view model class constructor.

C#

```
public QuestionnaireViewModel()
{
    this.questionnaire = new Questionnaire();
    this.AllColors = new[] { "Red", "Blue", "Green" };
}
```

Collection properties should always be initialized to either an empty collection or, if it is appropriate, to populate the collection in the constructor, you can do so, typically by calling a service. If you do that, make sure you do so in a way that will not break the designer.

Additionally, if you expose **ICommand** properties that the view can bind command properties to, you need to create an instance of a command object. In this case, because you will use the **DelegateCommand** type from Prism, you need to create an instance of that and point it to a handling method. **DelegateCommand** also has the ability to carry along a strongly typed parameter if the **CommandParameter** property of a source control is also set. This is not used in the QuickStart, so the argument type is just specified as object.

The following code shows the **OnSubmit** handler method, located in the **MainWindowViewModel** class.

C#

```
private void OnSubmit(object obj)
{
    Debug.WriteLine(BuildResultString());
}
```

To keep the solution simple, this handler method returns the values entered for the questions to the Output window in Visual Studio with the help of a helper method that is already implemented in the view model class. A real implementation of a command handler would typically do something like call out to a service to save the results to a repository or retrieve data if it was a Load type of operation. It might also cause navigation to another view to occur by calling to a navigation service.

Wiring Up the View Elements to the View Model

The bindings in the view elements point to the view model properties, as shown in the following table. Note that these bindings are located in both the **MainWindow** view and in the **QuestionnaireView** view.

Element name	Property	Value
NameTextBox	Text	{Binding Path= Questionnaire.Name, Mode=TwoWay}
AgeTextBox	Text	{Binding Path=Questionnaire.Age, Mode=TwoWay}
Question1Response	Text	{Binding Path=Questionnaire.Quest, Mode=TwoWay}
ColorRun	Foreground	{Binding Questionnaire.FavoriteColor, TargetNullValue=Black}
Colors	ItemsSource	{Binding Path=AllColors}
Colors	SelectedItem	{Binding Questionnaire.FavoriteColor, Mode=TwoWay}
SubmitButton	Command	{Binding SubmitCommand}
ResetButton	Command	{Binding ResetCommand}

Creating the View and View Model and Hooking Them Up

There are several ways of hooking up the view model with the view. You can create the view model in the view's code behind and set it in its **DataContext** property or set it declaratively in the view's Xaml code. To instantiate the view model in XAML, the view model class must have a default constructor.

Another approach, is creating a component that can locate the corresponding view model and put it in the **DataContext** automatically, this component is typically called **View Model Locator**.

Prism provides an implementation of the View Model Locator pattern, which is an attached property.

Open MainWindow.xaml and look for the code where the view model locator property is attached. The attached property is shown in the last line of the following code.

XAML

```
<Window x:Class="BasicMVVMQuickstart/Desktop.Views.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:basicMvvmQuickstartDesktop="clr-namespace:BasicMVVMQuickstart/Desktop"
        xmlns:viewModel="clr-
namespace:Microsoft.Practices.Prism.Mvvm;assembly=Microsoft.Practices.Prism.Mvvm.Desk
top"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:views="clr-namespace:BasicMVVMQuickstart/Desktop.Views"
        mc:Ignorable="d"
        Title="Basic MVVM Quickstart"
        Height="350"
        Width="525"
        d:DataContext="{d:DesignInstance
basicMvvmQuickstartDesktop:QuestionnaireViewDesignViewModel,
IsDesignTimeCreatable=True}"
        viewModel:ViewModelLocator.AutoWireViewModel="True">
```

Prism's view model locator is an attached property that when set to true it will try to locate the view model of the view, and then set the view's data context to an instance of the view model. To locate the corresponding view model, the view model locator uses two approaches. First it will look for the view model in a view name/view model registration mapping. If a registration is not found, it will fall back to a convention-based approach, that will locate the view models, by replacing ".View" from the view namespace with ".ViewModel" and appending "ViewModel" to the view's name. For more information about ways to hook up views to view models; see [Implementing the MVVM Pattern](#).

Adding Design-Time Support

When you use the view model locator, your view models are created at runtime. Therefore, when you are designing your view, the view model is not yet constructed and you will not see the view model data at design time.

To solve this situation, you can use the **d:DataContext** designer property and set it to a view model created specifically for design time. This view model will be constructed only at design time, it is a simplification on the real view model, and may contain mock data.

You can see how this property is used in the `MainWindow` page, in the following code.

XAML

```
d:DataContext="{d:DesignInstance
basicMvvmQuickstartDesktop:QuestionnaireViewDesignViewModel,
IsDesignTimeCreatable=True}"
```

Note that you need to specify the class that will be used as the **DesignInstance**, and then set the **IsDesignTimeCreatable** property to **true**. The design view model class used as **DesignInstance** is a class that must have a default constructor.

You can see how the design view model for the QuickStart is defined, in the following code:

C#

```
public class QuestionnaireViewDesignViewModel
{
    public QuestionnaireViewDesignViewModel()
    {
        this.QuestionnaireViewModel = new QuestionnaireViewModel();
    }

    public QuestionnaireViewModel QuestionnaireViewModel { get; set; }
}
```

This design view model just has to initialize the properties used in the view for binding and populate them with mock data. As it is used only for design, it is not necessary to derive from **BindableBase**, nor implement the **INotifyPropertyChanged** interface.

More Information

For more information about implementing the MVVM pattern, see the following topics:

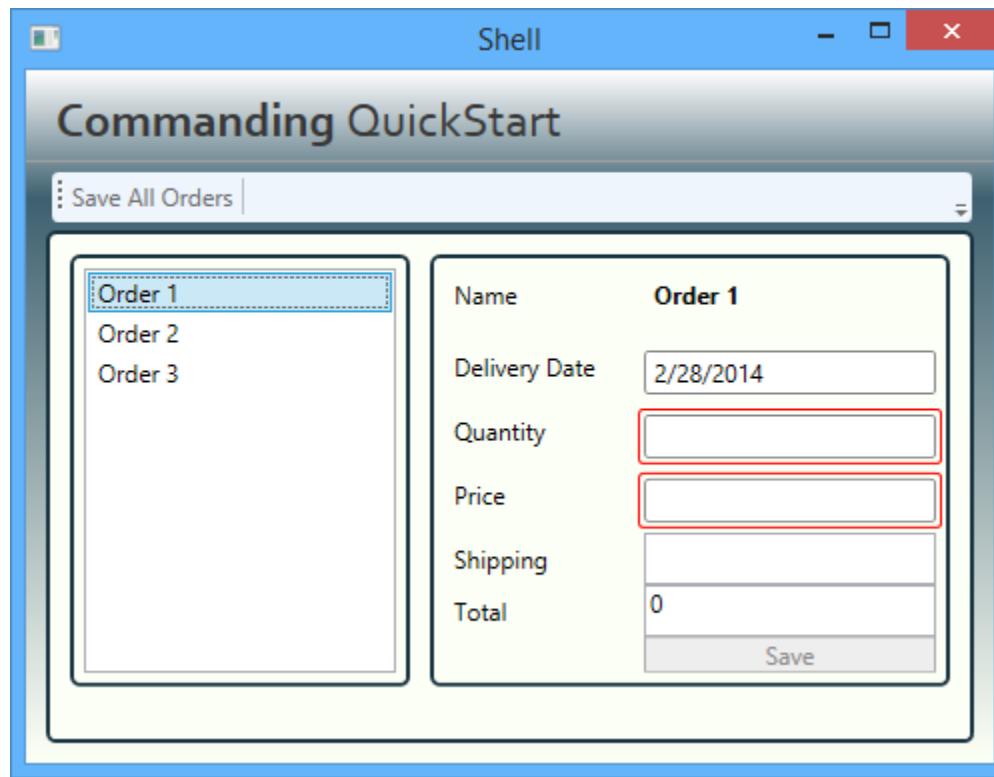
- [Implementing the MVVM Pattern](#)
 - [Advanced MVVM Scenarios](#)
-

Commanding QuickStart

The Commanding QuickStart sample demonstrates how to build a Windows Presentation Foundation (WPF) application that uses delegate and composite commands provided by the Prism Library to handle UI actions in a decoupled way. This is useful when implementing the Model-View-ViewModel (MVVM) pattern. The Prism Library also provides an implementation of the `ICommand` interface.

Business Scenario

The Commanding QuickStart is based on a fictitious product ordering system. The main window represents a subset of a larger system. In this window, the user can place customer orders and submit them. The following illustration shows the QuickStart's main window.



Commanding QuickStart

Building and Running the QuickStart

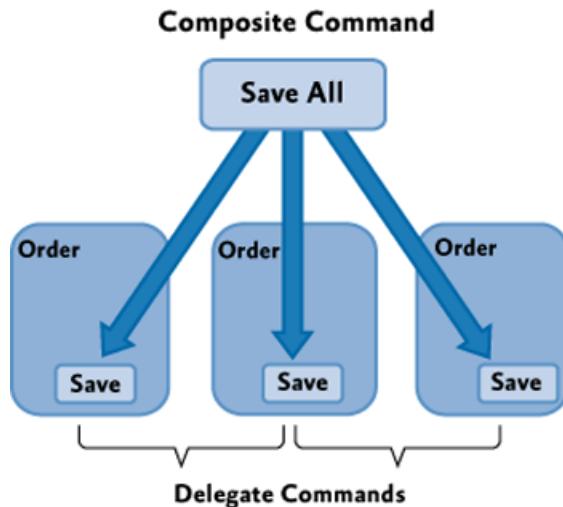
This QuickStart requires Visual Studio 2012 or later and the .NET Framework 4.5.1 to run.

To build and run the QuickStart

1. In Visual Studio, open the solution file `Quickstarts\Commanding\Commanding_Desktop.sln`.
2. On the **Build** menu, click **Rebuild Solution**.
3. Press F5 to run the QuickStart.

Implementation Details

The QuickStart highlights the key implementation details of an application that uses commands. The following illustration shows the key artifacts in the application.



Commanding QuickStart conceptual view

Note: The QuickStart contains a number of TODO comments to help navigate the important concepts in the code. Use the Task List window in Visual Studio to see a list of these important areas of code. Make sure you select **Comments** in the dropdown box. If you double-click an item in the list, the code file will open in the appropriate line.

Delegate Commands

By using the **DelegateCommand** command, you can supply delegates for the **Execute** and **CanExecute** methods. This means that when the **Execute** or **CanExecute** methods are invoked on the command, the delegates you supplied are invoked.

In the Commanding QuickStart, the **Save** button on each order form is associated to a delegate command. The delegates for the **Execute** and **CanExecute** methods are the **Save** and **CanSave** methods of the **OrderViewModel** class, respectively (this class is the view model for an order; for the class definition, see the file `Commanding.Modules.Order.Desktop\ViewModels\OrderViewModel.cs`).

The following code shows the constructor of the **OrderViewModel** class. In the method body, a delegate command named **SaveOrderCommand** is created—it passes delegates for the **Save** and **CanSave** methods as parameters.

C#

```

public OrderViewModel( Services.Order order )
{
    _order = order;

    //TODO: 01 - Each Order defines a Save command.
    this.SaveOrderCommand = new DelegateCommand<object>( this.Save, this.CanSave );
  
```

```

    // Track all property changes so we can validate.
    this.PropertyChanged += this.OnPropertyChanged;

    this.Validate();
}

```

The following code shows the implementation of the **Save** and **CanSave** methods.

C#

```

private bool CanSave( object arg )
{
    //TODO: 02 - The Order Save command is enabled only when all order data is valid.
    // Can only save when there are no errors and
    // when the order quantity is greater than zero.
    return this.errors.Count == 0 && this.Quantity > 0;
}

private void Save( object obj )
{
    // Save the order here.
    Console.WriteLine(
        String.Format( CultureInfo.InvariantCulture, "{0} saved.", this.OrderName ) );

    // Notify that the order was saved.
    this.OnSaved( new DataEventArgs<OrderPresentationModel>( this ) );
}

```

The following code shows the **OnPropertyChanged** method implementation. This method is an event handler for the **PropertyChanged** event, which gets raised whenever the user changes a value in the order form. This method updates the order's total, validates the data, and raises the **CanExecuteChanged** event of the **SaveOrderCommand** command to notify the command's invokers about the state change.

C#

```

private void OnPropertyChanged( object sender, PropertyChangedEventArgs e )
{
    // Total is a calculated property based on price, quantity and shipping cost.
    // If any of these properties change, then notify the view.
    string propertyName = e.PropertyName;
    if ( propertyName == "Price" || propertyName == "Quantity" || propertyName ==
"Shipping" )
    {
        this.NotifyPropertyChanged( "Total" );
    }

    // Validate and update the enabled status of the SaveOrder
    // command whenever any property changes.
    this.Validate();
}

```

```

    this.SaveOrderCommand.RaiseCanExecuteChanged();
}

```

The following code, located in the file Commanding.Modules.Order.Desktop\Views\OrdersEditorView.xaml, shows how the **Save** button is bound to the **SaveOrderCommand** command.

XAML

```
<Button AutomationProperties.AutomationId="SaveButton" Grid.Row="6" Grid.Column="1"
Content="Save" Command="{Binding SaveOrderCommand}"></Button>
```

Composite Commands

A **CompositeCommand** is a command that has multiple child commands. A **CompositeCommand** is used in the Commanding QuickStart for the **Save All** button on the main toolbar. When you click the **Save All** button, the **SaveAllOrdersCommand** composite command executes, and in consequence, all its child commands—**SaveOrderCommand** commands—execute for each pending order.

The **SaveAllOrdersCommand** command is a globally available command, and it is defined in the **OrdersCommands** class (the class definition is located at Commanding.Modules.Order.Desktop\OrdersCommands.cs). The following code shows the implementation of the **OrdersCommands** static class.

C#

```

public static class OrdersCommands
{
    public static CompositeCommand SaveAllOrdersCommand = new CompositeCommand();
}
```

The following code, extracted from the file Commanding.Modules.Order.Desktop\ViewModels\OrdersEditorViewModel.cs, shows how child commands are registered with the **SaveAllOrdersCommand** command. In this case, a proxy class is used to access the command. For more information, see "Proxy Class for Global Commands" later in this topic.

C#

```

private void PopulateOrders()
{
    _orders = new ObservableCollection<OrderPresentationModel>();

    foreach ( Services.Order order in this.ordersRepository.GetOrdersToEdit() )
    {
        // Wrap the Order object in a presentation model object.
        var orderPresentationModel = new OrderViewModel( order );
        _orders.Add( orderPresentationModel );

        // Subscribe to the Save event on the individual orders.
        orderPresentationModel.Saved += this.OrderSaved;
    }
}
```

```

    //TODO: 04 - Each Order Save command is registered with the application's
SaveAll command.
    commandProxy.SaveAllOrdersCommand.RegisterCommand(
orderPresentationModel.SaveOrderCommand );
}
}

```

When an order is saved, the **SaveOrderCommand** child command for that particular order must be unregistered. The following code shows how this is done in the implementation of the **OrderSaved** event handler, which executes when an order is saved.

C#

```

private void OrderSaved(object sender, DataEventArgs<OrderViewModel> e)
{
    if (e != null && e.Value != null)
    {
        OrderViewModel order = e.Value;
        if (this.Orders.Contains(order))
        {
            order.Saved -= this.OrderSaved;

this.commandProxy.SaveAllOrdersCommand.UnregisterCommand(order.SaveOrderCommand);
            this.Orders.Remove(order);
        }
    }
}

```

The following XAML markup code shows how the **SaveAllOrdersCommand** command is bound to the **SaveAllToolBarButton** button in the toolbar. This code is located at `Commanding.Modules.Order/Desktop/OrdersToolBar.xaml`.

XAML

```

<ToolBar>
    <Button AutomationProperties.AutomationId="SaveAllToolBarButton" Command="{x:Static
inf:OrdersCommands.SaveAllOrdersCommand}">Save All Orders</Button>
    <Separator />
</ToolBar>

```

Proxy Class for Global Commands

To create a globally available command, you typically create a static instance of a **CompositeCommand** class and expose it publicly through a static class. This approach is straightforward, because you can access the command instance directly from your code. However, this approach makes your classes that use the command hard to test in isolation, because your classes are tightly coupled to the command. When testability is a concern in an application, a proxy class can be used to access global commands. A proxy class can be easily replaced with mock implementations when writing unit tests.

The Commanding QuickStart implements a proxy class named **OrdersCommandProxy** to encapsulate the access to the **SaveAllOrdersCommand** (the class definition is located at

Commanding.Modules.Order.Desktop\OrdersCommands.cs). The class, shown in the following code, implements a public property to return the **SaveAllOrdersCommands** command instance defined in the **OrdersCommands** class.

C#

```
public class OrdersCommandProxy
{
    public virtual CompositeCommand SaveAllOrdersCommands
    {
        get { return OrdersCommands.SaveAllOrdersCommand; }
    }
}
```

In the preceding code, note that the **SaveAllOrdersCommands** property can be overwritten in a mock class to return a mock command.

For more information about creating globally available commands, see [Binding to a Globally Available Command](#) in [Communicating Between Loosely Coupled Components](#).

Acceptance Tests

The Commanding QuickStart includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the Commanding QuickStart acceptance tests

1. In Visual Studio, open the solution file QuickStarts\Commanding\Commanding.Tests.AcceptanceTest\Commanding.Tests.AcceptanceTest.sln.
2. Build the solution.
3. Open **Test Explorer**.
4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.

You should see the QuickStart window and the tests interact with the application. At the end of the test run, you should see that all tests have passed.

More Information

For more information about commands, see the following topics:

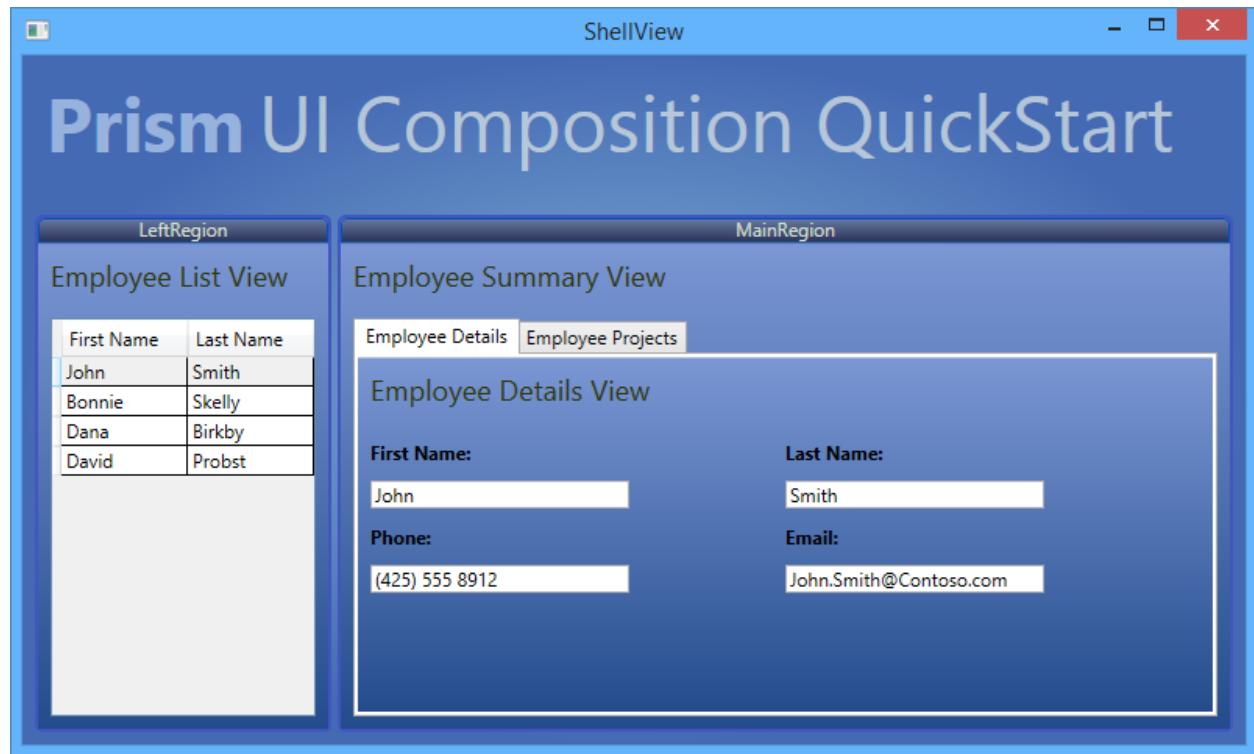
- [Implementing the MVVM Pattern](#)
- [Communicating Between Loosely Coupled Components](#)

UI Composition QuickStart

The UI Composition QuickStart sample illustrates how to use both the view discovery and view injection approaches for user interface (UI) composition with the Prism Library for WPF. When using view discovery, modules can register views (or presentation models) against a particular named location. When that location is displayed at run time, any views that have been registered for that location will be automatically created and displayed within it. In the view injection approach, views are programmatically added or removed from a named location by the modules that manage them. To enable this, the application contains a registry of named locations in the UI, and a module can look up one of the locations using the registry and then programmatically inject views into it.

Business Scenario

The UI Composition QuickStart is based on a fictitious resource management system. The main window represents a subset of a larger system. In this window, the user can review detailed information about employees of a company, update their contact information, and view the projects each employee is assigned to. The following illustration shows the QuickStart's main window.



UI Composition QuickStart

Building and Running the QuickStart

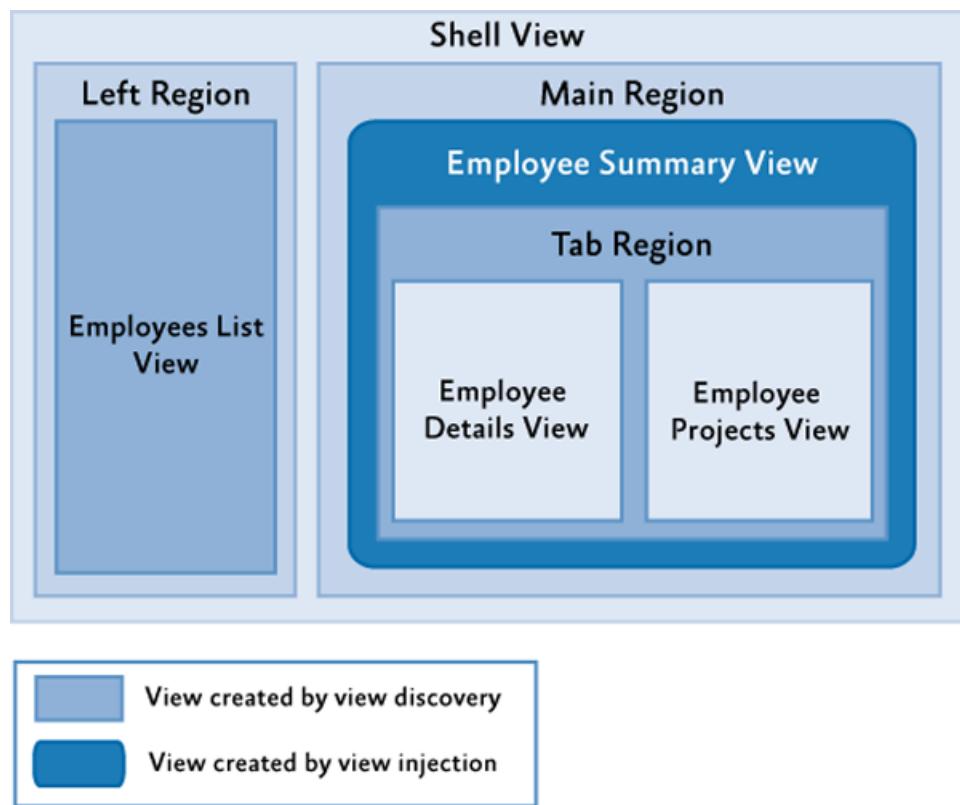
This QuickStart requires Visual Studio 2012 or later and the .NET Framework 4.5.1 to run.

To build and run the QuickStart

1. In Visual Studio, open the solution file QuickStarts\UIComposition_Desktop\UICompositionQuickstart/Desktop.sln.
 2. In the **Build** menu, click **Rebuild Solution**.
 3. Press F5 to run the QuickStart.
-

Implementation Notes

The QuickStart highlights the key implementation details of an application that uses regions, using both the view discovery and view injection approaches to composition. The following illustration shows the key artifacts in the application.



UI Composition QuickStart conceptual view

The following artifacts are illustrated in the preceding figure:

- **Shell view**. This is the application's main window. This window contains both the left and main regions.
- **Left region**. This region contains the view that includes the list of employees, through view discovery.
- **Employees List view**. This view displays a list of employees.

- **Main region.** This region has the employee summary view injected into it.
- **Employee Summary view.** This view displays information for an employee and contains a tab region. It is added to the application via view injection.
- **Tab region.** The tab region resides in the employee summary view, and has the employee details and projects views added via view discovery.
- **Employee Details view.** This view shows the details about the selected employee.
- **Employees Projects view.** This view displays the list of projects an employee is working on.

Note: The QuickStart contains TODO comments to help navigate the important concepts in the code. Use the Task List window in Visual Studio to see a list of these important areas of code. Make sure that **Comments** is selected in the dropdown box of the Task List window. If you double-click an item in the list, the code file will open in the appropriate line.

Composing the User Interface

The UI Composition QuickStart shows both view discovery and view injection in one application.

The regions are set up in XAML code:

- The Shell defines the two regions **LeftRegion** and **MainRegion** in **ShellView.xaml**.
- The **EmployeeSummaryView** contains a **Tab** control, which defines a region named **TabRegion** in **EmployeeSummaryView.xaml**.
- The **TabRegion** defines a **RegionContext**, which provides a reference to the currently selected employee to all child views, also in **EmployeeSummaryView.xaml**.

The application determines what views to display in the following manner:

1. The **OnStartup** method in **App.xaml.cs** creates and runs the QuickStart's bootstrapper.
2. In the bootstrapper's **ConfigureModuleCatalog** method, the module catalog is loaded. This module catalog defines a single **EmployeeModule** implemented in the **ModuleInit** class that is loaded by Prism as soon as it is available.
3. In the **EmployeeModule**, the **ModuleInit** class's **Initialize** method is called by the bootstrapper. This method does a variety of things, including registering an **IEmployeeDataService** (a repository of employee data) and creating an instance of the **MainRegionController**. Using lambda expressions, the **Initialize** method also registers an **EmployeeListView** with the **LeftRegion**, and both an **EmployeeDetailView** and **EmployeeProjectsView** with the **TabRegion** as an example of view discovery. At this point, these regions have not been created.
4. A bit later in the bootstrapping process, the bootstrapper's **CreateShell** method is called, creating an instance of the **ShellView**, which includes the **LeftRegion** and **MainRegion**.

5. In the process of creating the **LeftRegion**, Prism determines that a view has been registered with the region and uses the registered lambda expression to create an instance of the **EmployeeListView**. This view is activated and displayed.
6. Because there are no views registered with the **MainRegion**, nothing is shown here.

When an employee is selected from the **EmployeesListView**, the following occurs:

- The **MainRegionController**'s **EmployeeSelected** method is called because of an event subscription to the **EmployeeSelectedEvent** through the **EventAggregator**. This event is published by the **SelectedEmployeeChanged** method in the **EmployeeListViewModel** class.

In the **EmployeeSelected** method, the following occurs:

1. The selected employee is retrieved from the **EmployeeDataService**.
2. The **MainRegion** is retrieved from the **RegionManager**, and the method attempts to find a view named **EmployeeSummaryView**. Because this view has not been created, it retrieves an instance from the container and explicitly injects the view into the **MainRegion**, showing view injection.
3. When the **EmployeeSummaryView** is created, the **TabRegion** it contains is also created. This region has a **RegionContext** bound to the **CurrentEmployee** (in EmployeeViewSummary.xaml). When this happens, several other things occur:
 - a. Prism determines that the **TabRegion** has multiple views registered with it, and it creates instances of the **EmployeeDetailsView** and **EmployeeProjectsView** and displays them, showing another example of view discovery. Both of these views subscribe to the **RegionContext**'s **PropertyChanged** event.
 - b. The binding on the **RegionContext** is updated, and the **PropertyChanged** event is triggered.
 - c. Both the **EmployeeDetailsView** and **EmployeeProjectsView** are notified of the **RegionContext** property change and the associated view models are updated, causing the current employee to be displayed.

Applying View Discovery and View Injection

The view discovery approach allows pulling views inside regions, based on a registry where modules store a collection of pairs (such as views type, region name). This registry is named the **RegionViewRegistry**. When a region is created, it looks for all the view types associated with its region name in the **RegionViewRegistry**. The matching views are created and pulled into the region. When using this approach, the region instance does not have to be found explicitly by name to create the view and inject it into the region.

Typically, views that host other views have context that needs to be available to its child views. For example, if you have a view to select an employee to show its details, dynamically loaded child views probably need to know which employee is currently selected.

The view injection approach allows pushing views into a region that already exists. This requires creating an instance of the view, getting a reference to the region, and associating the two in the **RegionViewRegistry** using the region's **Add** method.

Typically, view injection is used when explicit control of the views in a region is necessary or when the view to display is determined algorithmically.

View Discovery Approach vs. View Injection Approach

The following are some aspects of the QuickStart that illustrate points to consider when deciding which approach you should use in different situations:

- The view discovery model does not have timing issues. For example, a module can try to add a view to a region that might not be created yet.
- It is simpler to show multiple instances of the same region because you do not need to know about scoped region managers to find the specific region instance to inject your views into.
- You could query the **RegionViewRegistry** class using the **GetContents** method to get all the views associated with a particular region. For example, this list can be bound to a menu.

In view discovery composition, a region is populated as soon as it gets added to the visual tree, so you have less control over when views are added to a particular region. If you want to load a view at a certain time, it will be difficult to achieve this with view discovery composition.

- You should not use view discovery composition if you need scoped region managers, such as to have multiple instances of the same view that contains a region at the same time. Because a region gets registered with a region manager, the name has to be unique.

Registering Views

The **RegionViewRegistry** class in the Prism Library is responsible of registering and retrieving the (region name, view type) pairs. Typically, application modules register their views in their **Initialize** method using a **RegionViewRegistry** instance. The following code example shows the registration of the **EmployeeListView** with the **LeftRegion** in the **Initialize** method of the **ModuleInit** class in the **EmployeeModule** module.

C#

```
this.regionManager.RegisterViewWithRegion(
    RegionNames.LeftRegion,
    () => this.container.Resolve<EmployeeListView>());
```

The **RegisterViewWithRegion** method of the Prism Library's **RegionViewRegistry** class is used to register the region name with its associated view in the registry. There are two ways to access this method:

- From the **RegionViewRegistry** directly.
- From a **RegionManager** instance, because this is an extension method of that class for easy access.

Note: This extension method is on the **RegionManager** for easy access, but it does not register the view with that instance of the region manager only. When a region with the specified name is created, regardless of which scoped region manager is registered, the view will be pulled into it.

The **RegisterViewWithRegion** method has two overloads:

- **RegisterViewWithRegion(string regionName, Type viewType);**
- **RegisterViewWithRegion(string regionName, Func<object> getContentDelegate);**

If you want to register a view directly, use the first overload. If you want to provide a delegate, such as to resolve the presenter that is responsible for creating the view in a "presenter first" or "ViewModel-first" approach, as seen in the earlier **Initialize** method, use the second overload.

When a region is created, it looks for its associated views in the registry. The matching views are pulled and loaded inside the region. If the first overload is used, a new instance of the view is created using the Service Locator.

Sharing Context Between Views

The **RegionContext** attached property is useful when you want to share context from a parent view that hosts a region to its child views. This attached property can hold any simple or complex object.

In the UI Composition QuickStart, the **RegionContext** is used to pass the selected employee ID to the ProjectListView view to obtain the projects the selected employee worked on.

The following code, located in the EmployeesSummaryView.xaml file (in the Views directory of the UIComposition.EmployeeModule project), shows how the **RegionContext** attached property is used in XAML.

XAML

```
<TabControl Grid.Row="1"
           AutomationProperties.AutomationId="EmployeeSummaryTabControl"
           Margin="8"
           regions:RegionManager.RegionName="TabRegion"
           regions:RegionManager.RegionContext="{Binding CurrentEmployee}"
           Width="Auto"
           Height="Auto"
           HorizontalAlignment="Stretch"
           ItemContainerStyle="{StaticResource HeaderStyle}">
</TabControl>
```

To obtain the **RegionContext** in a view, the **GetObservableContext** static method of the **RegionContext** class is used; it passes the view as a parameter and accesses its **Value** property, as shown in the following code example.

C#

```
// EmployeeDetailsView.xaml.cs
employeeDetailsViewModel.CurrentEmployee =
    RegionContext.GetObservableContext(this).Value as Employee;
```

The value of the **RegionContext** can be changed by simply assigning a new value to its **Value** property. You can also subscribe to an event to detect when the **RegionContext** property changes, as shown in the following code example, which subscribes its **PropertyChanged** event to the **RegionContextChanged** event handler.

C#

```
// EmployeeDetailsView.xaml.cs
RegionContext.GetObservableContext(this).PropertyChanged += (s, e)
    =>
    employeeDetailsViewModel.CurrentEmployee =
        RegionContext.GetObservableContext(this).Value
        as Employee;
```

Note: The **DataContext** property is not used to share context because the **DataContext** property is typically used for storing the view model of the view.

Acceptance Tests

The UI Composition QuickStart includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the UI Composition QuickStart acceptance tests

1. In Visual Studio, open the solution file
QuickStarts\UIComposition_Desktop\UIComposition.Tests.AcceptanceTest\UIComposition.Tests.AcceptanceTest.sln.
2. Build the Solution.
3. Open **Test Explorer**.
4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.

Outcome

You should see the QuickStart window and the tests automatically interact with the application. At the end of the test pass, you should see that all tests have passed.

More Information

For more information about UI composition, see [Composing the User Interface](#).

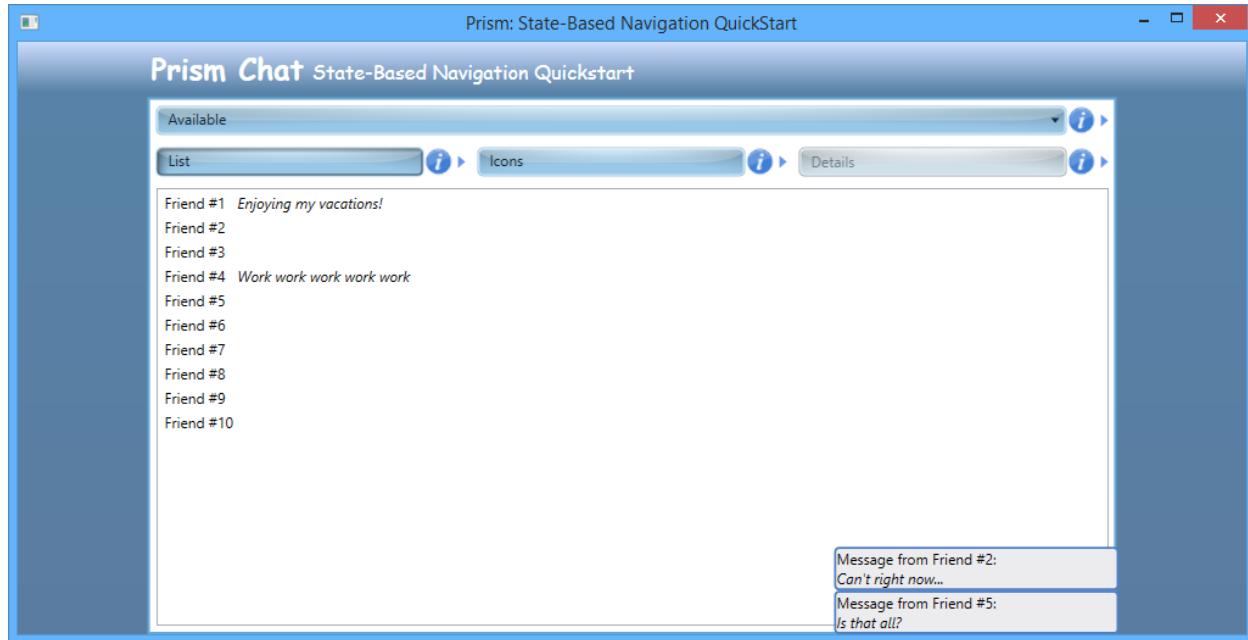
State-Based Navigation QuickStart

The State-Based Navigation QuickStart sample demonstrates navigation using the WPF Visual State Manager (VSM) with the Model-View-ViewModel (MVVM) pattern and the Prism Library. This approach uses the Visual State Manager to define the different application states that the application has, define animations for both the states and the transitions between states; the animations associated to states are active while the state is active for the duration of the specified timeline.

One important aspect of application design is getting the navigation right. To define the navigation of the application, you need to design the screens, interaction, and the visual appearance of the application.

Business Scenario

The main window of the State-Based Navigation QuickStart represents a subset of a chat application. This window shows the list of contacts of the user. The user can alternate among different views of their contacts: list, icons, or contact detail. The messages from the user's contacts are displayed as they arrive. In the detail view of a contact, you can send a message to that contact. The following illustration shows the QuickStart main window.



State-Based Navigation QuickStart user interface

Building and Running the QuickStart

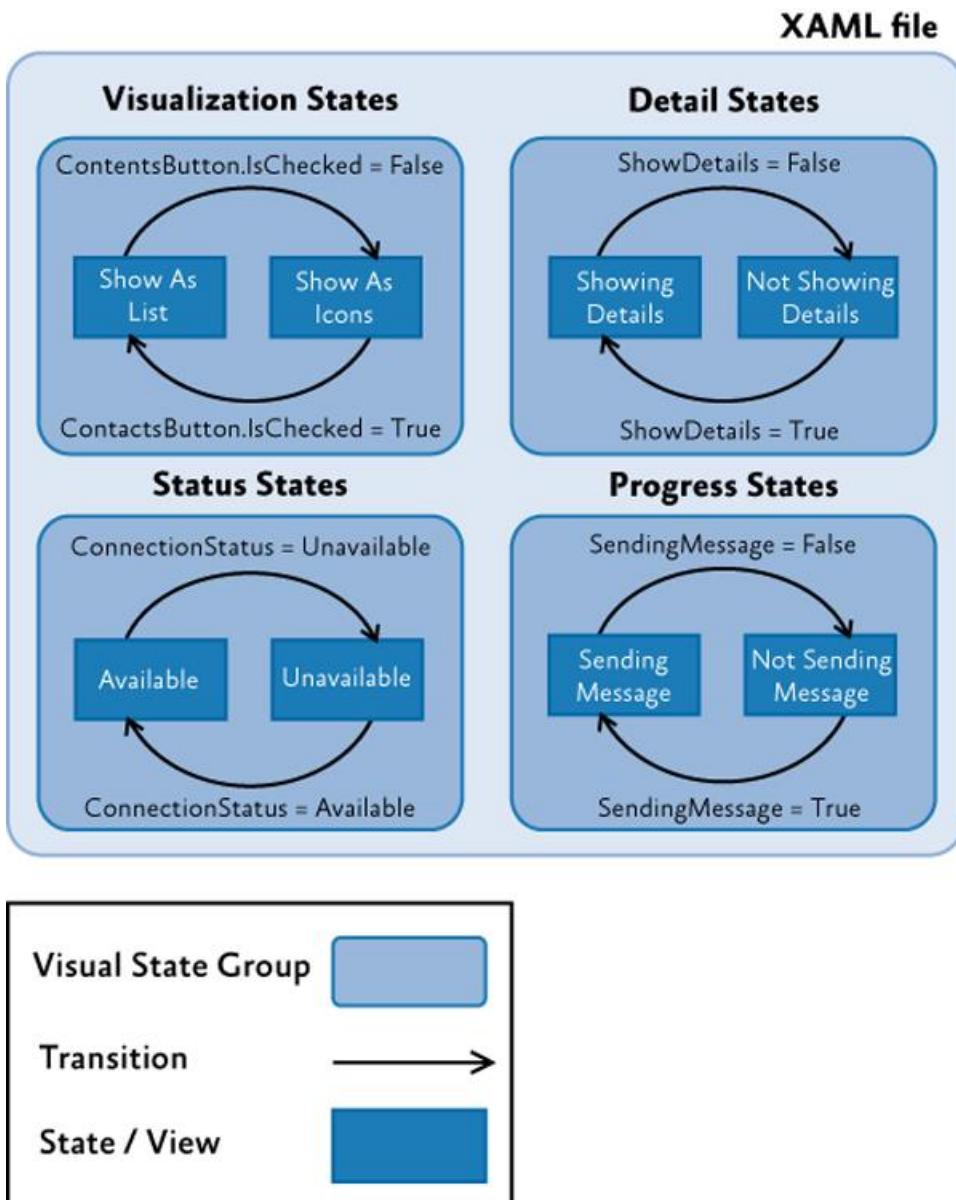
The QuickStart ships as source code—this means you must compile it before you run it. This QuickStart requires Microsoft Visual Studio 2012 or later and the .NET Framework 4.5.1.

To build and run the State-based Navigation QuickStart

1. In Visual Studio, open the solution file Quickstarts\State-Based Navigation/Desktop\State-Based Navigation.sln.
 2. In the **Build** menu, click **Rebuild Solution**.
 3. Press F5 to run the QuickStart.
-

Implementation Details

The QuickStart highlights the key elements and considerations to implement an approach for navigation that uses the VSM. For more information about the VSM, see [VisualStateManager Class](#) on MSDN. In this QuickStart, most of the UI is in a few classes (the **ChatView** and **SendMessagePopupView** classes), and the visual states determine what is shown and how to go from one state to another. Some states change visibility of elements within the view, some states change enablement, and some states activate components. This section describes the key artifacts of the QuickStart, which are shown in the following illustration.



State-Based Navigation QuickStart conceptual view

Notice that the Extensible Application Markup Language (XAML) file contains several states (you can compare states to views) grouped in visual state groups. There can be only one active state in each group. Therefore, the state of the application is a combination of four visual states (one of each visual state group). The different transitions are driven by the view. In the preceding illustration, the conditions represented over each transition arrow are the ones that trigger the transition from one state to another. The definition of the animations associated to the transitions and the behaviors that trigger them is also defined in the view's XAML file.

Note: In the QuickStart, there are only two states per **VisualStateGroup**. This is not mandatory; however, if you have more states, the transition logic could be more complex.

The following illustration shows states of the application and what visual states are active to create them.

Application State		
Active Visual States	ShowAsIcons, NotShowingDetails, Available, NotsendMessage	ShowAsIcons, ShowingDetails, Available, sendMessage

Application states and their active visual states

Logical Views (States)

Typically, the logical views are a form of a UI element that lets users interact with the application. In this application, the logical views are really just states to which the single physical view transitions. A state can involve a single user control or any complex set of user controls. The State-Based Navigation QuickStart has the following states: The list view, the icons view, and the contact view. Additionally, the QuickStart has the send message child view.

Most of these logical view definitions are contained in the Views/ChatViews.xaml file. The following code shows the different logical views within the XAML file.

XAML

```
<ContentControl x:Name="MainPane"
    HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch"
    Grid.Row="2">

    <Grid>
        ...
        <!-- Buttons (shared between all views) -->
        <Grid x:Name="ButtonsPanel" Grid.Row="0">
            ...
            <RadioButton x:Name="ShowAsListButton" ... />
            <RadioButton x:Name="ShowAsIconsButton" ... />
            <Button x:Name="ShowDetailsButton" ... />
        </Grid>

        <!-- Contacts view-->
        <ListBox x:Name="Contacts"
            ItemsSource="{Binding ContactsView}"
            Style="{Binding Source={StaticResource ContactsList}}"
            HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
            Grid.Row="2" Grid.RowSpan="3" Visibility="Collapsed"/>
    </Grid>

```

```

<!-- Avatars view-->
<ListBox x:Name="Avatars"
    ItemsSource="{Binding ContactsView}"
    Style="{Binding Source={StaticResource AvatarsList}}"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
Grid.Row="2" Grid.RowSpan="3" Visibility="Collapsed"
    AutomationProperties.AutomationId="AvatarsView"/>

<!-- Details view -->
<Grid x:Name="Details"
    Background="White" Visibility="Collapsed"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
Grid.RowSpan="5">
    ...
</Grid>

</Grid>
</ContentControl>

```

Notice that the views have their **Visibility** property set to **Collapsed**. This means that the controls that compose each view will not be shown and no space will be reserved for them. In this way, navigation between the different views will consist of all views initially collapsed, and navigating into the associated visual state will trigger an animation that will change their visibility to **Visible** (and the animation for the previous state will be stopped, resulting in the visibility for its associated logical view to be reset to the original **Collapsed** value).

Transitions Between Visual States

Transitions determine how to go from one view to another. The [DataStateBehavior](#) behavior is used to switch between two visual states based on whether a conditional property binding evaluates to **True** or to **False**. With the [DataStateBehavior](#) behavior, you can compare two values. One value comes from a binding. You can declare the other value to compare to explicitly. If the two values are equal, the visual state specified for **True** is activated. If the two values are not equal, the visual state specified for **False** is activated. The following code shows the behaviors defined in the chat view.

XAML

```

<i:Interaction.Behaviors>
    <ei:DataStateBehavior Binding="{Binding ShowDetails}"
        Value="True"
        TrueState="ShowDetails" FalseState="ShowContacts"/>
    <ei:DataStateBehavior Binding="{Binding IsChecked, ElementName=ShowAsListButton}"
        Value="True"
        TrueState="ShowAsList" FalseState="ShowAsIcons"/>
    <ei:DataStateBehavior Binding="{Binding ConnectionStatus}"
        Value="Available"
        TrueState="Available" FalseState="Unavailable"/>
    <ei:DataStateBehavior Binding="{Binding SendingMessage}"
        Value="True"

```

```

    TrueState="SendingMessage"
    FalseState="NotSendingMessage"/>
</i:Interaction.Behaviors>

```

Notice that depending on the value of the bound property, different states are shown. Apart from the contact list, icons, and details view, there are states for enabling or disabling the application (when the service is not available) and for activating or deactivating the busy indicator (when the application is busy).

Typically, an animation is triggered to make the transition smooth from one state to another. When navigating to a different state, the source view is hidden and the target one is shown. The animation associated to states is permanent. The following code example shows the flipping animation that occurs during a transition. The animation associated to transitions is transient.

XAML

```

<VisualStateGroup x:Name="VisualizationStates">

    <VisualStateGroup.Transitions>

        <VisualTransition From="ShowAsIcons" To="ShowAsList">
            <Storyboard SpeedRatio="2">
                ...
                <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Angle"
Storyboard.TargetName="rotate">
                    <EasingDoubleKeyFrame KeyTime="0:0:0" Value="360"/>
                    <EasingDoubleKeyFrame KeyTime="0:0:0.5" Value="270"/>
                    <EasingDoubleKeyFrame KeyTime="0:0:0.5" Value="90"/>
                    <EasingDoubleKeyFrame KeyTime="0:0:1" Value="0"/>
                </DoubleAnimationUsingKeyFrames>

                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="(FrameworkElement.Visibility)"
Storyboard.TargetName="Avatars">
                    <DiscreteObjectKeyFrame KeyTime="0:0:0.5" >
                        <DiscreteObjectKeyFrame.Value>
                            <Visibility>Collapsed</Visibility>
                        </DiscreteObjectKeyFrame.Value>
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>

                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="(FrameworkElement.Visibility)"
Storyboard.TargetName="Contacts">
                    <DiscreteObjectKeyFrame KeyTime="0:0:0.5" >
                        <DiscreteObjectKeyFrame.Value>
                            <Visibility>Visible</Visibility>
                        </DiscreteObjectKeyFrame.Value>
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
        </VisualTransition>
    </VisualStateGroup.Transitions>
</VisualStateGroup>

```

```

        </ObjectAnimationUsingKeyFrames>

    </Storyboard>
</VisualTransition>

<VisualTransition From="ShowAsList" To="ShowAsIcons" ... />

</VisualStateGroup.Transitions>
...
</VisualStateGroup>
```

The states are grouped in different visual state groups. Only one state in a state group can be displayed at a time. For that reason, the **ShowAsList** state (contact list view), and the **ShowAsIcon** state (icons view) are mutually exclusive. The **ShowingDetails** and **NotShowingDetails** states belong to a different group; therefore, the application can be on the **ShowAsIcons** and **ShowingDetails** state at the same time. In this case, the **ShowingDetails** state goes to the foreground and overlaps the icons view; when transitioning to the **NotShowingDetails** states, the details view is collapsed, and the previous view, icons view, is shown. In this way, you do not have to store the previous state for returning because it is active in the background.

Interaction Requests

Interaction requests provide an abstract approach for view models to request interaction with the user. For more information about interaction requests, see [Using Interaction Request Objects in Advanced MVVM Scenarios](#).

The QuickStart uses interaction requests for two different situations: receiving and sending messages:

- **Receiving messages.** The code in the view models create the objects that support the interactions (by raising an event with a payload to communicate with the view) and expose them through properties so they can be consumed by views. In the following code example from the **ChatViewModel** class, notice that the **ShowReceivedMessageRequest** property is defined and then used on the **OnMessageReceived** event handler to raise the **Message** instance.

C#

```

public IInteractionRequest ShowReceivedMessageRequest
{
    get { return this.showReceivedMessageRequest; }
}

private void OnMessageReceived(object sender, MessageReceivedEventArgs a)
{
    this.showReceivedMessageRequest.Raise(a.Message);
}
```

On the view side, it should detect an interaction request and then present an appropriate display for the request. The custom **InteractionRequestTrigger** automatically subscribes to the **Raised** event of the bound **IInteractionRequest**. The following code example, located in the ChatView.xaml file, shows this.

C#

```
<prism:InteractionRequestTrigger SourceObject="{Binding ShowReceivedMessageRequest}">
    <cb:ShowNotificationAction TargetName="NotificationList" />
</prism:InteractionRequestTrigger>
```

In the State-Based Navigation QuickStart, the custom **ShowNotificationAction** class is used to temporarily add the received message to a collection and sets this collection as the **DataContext** of a pop-up window. In this manner, the messages will be displayed in a non-modal window for a determined amount of time before disappearing.

- **Sending messages.** To display the send message window, the **SendMessageRequest** interaction request is used. The **Raise** method of this interaction request is invoked in the **SendMessage** method shown in the following code example from the **ChatViewModel**.

C#

```
public IInteractionRequest SendMessageRequest
{
    get { return this.sendMessageRequest; }
}

public void SendMessage()
{
    var contact = this.CurrentContact;
    SendMessageViewModel viewModel = new SendMessageViewModel();
    viewModel.Title = "Send message to " + contact.Name;

    this.sendMessageRequest.Raise(
        viewModel,
        sendMessage =>
    {
        if (sendMessage.Confirmed)
        {
            this.SendingMessage = true;

            this.chatService.SendMessage(
                contact,
                sendMessage.Message,
                result =>
            {
                this.SendingMessage = false;
            });
    });
}
```

```
    });
}
```

On the view side, when the interaction request is detected, the **PopupWindowAction** displays the **SendMessagePopView** pop-up window, as shown in the following code example from the ChatView.xaml file.

```
C#
<prism:InteractionRequestTrigger SourceObject="{Binding SendMessageRequest}">
    <prism:PopupWindowAction IsModal="True">
        <prism:PopupWindowAction.WindowContent>
            <vs:SendMessagePopView />
        </prism: PopupWindowAction.WindowContent>
    </prism:PopupWindowAction>
</prism:InteractionRequestTrigger>
```

Note that the **IsModal** property of the **PopupWindowAction** action is set to true to specify that this interaction should be modal. To specify the view that will be displayed when the interaction occurs, use the **WindowContent** property.

Chat Service

The chat service is used for retrieving the contacts and their data; it is also used for sending and receiving messages from other users. The following code example shows the service interface.

```
C#
public interface IChatService
{
    event EventHandler ConnectionStatusChanged;
    event EventHandler<MessageReceivedEventArgs> MessageReceived;
    bool Connected { get; set; }
    void GetContacts(Action<IOperationResult<IEnumerable<Contact>>> callback);
    void SendMessage(Contact contact, string message, Action<IOperationResult>
callback);
}
```

The service contains the following members:

- The **Connected** property. This property stores the state of the service: **Connected/Disconnected**.
- The **ConnectionStatusChanged** event handler. This event handler reacts to changes in the connection status.
- The **MessageReceived** event handler. This event handler reacts when a new message is received.
- The **GetContacts** method. This method retrieves the contacts of the user.
- The **SendMessage** method. This method sends messages to the users' contacts.

The service has a timer to simulate incoming messages from other users. On every tick of the timer, there is a 33 percent chance that a message will be received based on a random draw. Additionally, the timer is also used to simulate connection drops; however, the chance of this happening is quite low (1/150). You can see this in the following code example that shows the **OnTimerTick** event handler.

C#

```
private void OnTimerTick(object sender, EventArgs args)
{
    if (this.Connected)
    {
        var coinToss = this.random.Next(3);
        if (coinToss == 0)
        {
            this.ReceiveMessage(
                this.GetRandomMessage(this.random.Next(receivedMessages.Length)),
                this.GetRandomContact(this.random.Next(this.contacts.Count)));
        }
        else
        {
            coinToss = this.random.Next(150);
            if (coinToss == 0)
            {
                this.Connected = false;
            }
        }
    }
}
```

Custom Behaviors

Behaviors are a self-contained unit of functionality. There are two types of behaviors:

- Behaviors that do not have the concept of invocation; instead, it acts more like an add-on to an object.
- Triggers and actions that are closer to the invocation model.

Additional functionality can be easily attached to an object in the XAML or through the designer. They can react to handle an event or a trigger in the UI. The following behaviors are used and defined in the QuickStart (located in the Infrastructure/Behaviors folder):

- **ShowNotificationAction.** This custom behavior allows a view model to push notifications into a target element in the UI. In the QuickStart, it is used to display the chat messages that are received by the user in the lower-right corner of the UI.

The following behaviors are part of the Prism Library Prism.Interactivity project:

- **PopupWindowAction.** This concrete implementation displays a specified window or the default one configured with a data template.
-

Acceptance Tests

The State-Based Navigation QuickStart includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the State-Based Navigation QuickStart acceptance tests

1. In Visual Studio, open the solution file Quickstarts\State-Based Navigation/Desktop\State-Based Navigation.Tests.AcceptanceTest\State-Based Navigation.Tests.AcceptanceTest.sln.
 2. Build the Solution.
 3. Open Test Explorer.
 4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.
-

Outcome

You should see the QuickStart window and the tests automatically interact with the application. At the end of the test run, you should see that all tests have passed.

More Information

To learn about other navigation topics included with Prism, see the following topics:

- [Navigation](#)
 - [View-Switching Navigation QuickStart](#)
-

View-Switching Navigation QuickStart

The View-Switching Navigation QuickStart sample demonstrates how to use the Prism Region Navigation API with the Model-View-ViewModel (MVVM) pattern. The Prism Region Navigation utilizes a Uniform Resource Identifier (URI) approach to switch between views. The QuickStart simulates the navigation of a simple email, contacts, and calendar application. The left region provides navigation to each of the main views. The views demonstrate backward navigation and asynchronous dialog interactions.

The View-Switching Navigation QuickStart is more complex than a typical QuickStart because it demonstrates multiple navigation scenarios. Navigation supports just-in-time view creation, and therefore, interacts with the dependency injection container. Additionally, to be compatible with the Model-View-ViewModel (MVVM) approach, navigation interacts with views and with view models (via the **DataContext** property).

This QuickStart demonstrates the following navigation capabilities:

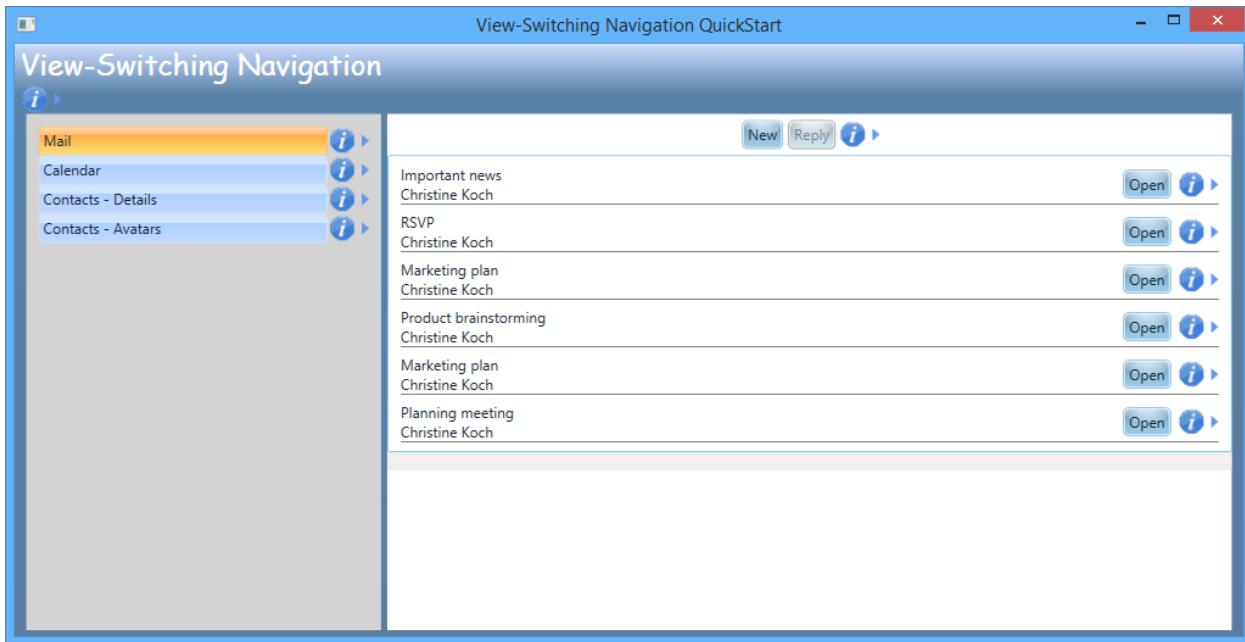
- Navigating to a view in a region
- Navigating to a view in a region contained in another view (nested navigation)
- Navigation journal support
- Just-in-time view creation
- Passing contextual information when navigating to a view
- Views and view models participating in navigation, including confirming or canceling navigation
- Using navigation as part of an application built through modularity and user interface (UI) composition

Business Scenario

The main window of the View-Switching Navigation QuickStart shows a simple email client application. In this window, the navigation pane is located on the left of the screen and provides direct access to the application's features. These features are Mail, Calendar, and Contacts (which has the Details and the Avatars view). In the main pane, the selected feature is shown. Notice that the Mail feature is selected when the application starts. This is coordinated by the shell when the **Email** module is loaded.

Because the modules do not have dependencies between them, they are loaded and initialized in random order. To make sure that the items in the left pane are ordered, a **ViewSortHint** attribute is applied to each of the views. For more information about the **ViewSortHint** attribute, see [The Contacts Module \(Navigation to a Nested View\)](#) section.

The following illustration shows the QuickStart main window.



View-Switching Navigation QuickStart user interface

Note: The UI of the QuickStart has information icons. You can click them to display or hide information and implementation notes about the different pieces of the QuickStart.

Building and Running the QuickStart

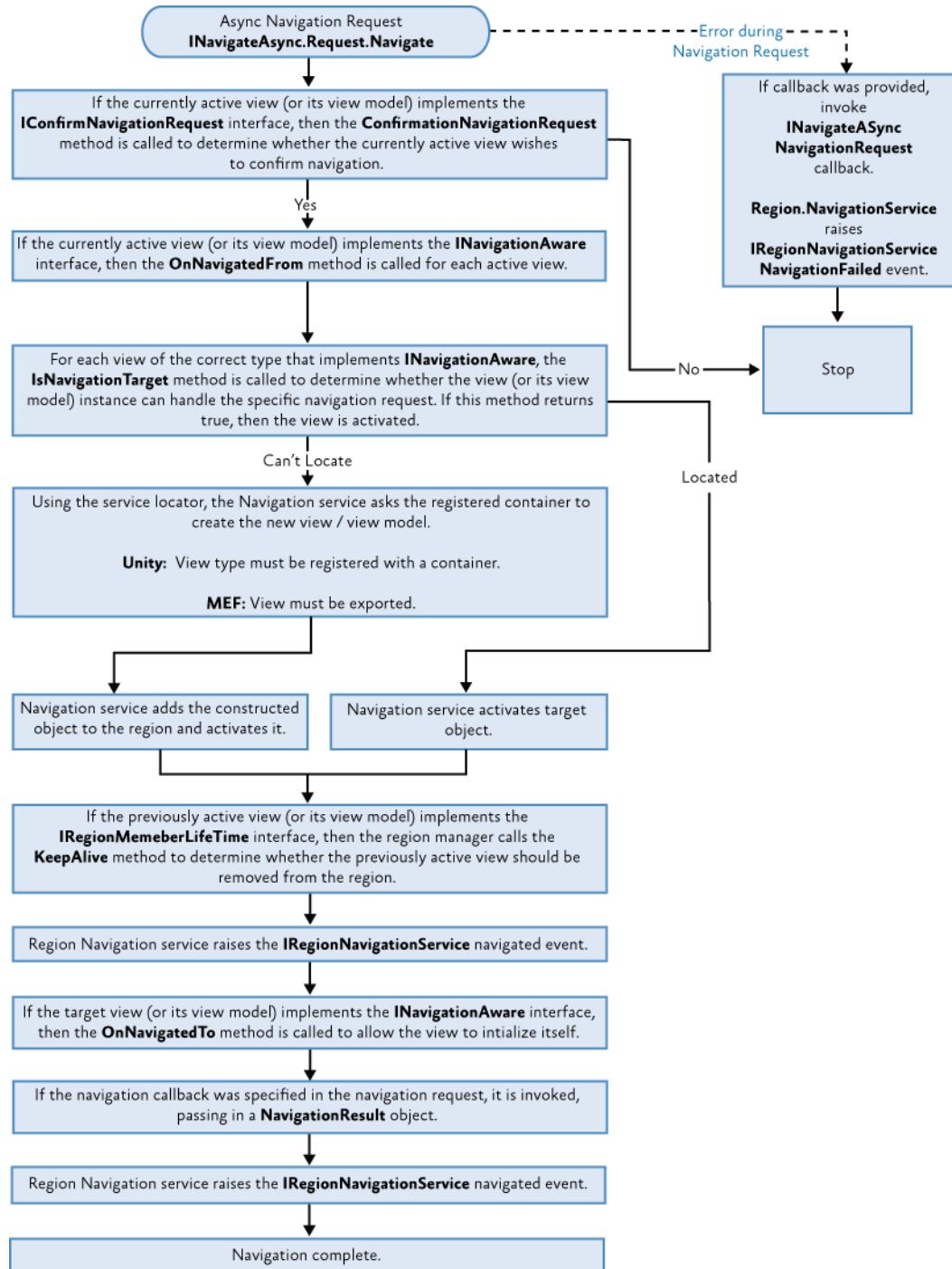
This QuickStart requires Microsoft Visual Studio 2012 or later and the .NET Framework 4.5.1.

To build and run the View-Switching Navigation QuickStart

1. In Visual Studio, open the solution file Quickstarts\View-Switching Navigation_Desktop\ ViewSwitchingNavigation.sln.
2. In the **Build** menu, click **Rebuild Solution**.
3. Press **F5** to run the QuickStart.

Implementation Details

The QuickStart highlights the key elements that you should consider when you use the navigation features provided by the Prism Library to implement navigation in a composite application. In this approach, the user interface is divided among different modules. Each module populates the navigation region on the left side, and participates in navigation to coordinate the view in the main content region on the right side. This section describes the key artifacts of the QuickStart. The following figure shows the workflow that occurs when a user navigates from one location to another.



Prism Region Navigation Workflow

Navigation Support in the Prism Library

The Prism Library supports navigation through the use of *regions*. Navigation classes and components are located in the **Microsoft.Practices.Prism.Regions** namespace. Each region implements the **INavigateAsync** interface, as shown in the following code example.

C#

```
public interface IRegion : INavigateAsync, INotifyPropertyChanged
{
    ...
}
```

The **IRegion** interface provides the **NavigationService** property to allow navigation between regions. Each region has its own **NavigationService**, and each **NavigationService** has its own **Journal**, or record of the current, past, and future navigation within the region. The **NavigationService** returns an **IRegionNavigationService**. The **IRegionNavigationService** interface is shown in the following code example.

C#

```
public interface IRegionNavigationService : INavigateAsync
{
    IRegion Region { get; set; }

    IRegionNavigationJournal Journal { get; }

    event EventHandler<RegionNavigationEventArgs> Navigating;

    event EventHandler<RegionNavigationEventArgs> Navigated;

    event EventHandler<RegionNavigationFailedEventArgs> NavigationFailed;

}
```

The **NavigationService** has a reference to the region to which it belongs and a reference to the navigation journal. Additionally, it contains the **Navigating** and **Navigated** events. The **Navigating** event is triggered during navigation to a page, and the **Navigated** event is raised when the region is navigated to content.

The method from the service that is used to navigate is **RequestNavigate**. This method requires the navigation target and a callback that will be invoked when the navigation is complete.

This method initiates the workflow described in the preceding figure, where it calls the **DoNavigate** method of the service with the source and the callback. It then calls **RequestCanNavigateOnCurrentlyActiveViews**. The callback is invoked when navigation ends, whether or not the navigation is successful. After the callback, the active views and their view models are queried to determine if they implement the **IConfirmNavigationRequest** interface. If they implement this interface, the user will be prompted when he or she navigates away from that view. Finally, the **ExecuteNavigation** method is invoked. This method is shown in the following code example.

C#

```
private void ExecuteNavigation(NavigationContext navigationContext, object[] activeViews, Action<NavigationResult> navigationCallback)
{
```

```

try
{
    NotifyActiveViewsNavigatingFrom(navigationContext, activeViews);

    object view = this.regionNavigationContentLoader.LoadContent(this.Region,
navigationContext);

    // Raise the navigating event just before activating the view.
    this.RaiseNavigating(navigationContext);

    this.Region.Activate(view);

    // Update the navigation journal before notifying others of navigation
    IRegionNavigationJournalEntry journalEntry =
this.serviceLocator.GetInstance<IRegionNavigationJournalEntry>();
    journalEntry.Uri = navigationContext.Uri;
    this.journal.RecordNavigation(journalEntry);

    // The view can be informed of navigation
    InvokeOnNavigationAwareElement(view, (n) =>
n.OnNavigatedTo(navigationContext));

    navigationCallback(new NavigationResult(navigationContext, true));

    // Raise the navigated event when navigation is completed.
    this.RaiseNavigated(navigationContext);
}
catch (Exception e)
{
    this.NotifyNavigationFailed(navigationContext, navigationCallback, e);
}
}

```

The preceding method notifies the active views that the user is navigating away from them, acquires the target view through the content loader, activates the target view, and then updates the journal. The view and the view model are then informed that the user is navigating to them, the callback is invoked, and the navigation completed event is raised.

The Journal

The journal is a stack that maintains the history of the navigated views. It stores the forward, current, and backward history of visited pages. The **RecordNavigation** method is used for registering the current view in the stack. The journal avoids adding a view to the stack if you are internally navigating the journal views.

Note: It is important that you carefully define your application Uniform Resource Identifier (URI) structure before you implement navigation.

Using the Prism Library for Navigation

This section describes how the QuickStart uses the Prism Library to demonstrate navigation. The Shell view has two regions: the navigation region and the main region. You can see the region definition in the following code located in the ViewSwitchingNavigation\Shell.xaml file.

XAML

```
<Grid x:Name="LayoutRoot"
      Background="{StaticResource MainBackground}>
  ...
    <Border Grid.Column="0" Grid.Row="2" Background="LightGray" MinWidth="250"
Margin="5,0,0,5">
      <ItemsControl x:Name="NavigationItemsControl"
prism:RegionManager.RegionName="MainNavigationRegion" Grid.Column="0" Margin="5"
Padding="5" />
      </Border>
      <ContentControl prism:RegionManager.RegionName="MainContentRegion"
                      Grid.Column="1" Grid.Row="2" Margin="5,0,5,5"
HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch"/>
    </Grid>
</Grid>
```

Each QuickStart module (Mail, Calendar, and Contacts) registers the navigation buttons in the navigation region, and the corresponding views (the ones that hold the specific feature) in the main content region.

The shell contains code that ensures that the email view is navigated to when the email module is loaded. Because of this, the email view is shown when the application starts. The section of the following code that appears in bold demonstrates this.

C#

```
[Export]
public partial class Shell : UserControl, IPartImportsSatisfiedNotification
{
    private const string EmailModuleName = "EmailModule";
    private static Uri InboxViewUri = new Uri("/InboxView", UriKind.Relative);
    public Shell()
    {
        InitializeComponent();
    }

    [Import(AllowRecomposition = false)]
    public IModuleManager ModuleManager;

    [Import(AllowRecomposition = false)]
    public IRegionManager RegionManager;

    public void OnImportsSatisfied()
```

```

    {
        this.ModuleManager.LoadModuleCompleted +=
            (s, e) =>
        {
            if (e.ModuleInfo.ModuleName == EmailModuleName)
            {
                this.RegionManager.RequestNavigate(
                    RegionNames.MainContentRegion,
                    InboxViewUri);
            }
        };
    }
}

```

The Calendar Module (Cross-Navigation to Other Modules)

The Calendar module shows cross-navigation to another area/module. The following code example shows the **Initialize** method of the Calendar module.

C#

```

public void Initialize()
{
    this.regionManager.RegisterViewWithRegion(RegionNames.MainNavigationRegion,
typeof(CalendarNavigationView));
}

```

The **Initialize** method registers the **CalendarNavigationView** view, which is the button used for displaying the calendar feature. The **CalendarView** view could have been registered here, but we are using a different mechanism to make the view available to the region (through just-in-time creation during navigation).

The event handler of this button uses the region manager to call the **RequestNavigate** method, passing the name of the region, and the URI to navigate to, as shown in the following code example.

C#

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    this.regionManager.RequestNavigate(RegionNames.MainContentRegion,
calendarViewUri);
}

```

When the navigation is completed—that is, when calendar view is loaded in the main region and the **Navigated** event is triggered—the button is checked to inform the user that the view displayed in the main region is the calendar. This is shown in the following code example. Every module of the QuickStart has similar code to perform these tasks.

C#

```

public void MainContentRegion_Navigated(object sender, RegionNavigationEventArgs e)
{
}

```

```

        this.UpdateNavigationButtonState(e.Uri);
    }

private void UpdateNavigationButtonState(Uri uri)
{
    this.NavigateToCalendarRadioButton.IsChecked = (uri == calendarViewUri);
}

```

Because this QuickStart implements the MVVM pattern, the logic is located in the view model classes (except for the navigation item views). You can construct URLs that use a query string to pass context to a view or view model. For example, in the view model class of the Calendar view, when a user clicks a meeting, a query string is used to identify the email message to display. You can see the building of the URI and the request to navigate in the following code example.

C#

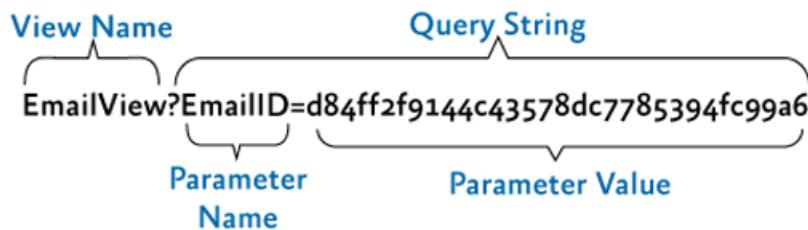
```

private void OpenMeetingEmail(Meeting meeting)
{
    var parameters = new NavigationParameters();
    parameters.Add>EmailIdName,
meeting.EmailId.ToString(GuidNumericFormatSpecifier));

    this.regionManager.RequestNavigate(RegionNames.MainContentRegion, new
Uri>EmailViewName + parameters, Urikind.Relative));
}

```

In the preceding code, using Prism's **NavigationParameters** class, the ID of a specific mail is specified. This class forms query parameters to be added to the queries by taking the name and the value of the parameter. The **ToString** method of this class is overridden to create a query string with all the specified parameters. This example shows how to use the **NavigationParameters** class to pass string parameters using the Query String, but it can also be used to pass object parameters, using an overload of the **RequestNavigate** method. Finally, the query is appended to the name of the view. The result will be similar to that shown in the following illustration.



A complex URI structure

Finally, using the **RequestNavigate** method, the region will navigate to the created Uri.

The Contacts Module (Navigation to a Nested View)

The Contacts module demonstrates navigation to a view nested within another view's region. The views in this module implement the **INavigationAware** interface to participate in the navigation. The contacts view has a region in which a sub-view is displayed.

There are two navigation item views for the contact module: one for displaying contact details and the other to display contact avatars. Each option has a different URI and click event handler, as shown in the following code located in the

`ViewSwitchingNavigation.Contacts\Views\ContactsDetailNavigationItemView.xaml.cs` file.

C#

```
[Export]
[ViewSortHint("03")]
public partial class ContactsDetailNavigationItemView : UserControl,
IPartImportsSatisfiedNotification
{
    private const string mainContentRegionName = "MainContentRegion";

    private static Uri contactsDetailsViewUri = new Uri("ContactsView?Show=Details",
UriKind.Relative);

    ...

    private void NavigateToContactDetailsRadioButton_Click(object sender,
RoutedEventArgs e)
    {
        this.regionManager.RequestNavigate(mainContentRegionName,
contactsDetailsViewUri);
    }
}
```

In the preceding code, the **ViewSortHint** attribute is used to specify the order in which views will be shown. In this case, the **ContactstViewNavigationItem** is placed third in the list. An alphanumeric comparison of the sort hints occurs to determine the order. This is used for placing the navigation buttons of the QuickStart in the same order in every run.

A view or a view model should implement the **INavigationAware** interface when it needs to be notified of navigation activities and so that it can receive the URI query. This interface provides the following navigation events.

- **IsNavigationTarget**. Called to determine if this instance can handle the navigation request.
- **OnNavigatedFrom**. Called when the implementer is being navigated away from.
- **OnNavigatedTo**. Called when the implementer has been navigated to.

The contact view implements the **INavigationAware** interface. When the contact view is navigated to, using any of the navigation buttons or by going back, the **OnNavigatedTo** event is used to determine which sub-view will be loaded, based on the URI query. This can be seen in the following code, extracted from the ViewSwitchingNavigation.Contacts\Views\ContactsView.xaml.cs file.

C#

```
void INavigationAware.OnNavigatedTo(NavigationContext navigationContext)
{
    // Navigating an inner region based on context
    // The ContactsView will navigate an inner region based on the
    // value of the 'Show' query parameter. If the value is 'Avatars'
    // we will navigate to the avatar view otherwise we'll use the details view.
    NavigationParameters parameters = navigationContext.Parameters;
    if (parameters != null && string.Equals(parameters[ShowParameterName].ToString(),
    AvatarsValue))
    {
        regionManager.RequestNavigate(ContactRegionNames.ContactDetailsRegion, new
        Uri(ContactAvatarViewName, UriKind.Relative));
    }
    else
    {
        regionManager.RequestNavigate(ContactRegionNames.ContactDetailsRegion, new
        Uri(ContactDetailViewName, UriKind.Relative));
    }
}
```

In the preceding code, the view that will be loaded in the inner region of the contact view depends on the value of the **Show** parameter of the URI query.

The first time you navigate to any of these views, the specified view will be created by the **NavigationService**.

The Email Module

The Email feature demonstrates navigation to a view that handles additional navigation based on user activity. The view models in this module implement the **INavigationAware** interface to participate in the navigation.

In the Email module, most of the work is performed by the view models. This module is composed of three views: mail list (**InboxView** view), open mail (**EmailView** view), and compose email (**ComposeEmailView** view). The following code example shows the methods that handle the **New**, **Reply**, and **Open** button actions. Notice that they just create a query string and then navigate to the corresponding view in the main region.

C#

```
private void ComposeMessage(object ignored)
{
    this.regionManager.RequestNavigate(RegionNames.MainContentRegion,
    ComposeEmailViewUri);
```

```

}

private void ReplyMessage(object ignored)
{
    var currentEmail = this.Messages.CurrentItem as EmailDocument;
    if (currentEmail != null)
    {
        var parameters = new NavigationParameters();
        parameters.Add(ReplyToKey, currentEmail.Id.ToString("N"));
this.regionManager.RequestNavigate(RegionNames.MainContentRegion, ComposeEmailViewKey
+ parameters);
    }
}

private bool CanReplyMessage(object ignored)
{
    return this.Messages.CurrentItem != null;
}

private void OpenMessage(EmailDocument document)
{
    NavigationParameters parameters = new NavigationParameters();
    parameters.Add>EmailIdKey, document.Id.ToString("N"));
this.regionManager.RequestNavigate(RegionNames.MainContentRegion, new
Uri>EmailViewKey + parameters, UriKind.Relative));
}

```

The **ComposeEmailViewModel** implements the **IConfirmNavigationRequest** interface used for determining whether the view or view model accepts being navigated away from. This interface has the **ConfirmNavigationRequest** method that allows the cancelation of a navigation request. In the compose email screen, the user might start writing a message, but not send it. Therefore, the user should be prompted to confirm that he or she wants to discard the message before navigating away. The following code shows the implementation of the **ConfirmNavigationRequest** method in the **ComposeEmailViewModel** class. The view model uses the **InteractionRequest** to prompt the user, and if the user confirms that he or she wants to navigate away, the navigation continues when **continuationCallback** is passed as a parameter.

Note: You must invoke the **continuationCallback** action or you will halt this current navigation request and no further processing of this request will take place.

C#

```

void IConfirmNavigationRequest.ConfirmNavigationRequest(NavigationContext
navigationContext, Action<bool> continuationCallback)
{
    if (this.sendState == NormalStateKey)
    {
        this.confirmExitInteractionRequest.Raise(

```

```

        new Confirmation { Content =
Resources.ConfirmNavigateAwayFromEmailMessage, Title =
Resources.ConfirmNavigateAwayFromEmailTitle },
        c => { continuationCallback(c.Confirmed); });
}
else
{
    continuationCallback(true);
}
}

```

In the **ComposeEmailViewModel** class, the **OnNavigatedTo** method is used to determine if the user is composing a new email message or replying to an existing one. The navigation context offers the context information through the **Parameters** property, which is a string/object dictionary built from the parameters passed in the **RequestNavigate** method or through the navigation URI. In the following code example, if the **ReplyTo** parameter contains a value, the relevant information from the email service will be retrieved to populate the response values. If not, an empty email will be displayed.

C#

```

void INavigationAware.OnNavigatedTo(NavigationContext navigationContext)
{
    var emailDocument = new EmailDocument();
    var parameters = navigationContext.Parameters;
    var replyTo = parameters[ReplyToParameterKey];
    Guid replyToId;
    if (replyTo != null && Guid.TryParse(replyTo, out replyToId))
    {
        var replyToEmail = this.emailService.GetEmailDocument(replyToId);
        if (replyToEmail != null)
        {
            emailDocument.To = replyToEmail.From;
            emailDocument.Subject = Resources.ResponseMessagePrefix +
replyToEmail.Subject;

            emailDocument.Text =
                Environment.NewLine +
                replyToEmail.Text
                    .Split(Environment.NewLine.ToCharArray())
                    .Select(l => l.Length > 0 ? Resources.ResponseLinePrefix + l : l)
                    .Aggregate((l1, l2) => l1 + Environment.NewLine + l2);
        }
    }
    else
    {
        var to = parameters[ToParameterKey];
        if (to != null)
        {
            emailDocument.To = to;
        }
    }
}

```

```

        }
    }

    this.EmailDocument = emailDocument;
    this.navigationJournal = navigationContext.NavigationService.Journal;
}

```

Notice that navigation journal is instantiated at the end of the preceding code. The journal is used to navigate to the previous view. The journal provides the **GoBack** method for navigating backwards in the navigating history. This is used after the user sends or cancels the composition of an email message. You can see this method's usage in the following code example.

C#

```

private void CancelEmail()
{
    if (this.navigationJournal != null)
    {
        this.navigationJournal.GoBack();
    }
}

```

Unit and Acceptance Tests

The View-Switching Navigation QuickStart includes unit tests within the solution. Unit tests verify if individual units of source code work as expected.

To run the View-Switching Navigation QuickStart unit tests

1. Build the Solution.
2. Open **Test Explorer**.
3. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the unit tests.

The View Switching Navigation QuickStart includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a prescribed series of steps. You can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the View-Switching Navigation QuickStart acceptance tests

1. In Visual Studio, open the solution file QuickStarts\View-Switching Navigation/Desktop\ViewSwitchingNavigation.Tests.AcceptanceTest\ViewSwitchingNavigation.Tests.AcceptanceTest.sln.
2. Build the Solution.
3. Open **Test Explorer**.

4. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.
-

Outcome

You should see the QuickStart window and the tests automatically interact with the application. At the end of the test run, you should see that all tests have passed.

More Information

To learn about other aspects of navigation in Prism, see the following topics:

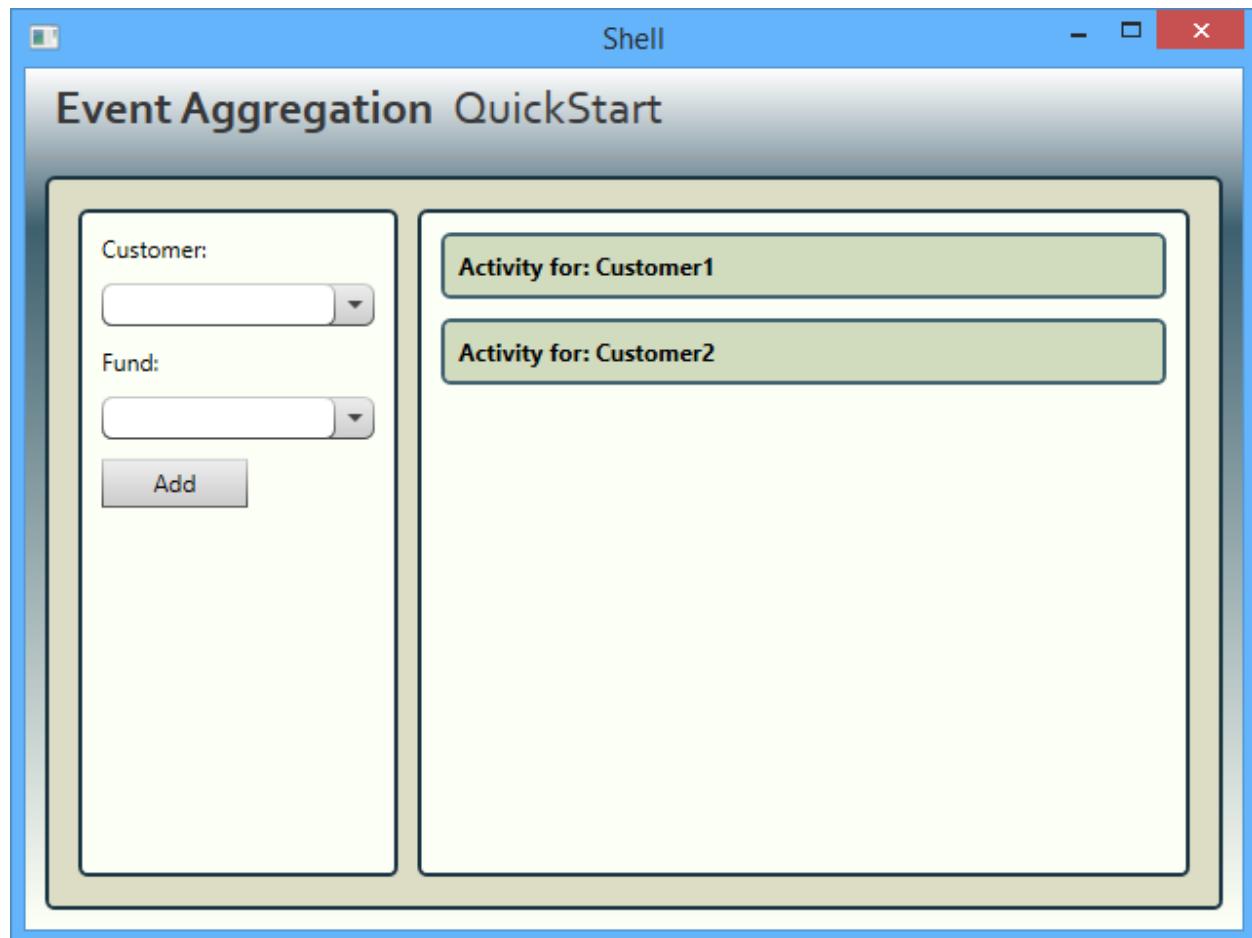
- [Navigation](#)
 - [State-Based Navigation QuickStart](#)
-

Event Aggregation QuickStart

The Event Aggregation QuickStart sample demonstrates how to build a composite application that uses the Prism Library's Event Aggregator service. This service enables you to establish loosely coupled communications between components in your application. The Event Aggregator is a Portable Class Library (PCL) so it can be used on WPF, Windows Phone 8, and Windows Store apps.

Business Scenario

The main window of the Event Aggregation QuickStart represents a subset of a fictitious financial system. In this window, users can add funds to customers and see the activity log for each customer. The following illustration shows the QuickStart main window.



Event Aggregation QuickStart user interface

Building and Running the QuickStart

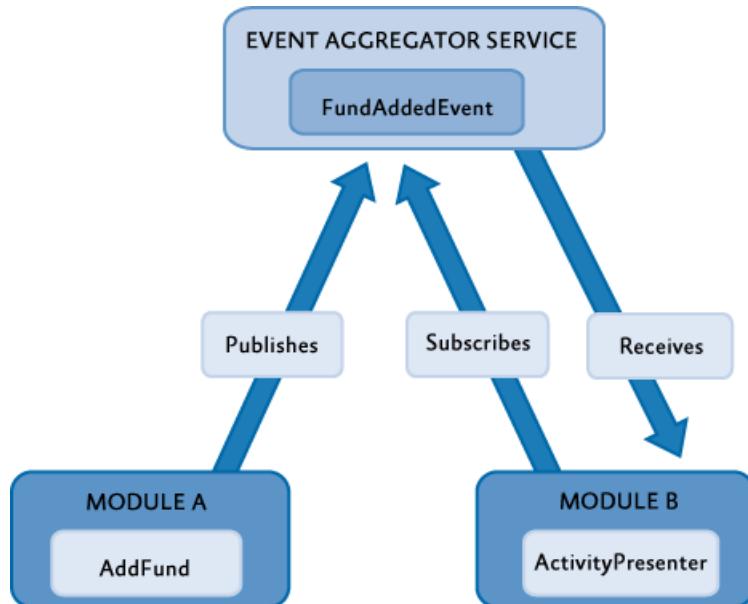
This QuickStart requires Visual Studio 2012 or later and the .NET Framework 4.5.1 to run.

To build and run the Event Aggregation QuickStart

1. In Visual Studio, open the solution file
Quickstarts\EventAggregation\EventAggregation_Desktop.sln.
2. On the **Build** menu, click **Rebuild Solution**.
3. Press **F5** to run the **QuickStart**.

Implementation Details

The QuickStart highlights the key elements that interact when using the Event Aggregator service. This section describes the key artifacts of the QuickStart, which are shown in the following illustration.



Event Aggregation QuickStart conceptual view

The FundAddedEvent Event

The **FundAddedEvent** event is raised when the user adds a fund for a customer. This event is used by the modules ModuleA and ModuleB to communicate in a loosely coupled way. The following code shows the event class signature; the class extends the **PubSubEvent<TPayload>** class, specifying **FundOrder** as the payload type. This code is located at `EventAggregation.Infrastructure.Dektop\FundAddedEvent.cs`.

C#

```

public class FundAddedEvent : PubSubEvent<FundOrder>
{
}
  
```

The following code is the class definition for the **FundOrder** class; this class represents a fund order and specifies the ticker symbol and the customer's identifier. This code is located at `EventAggregation.Infrastructure.Desktop\FundOrder.cs`.

C#

```
public class FundOrder
{
    public string CustomerId { get; set; }
    public string TickerSymbol { get; set; }
}
```

Event Publishing

When the user adds a fund for a customer, the event **FundAddedEvent** is published by the **AddFundPresenter** class (located at ModuleA/Desktop/AddFundPresenter.cs). The following code shows how the **FundAddedEvent** is published.

C#

```
void AddFund(object sender, EventArgs e)
{
    FundOrder fundOrder = new FundOrder();
    fundOrder.CustomerId = View.Customer;
    fundOrder.TickerSymbol = View.Fund;

    if (!string.IsNullOrEmpty(fundOrder.CustomerID) &&
!string.IsNullOrEmpty(fundOrder.TickerSymbol))
        eventAggregator.GetEvent<FundAddedEvent>().Publish(fundOrder);
}
```

In the preceding code, first a **FundOrder** instance is created and set up. Then, the **FundAddedEvent** is retrieved from the Event Aggregator service and the **Publish** method is invoked on it; this supplies the recently created **FundOrder** instance as the **FundAddedEvent** event's parameter.

Event Subscription

The ModuleB module contains a view named **ActivityView**. An instance of this view shows the activity log for a single customer. The ModuleB initializer class creates two instances of this view, one for Customer1 and one for Customer2, as shown in the following code (this code is located at ModuleB/Desktop\ModuleB.cs).

C#

```
public void Initialize()
{
    ActivityView activityView1 = Container.Resolve<ActivityView>();
    ActivityView activityView2 = Container.Resolve<ActivityView>();

    activityView1.CustomerId = "Customer1";
    activityView2.CustomerId = "Customer2";

    IRegion rightRegion = RegionManager.Regions["RightRegion"];
    rightRegion.Add(activityView1);
    rightRegion.Add(activityView2);
}
```

When an instance of the **ActivityView** view is created, its presenter subscribes an event handler to the **FundAddedEvent** event using a filter expression. This filter expression defines a condition that the event's argument must meet for the event handler to be invoked. In this case, the condition is satisfied if the fund order corresponds to the customer associated to the view. The event handler contains code to display the new fund added to the customer in the user interface.

The following code shows the **CustomerId** property of the **ActivityPresenter** class. In the property setter, an event handler for the **FundAddedEvent** event is subscribed using the Event Aggregator service.

C#

```
public string CustomerId
{
    get { return _customerId; }
    set
    {
        _customerId = value;

        FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>();

        if (subscriptionToken != null)
        {
            fundAddedEvent.Unsubscribe(subscriptionToken);
        }

        subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler,
ThreadOption.UIThread, false, FundOrderFilter);

        View.Title = string.Format(CultureInfo.CurrentCulture,
Resources.ActivityTitle, CustomerId);
    }
}
```

The following line, extracted from the preceding code, shows how the event handler is subscribed to the **FundAddedEvent** event.

C#

```
subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler,
ThreadOption.UIThread, false, FundOrderFilter);
```

In the preceding line, the following parameters are passed to configure the subscription:

- The **FundAddedEventHandler** action. This event handler is executed when the **Add** button is clicked and the filter condition is satisfied.
- The **ThreadOption.UIThread** option. This option specifies that the event handler will run on the user interface thread.

- The **KeepSubscriberReferenceAlive** flag. This flag is **false** and indicates that the lifetime of the subscriber's reference is not managed by the event. This is set to **false** because the lifetime of the subscriber, the presenter class, is managed by its view, which contains a reference to it.
- The **filter** predicate. This filter is a condition that specifies that the event handler is invoked only when the fund is added to the view's corresponding customer.

Unit and Acceptance Tests

The Event Aggregator QuickStart includes unit tests within the solution. Unit tests verify if individual units of source code work as expected.

Unit Tests

To run the Event Aggregator QuickStart unit tests

- On the **Test** menu of Visual Studio, point to **Run**, and then click **All Tests**.

Outcome

You should see the Test Results pane in Visual Studio indicating that all the unit tests passed.

Acceptance Tests

The Event Aggregator QuickStart includes a separate solution that includes acceptance tests. The acceptance tests describe how the application should perform when you follow a series of steps; you can use the acceptance tests to explore the functional behavior of the application in a variety of scenarios.

To run the Event Aggregator QuickStart acceptance tests

1. In Visual Studio, open the solution file
QuickStarts\EventAggregation\EventAggregation.Tests.AcceptanceTest\EventAggregation.Tests.AcceptanceTest.sln.
2. Open **Test Explorer**.
3. After building the solution, Visual Studio finds the tests. Click the **Run All** button to run the acceptance tests.

Outcome

You should see the QuickStart window and the tests automatically interact with the application. At the end of the test run, you should see that all tests have passed.

More Information

For more information about event aggregation, see [Communicating Between Loosely Coupled Components](#).

17: Getting Started Using the Prism Library Hands-on Lab

In this lab and associated sample, you will learn the basic concepts of modular application development using the Prism Library, and apply them to create a solution that you can use as the starting point for building a composite Windows Presentation Foundation (WPF) application. After completing this lab, you will be able to do the following:

- You will create a new solution based on the Prism Library.
- You will create and load a module.
- You will create a view and show it in the shell window.

System Requirements

This guidance was designed to run on the Microsoft Windows 8, Windows 7, Windows Vista, Windows Server 2012, or Windows Server 2008 operating system. WPF applications built using this guidance require the .NET Framework 4.5.

Before you can use the Prism Library, the following must be installed:

- Microsoft Visual Studio 2013 Professional, Premium, or Ultimate editions
- Microsoft .NET Framework 4.5 (installed with Visual Studio 2013)
- Optional tool:
 - [Microsoft Blend for Visual Studio 2013](#)

Preparation

This topic requires you to have the following Prism Library and Unity Application Block (Unity) assemblies which can be downloaded from NuGet:

- [Prism](#)
- [Prism.UnityExtensions](#)

Note: This hands-on lab uses the Unity container, but you can also use the Managed Extensibility Framework (MEF) with the Prism Library.

This hands-on lab assumes that you understand Prism basic concepts. For more information, see [Prism Key Concepts](#) in [Introduction](#).

Procedures

This lab includes the following tasks:

- Task 1: Creating a Solution Using the Prism Library
- Task 2: Adding a Module
- Task 3: Adding a View

The next sections describe each of these tasks.

Note: The instructions for this hands-on lab are based on the HelloWorld solution. To open the solution in Visual Studio, run the file Desktop only - Open QS - Hello World QuickStart.lnk.

Task 1: Creating a Solution Using the Prism Library

In this task, you will create a solution using the Prism Library. You will be able to use this solution as a starting point for your composite WPF application. The solution includes recommended practices and techniques and is the basis for the procedures in Prism. To create a solution with the Prism Library, the following tasks must be performed:

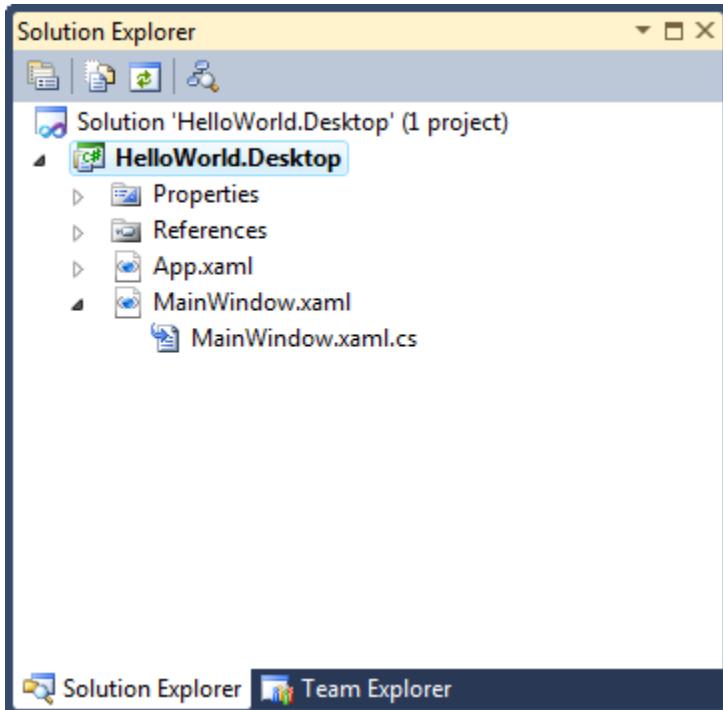
1. **Create a solution with a shell project.** In this task, you create the initial Visual Studio solution and add a WPF Application project that is the basis of solutions built using Prism Library. This project is known as the shell project.
2. **Set up the shell window.** In this task, you set up a window, the shell window, to host different user interface (UI) components in a decoupled way.
3. **Set up the application's bootstrapper.** In this task, you set up code that initializes the application.

The following procedure describes how to create a solution with a shell project. A shell project is the basis of a typical application built using the Prism Library—it is a WPF Application project that contains the application's startup code, known as the bootstrapper, and a main window where views are typically displayed (the shell window).

To create a solution with a shell project

1. In Visual Studio, create a new WPF application. To do this, point to **New** on the **File** menu, and then click **Project**. In the **Project types** list, select **Windows** inside the **Visual C# node**. In the **Templates** box, click **WPF Application**. Finally, set the project's name to **HelloWorld.Desktop**, specify a valid location, and then click **OK**.

Visual Studio will create the HelloWorld project, as shown in the following illustration. This project will be the shell project of your application.



HelloWorld project

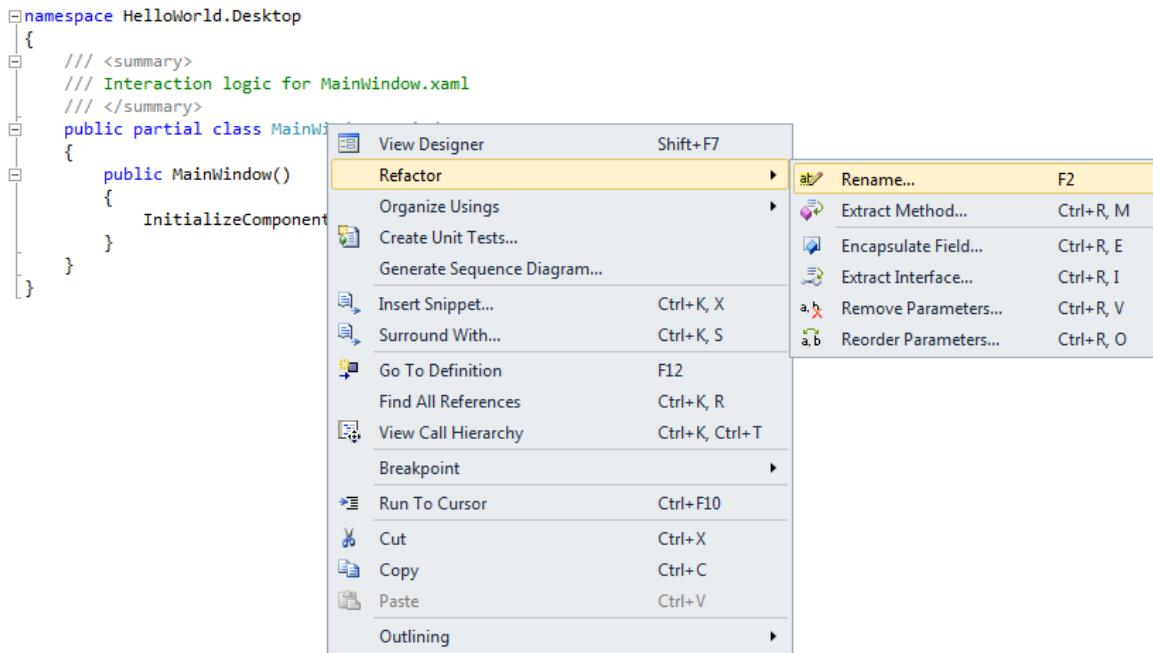
2. In the HelloWorld Project, add references to the following NuGet packages:
 - **Prism.** In the NuGet Package Manager search for Prism. This package contains the implementation of the Prism Library composition components such as modularity, logging services, communication services, and definitions for several core interfaces. It also contains the implementation of Prism Library components that target WPF applications, including regions, events, and MVVM.
 - **Prism.UnityExtensions.** In the NuGet Package Manager search for Prism.UnityExtensions. This package includes the Microsoft.Practices.UnityExtensions assembly which contains utility classes you can reuse in applications built with the Prism Library that consume the [Unity Application Block](#). For example, it contains a bootstrapper base class, the **UnityBootstrapper** class, that creates and configures a Unity container with default Prism Library services when the application starts.

The shell window is the top-level window of an application based on the Prism Library. This window is a place to host different UI components that exposes a way for itself to be dynamically populated by others, and it may also contain common UI elements, such as menus and toolbars. The shell window sets the overall appearance of the application.

The following procedure explains how to set up the shell window.

To set up the shell window

1. In Solution Explorer, rename the file MainWindow.xaml to Shell.xaml.
2. Open the code-behind file Shell.xaml.cs and rename the **MainWindow** class to **Shell** using the Visual Studio refactoring tools. To do this, right-click **MainWindow** in the class signature, point to **Refactor**, and then click **Rename**, as shown in the following illustration. In the **Rename** dialog box, type **Shell** as the new name, and then click **OK**. If the **Preview Changes – Rename** dialog box appears, click **Apply**.



MainWindow renaming using Visual Studio refactoring tools

3. In XAML view, open the Shell.xaml file, and then set the following attribute values to the **Window** root element:
 - **x:Class = "HelloWorld.Desktop.Shell"** (this matches the code behind class's name)
 - **Title = "Hello World"**

Your code should look like the following.

XAML

```

<Window x:Class="HelloWorld.Desktop.Shell"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Hello World" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>

```

Regions

The following procedure describes how to add an **ItemsControl** control to the shell window and associate a region to it. In a subsequent task, you will dynamically add a view to this region.

To add a region to the shell window

1. In the Shell.xaml file, add the following namespace definition to the root **Window** element. You need this namespace to use an attached property for regions that is defined in the Prism Library.

XAML

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

2. Replace the **Grid** control in the shell window with an **ItemsControl** control named **MainRegion**, as shown in the following code.

XAML

```
<Window x:Class="HelloWorld.Desktop.Shell"
    xmlns:prism="http://www.codeplex.com/prism"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hello World" Height="300" Width="300">

    <ItemsControl Name="MainRegion"/>

</Window>
```

The following illustration shows the shell window in the Design view.



Shell window with an ItemsControl control

3. In the **ItemsControl** control definition, set the attached property **prism:RegionManager.RegionName** to "MainRegion", as shown in the following code. This attached property indicates that a region named MainRegion is associated to the control.

XAML

```
<ItemsControl Name="MainRegion" prism:RegionManager.RegionName="MainRegion"/>
```

Note: When the shell window is instantiated, WPF resolves the value of the **prism:RegionManager.RegionName** attached property and invokes a callback in the **RegionManager** class. This callback creates a region and associates it with the **ItemsControl** control.

Bootstrapper

The bootstrapper is responsible for the initialization of an application built using the Prism Library. The Prism Library includes **UnityBootstrapper** and **MefBootstrapper** classes, which implement most of the functionality necessary to use either Unity or MEF as the container in your application. If you are using a container other than Unity or MEF, you should write your own container-specific bootstrapper.

The following procedure explains how to set up the application's bootstrapper.

To set up the application's bootstrapper

1. Add a new class file named **Bootstrapper.cs** to the **HelloWorld** project.
2. Add the following **using** statements at the top of the file. You will use them to refer to elements referenced in the **UnityBootstrapper** class.

C#

```
using System.Windows;
using Microsoft.Practices.Prism.Modularity;
using Microsoft.Practices.Prism.UnityExtensions;
using Microsoft.Practices.Unity;
```

3. Update the **Bootstrapper** class's signature to inherit from the **UnityBootstrapper** class.

C#

```
class Bootstrapper : UnityBootstrapper
{
}
```

4. Override the **CreateShell** method in the **Bootstrapper** class. In this method, create an instance of the shell window and return it, as shown in the following code.

C#

```
protected override DependencyObject CreateShell()
{
    return new Shell();
```

}

Note: You return the shell object to have the **UnityBootstrapper** base class attach an instance of the region manager service to it. The region manager service is a service included in the Prism Library that manages regions in the application. By having a region manager instance attached to the shell window, you can declaratively register regions from XAML code that will exist in the scope of the shell window and child views.

5. Override the **InitializeShell** method in the **Bootstrapper** class. In this method, display the shell to the user.

C#

```
protected override void InitializeShell()
{
    base.InitializeShell();

    Application.Current.MainWindow = (Window)this.Shell;
    Application.Current.MainWindow.Show();
}
```

6. Override the **ConfigureModuleCatalog** method. In this template method, you populate the module catalog with modules. The module catalog interface is **Microsoft.Practices.Prism.Modularity.IModuleCatalog**, and it contains metadata for all the modules in the application. Because the application contains no modules at this point, the implementation of the **ConfigureModuleCatalog** method should simply call the base implementation and return. You can paste the following code in your **Bootstrapper** class to implement the method.

C#

```
protected override void ConfigureModuleCatalog()
{
    base.ConfigureModuleCatalog();
}
```

More details about module loading and module catalogs are described [Task 2: Adding a Module](#) later in this topic.

7. Open the file App.xaml.cs and initialize the Bootstrapper in the handler for the **Startup** event of the application, as shown in the following code. By doing this, the bootstrapper code will be executed when the application starts.

C#

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        Bootstrapper bootstrapper = new Bootstrapper();
```

```
        bootstrapper.Run();
    }
}
```

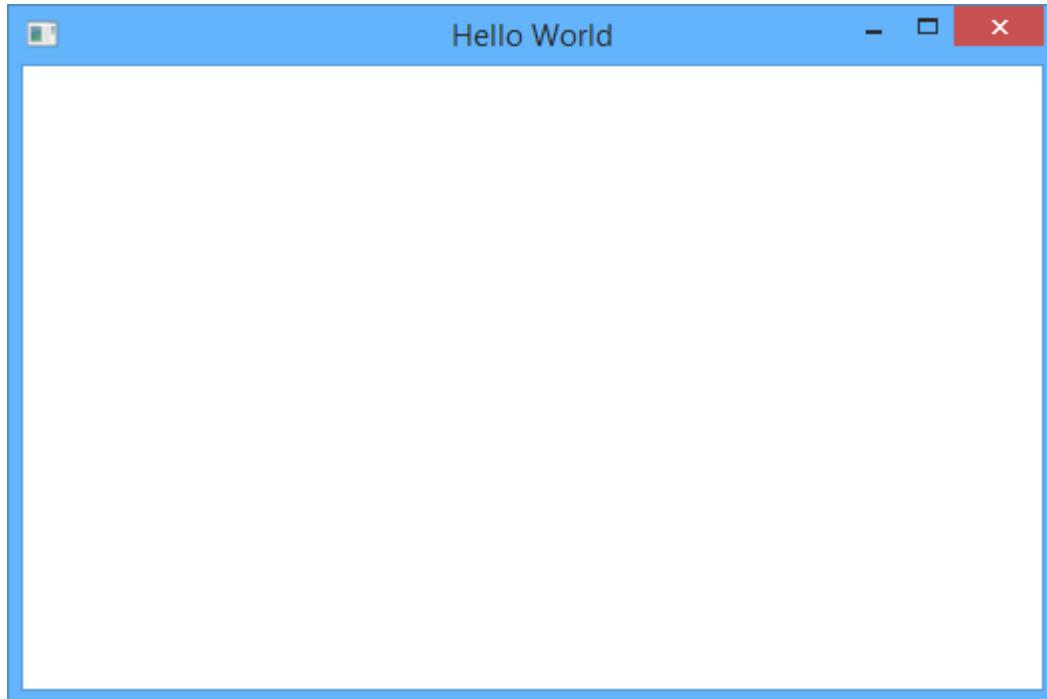
8. Open the App.xaml file and remove the attribute **StartupUri**. Because you are manually instantiating the shell window in your bootstrapper, this attribute is not required. The code in the App.xaml file should look like the following.

C#

```
<Application x:Class="HelloWorld.Desktop.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

9. Build and run the application. You should see an empty Hello World window, as shown in the following illustration.



Hello World window

Task 2: Adding a Module

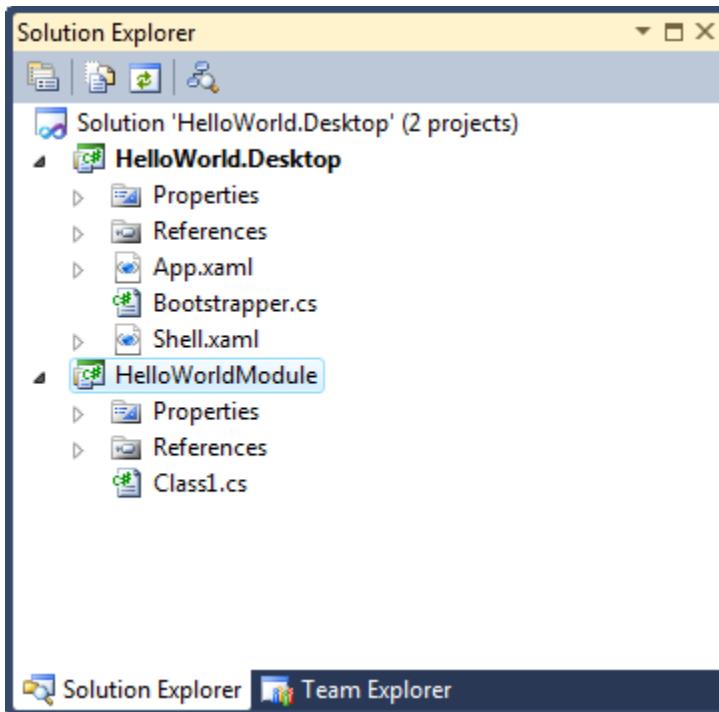
In this task, you will create a module and add it to your solution. A module in Prism is a logical unit in your application. Adding a module to your solution involves the following tasks:

1. **Creating a module.** In this task, you create a module project with a module class.
2. **Configuring how the module is loaded.** In this task, you configure your application to load the module.

The following procedure describes how to create a module.

To create a module

1. Add a new class library project to your solution. To do this, right-click the **HelloWorld.Desktop** solution node in Solution Explorer, point to **Add**, and then click **New Project**. In the **Project types** list, select **Windows** in the **Visual C#** node. In the **Templates** box, click **Class Library**. Finally, set the project's name to **HelloWorldModule**, and then click **OK**. The following illustration shows your solution.



Solution with a module named HelloWorldModule

2. Add references in your module to the following WPF assemblies. To do this, right-click the **HelloWorldModule** project in Solution Explorer, and then click **Add Reference**. In the **Add Reference** dialog box, select the **Assemblies** tab, then select the following assemblies, and then click **OK**:
 - **PresentationCore.dll**

- **PresentationFramework.dll**
 - **WindowsBase.dll**
 - **System.Xaml.dll**
3. Add references in your module to the following Prism Library assemblies. To do this, right-click the **HelloWorld.Desktop** solution in Solution Explorer, and then click **Manage NuGet Packages for Solution**. Click the **Installed Packages** button, select the following assemblies, and then click **Manage**:
- **Prism**
- In the **Selected Projects** dialog, select **HelloWorldModule**, and then click **OK**. Finally, close the Manage NuGet Packages window by clicking **Close**.
4. Rename the **Class1.cs** file to **HelloWorldModule.cs**. To do this, right-click the file in Solution Explorer, click **Rename**, type the new name, and then press ENTER. In the dialog box that asks if you want to perform a rename of all references to your class, click **Yes**.
 5. Open the file **HelloWorldModule.cs** and add the following **using** statement at the top of the file. You will use it to refer to modularity elements provided by the Prism Library.

C#

```
using Microsoft.Practices.Prism.Modularity;
```

6. Change the class signature to implement the **IModule** interface, as shown in the following code.

C#

```
public class HelloWorldModule : IModule
{
}
```

7. In the **HelloWorldModule** class, add an empty definition of the **Initialize** method, as shown in the following code.

C#

```
public void Initialize()
{
}
```

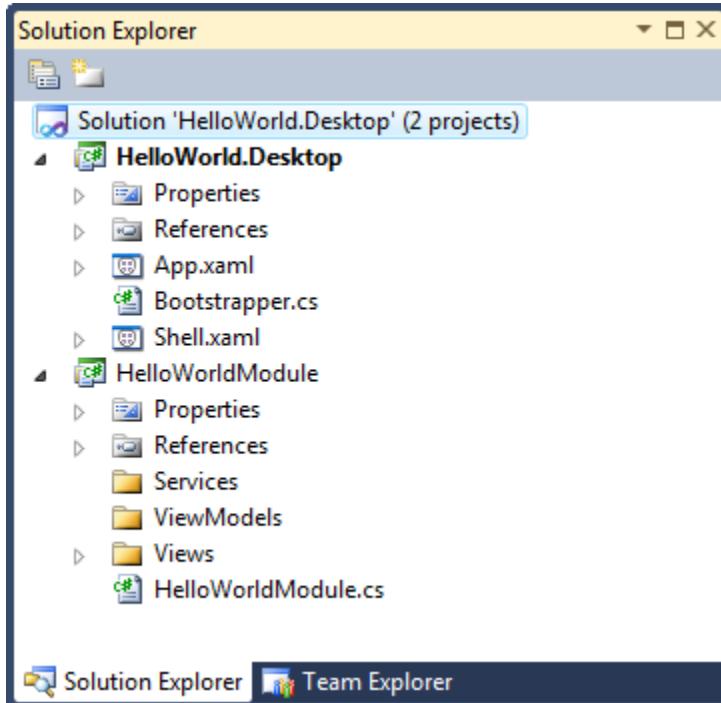
8. Add a Views folder to the **HelloWorldModule** project. In this folder, you will store your view implementations. To do this, right-click the **HelloWorldModule** project in Solution Explorer, point to **Add**, and then click **New Folder**. Change the folder name to **Views**.

This step is recommended to organize your projects; this is useful when a module contains several artifacts. The following are other common folders that you can add to your module:

- **Services**. In this folder, you store service implementations and service interfaces.

- **ViewModels.** In this folder, you store view models.

The following illustration shows the solution with the **HelloWorldModule** module.



Solution with the HelloWorldModule

9. Build the solution.

At this point, you have a solution based on the Prism Library with a module. However, the module is not being loaded into the application. The following section describes module loading and how you can load modules with the Prism Library.

Module in the Application Life Cycle

Modules go through a three-step process during application startup:

1. Modules are discovered by the module catalog. The module catalog contains a collection of metadata about those modules. This metadata can be consumed by the module manager service.
2. The module manager service coordinates the modules initialization. It manages the retrieval and the subsequent initialization of the modules. It loads modules—retrieving them if necessary—and validates them.
3. Finally, the module manager instantiates the module and calls the module's **Initialize** method.

Populating the Module Catalog

The Prism Library provides several ways to populate the module catalog. In WPF, you can populate the module catalog from code, from a XAML file, from a configuration file, or from a directory. The following procedure explains how to populate the catalog from code to load the **HelloWorldModule** module into the **HelloWorld.Desktop** application.

To populate the module catalog with the **HelloWorld** module from code

1. In your shell project, add a reference to the module project. To do this in Solution Explorer, right-click the **HelloWorld.Desktop** project, and then click **Add Reference**. In the **Reference Manager** dialog box, click the **Solution** tab, select the **HelloWorldModule** project, and then click **OK**.
2. Open the **Bootstrapper.cs** file and explore the **ConfigureModuleCatalog** method. The method implementation is shown in the following code.

C#

```
protected override void ConfigureModuleCatalog()
{
    base.ConfigureModuleCatalog();
}
```

The **ModuleCatalog** class is used to define the application's modules from code—it implements the methods included in the **IModuleCatalog** interface and adds an **AddModule** method for developers to manually register modules that should be loaded in the application. The signature of this method is shown in the following code.

C#

```
public ModuleCatalog AddModule(Type moduleType, InitializationMode
initializationMode, params string[] dependsOn)
```

The **AddModule** method returns the same module catalog instance and takes the following parameters:

- **The module initializer class's type of module to load.** This type must implement the **IModule** interface.
 - **The Initialization mode.** This parameter indicates how the module will be initialized. The possible values are **InitializationMode.WhenAvailable** and **InitializationMode.OnDemand**.
 - **An array containing the names of the modules that the module depends on, if any.** These modules will be loaded before your module to ensure your module dependencies are available when it is loaded.
3. Update the **ConfigureModuleCatalog** method to register the **HelloWorldModule** module with the module catalog instance. To do this, you can replace the **ConfigureModuleCatalog** implementation with the following code.

C#

```
protected override void ConfigureModuleCatalog()
{
    base.ConfigureModuleCatalog();

    ModuleCatalog moduleCatalog = (ModuleCatalog)this.ModuleCatalog;
    moduleCatalog.AddModule(typeof(HelloWorldModule.HelloWorldModule));
}
```

Note: In this example, the modules are directly referenced by the shell. That is why this example is able to use **typeof(Module)** to add modules to the catalog. But keep in mind that modules whose type is not already available can also be added to the catalog.

The **WhenAvailable** initialization mode is the default value if no initialization mode is specified.

4. Build and run the solution. To verify that the **HelloWorldModule** module gets initialized, add a breakpoint to the **Initialize** method of the **HelloWorldModule** class. The breakpoint should be hit when the application starts.

Task 3: Adding a View

In this task, you will create and add a view to the **HelloWorldModule** module. Views are objects that contain visual content. Views are often user controls, but they do not have to be user controls. Adding a view to your module involves the following tasks:

1. **Creating the view.** In this task, you implement the view by creating the visual content and writing code to manage the UI elements in the view.
2. **Showing the view in a region.** In this task, you obtain a reference to a region and add the view to it.

The following procedure describes how to create a view.

To create a view

1. Add a new WPF user control to your module. To do this, right-click the Views folder in Solution Explorer, point to **Add**, and then click **New Item**. In the **Add New Item** dialog box, select the User Control (WPF) template, set the name to **HelloWorldView.xaml**, and then click **Add**.
2. Add a "Hello World" text block to the view. To do this, you can replace your code in the file **HelloWorldView.xaml** with the following code.

XAML

```
<UserControl x:Class="HelloWorldModule.Views.HelloWorldView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
```

```

        <TextBlock Text="Hello World" Foreground="Green"
HorizontalAlignment="Center" VerticalAlignment="Center" FontFamily="Calibri"
FontSize="24" FontWeight="Bold"></TextBlock>
</Grid>
</UserControl>

```

3. Save the file.

Note: To keep this hands-on lab simple, the procedure did not explain how to create a view following the Model-View-ViewModel (MVVM) pattern. For more information about the MVVM pattern, see [Implementing the MVVM Pattern](#).

Region Manager

The region manager service is responsible for maintaining a collection of regions and creating new regions for controls. This service implements the **Microsoft.Practices.Prism.Regions.IRegionManager** interface. Typically, you interact directly with this service to locate regions in a decoupled way through their name and add views those regions. By default, the **UnityBootstrapper** base class registers an instance of this service in the application container. This means that you can obtain a reference to the region manager service in the HelloWorld application by using dependency injection.

The following procedure explains how to obtain an instance of the region manager and add the HelloWorldView view to the shell's main region.

To show the view in the shell

1. Open the HelloWorldModule.cs file.
2. Add the following **using** statement to the top of the file. You will use it to refer to the region elements in the Prism Library.

C#

```
using Microsoft.Practices.Prism.Regions;
```

3. Create a private read-only instance variable to hold a reference to the region manager. To do this, paste the following code inside the class body.

C#

```
private readonly IRegionManager regionManager;
```

4. Add the constructor of the **HelloWorldModule** class to obtain a region manager instance through constructor dependency injection and store it in the **regionManager** instance variable. To do this, the constructor has to take a parameter of type **Microsoft.Practices.Prism.Regions.IRegionManager**. You can paste the following code inside the class body to implement the constructor.

C#

```
public HelloWorldModule(IRegionManager regionManager)
{
```

```
    this.regionManager = regionManager;
}
```

5. In the **Initialize** method, invoke the **RegisterViewWithRegion** method on the **RegionManager** instance. This method registers a region name with its associated view type in the region view registry; the registry is responsible for registering and retrieving of these mappings.

The **RegisterViewWithRegion** method has two overloads. When you want to register a view directly, you use the first overload that requires two parameters, the region name and the type of the view. This is shown in the following code.

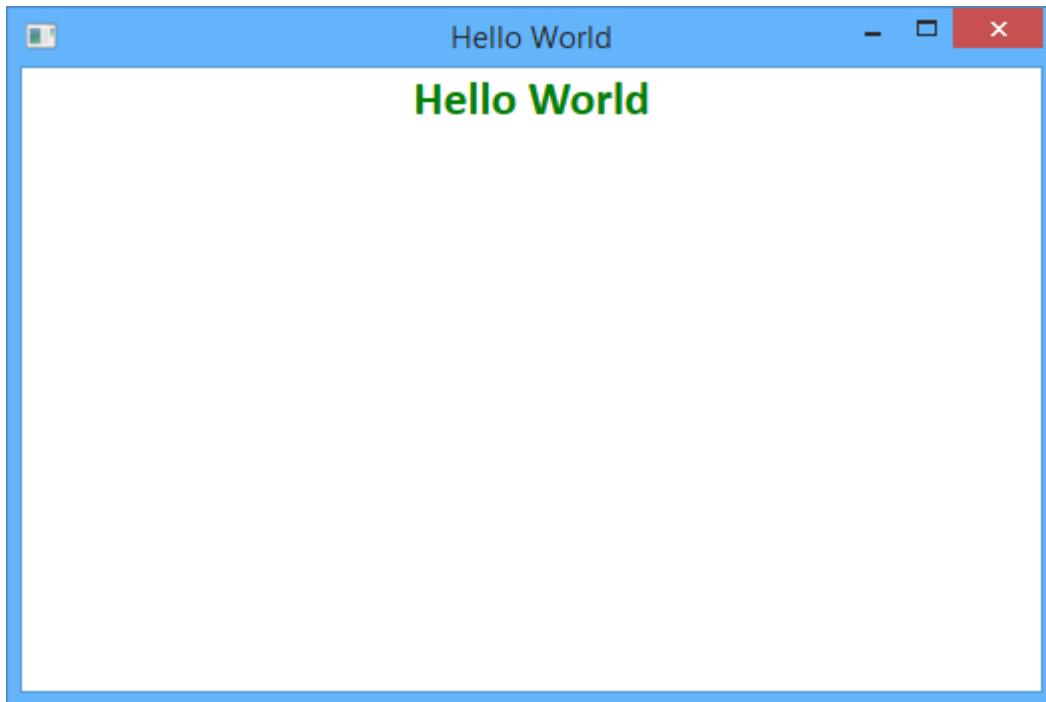
C#

```
public void Initialize()
{
    regionManager.RegisterViewWithRegion("MainRegion",
    typeof(Views.HelloWorldView));
}
```

The UI composition approach used in the preceding code is known as view discovery. When using this approach, you specify the views and the region where the views will be loaded. When a region is created, it asks for its associated views and automatically loads them.

The region's name must match the name defined in the **RegionName** attribute of the region.

6. Build and run the application. You should see the Hello World window with a "Hello World" message, as shown in the following illustration.



Hello World message

Note: To open the solution that results from performing the steps in this Hands-on Lab in Visual Studio, run the file Desktop only - Open QS - Hello World QuickStart.lnk.

18: Publishing and Updating Applications Using the Prism Library Hands-on Lab

In this lab, you will learn how to publish, deploy, and update a composite Prism Windows Presentation Foundation (WPF) application that uses dynamic module loading with ClickOnce. After completing this lab, you will be able to do the following:

- Publish an existing WPF Prism shell application project with ClickOnce.
 - Add dynamically loaded modules to the published application for deployment.
 - Deploy the application to a client computer.
 - Publish an update to the application.
 - Deploy the update to a client computer.
-

System Requirements

This guidance was designed to run on the Microsoft Windows 8, Windows 7, Windows Vista, Windows Server 2012, or Windows Server 2008 operating system. WPF applications built using this guidance require the .NET Framework 4.5.

Before you can use the Prism Library, the following must be installed:

- Microsoft Visual Studio 2013 Professional, Premium, or Ultimate editions
 - Microsoft .NET Framework 4.5 (installed with Visual Studio 2013)
 - Optional tool:
 - [Microsoft Blend for Visual Studio 2013](#)
-

Preparation

This topic requires you to have Prism and the Prism QuickStarts in the default installed directory structure. This lab uses the ModularityWithUnity.Desktop QuickStart that is included with the Prism installed source code.

Note: This hands-on lab uses the QuickStart that uses a Unity container, but you can also use the Managed Extensibility Framework (MEF) variant of the QuickStart.

To compile the solution

1. Open the solution file
 \Quickstarts\Modularity\Desktop\ModularityWithUnity\ModularityWithUnity.Desktop.sln.
 2. Build the solution.
-

Additionally, this lab uses the Manifest Manager Utility, which is available on the Prism CodePlex site at <http://compositewpf.codeplex.com/releases/view/14771> in the **Download** section. You will need to download and extract the source code for that utility, and build it to use it later in the lab. You can either run it from a separate instance of Visual Studio or you can build once and just run the binaries for the second task in this lab.

Note: This hands-on lab assumes that you understand Prism modularity and deployment concepts. For more information, see [Modular Application Development](#) and [Deploying Applications](#).

Procedures

This lab includes the following tasks:

- Task 1: Publishing an initial version of the shell application
 - Task 2: Updating the manifests to include dynamically loaded module assemblies
 - Task 3: Deploying the initial version to a client machine
 - Task 4: Publishing an updated version of the application and updating the manifests
 - Task 5: Deploying the updated version to a client computer
-

The next sections describe each of these tasks.

Note: The instructions for this hands-on lab are based on the ModularityWithUnity.Desktop solution.

Task 1: Publishing an Initial Version of the Shell Application

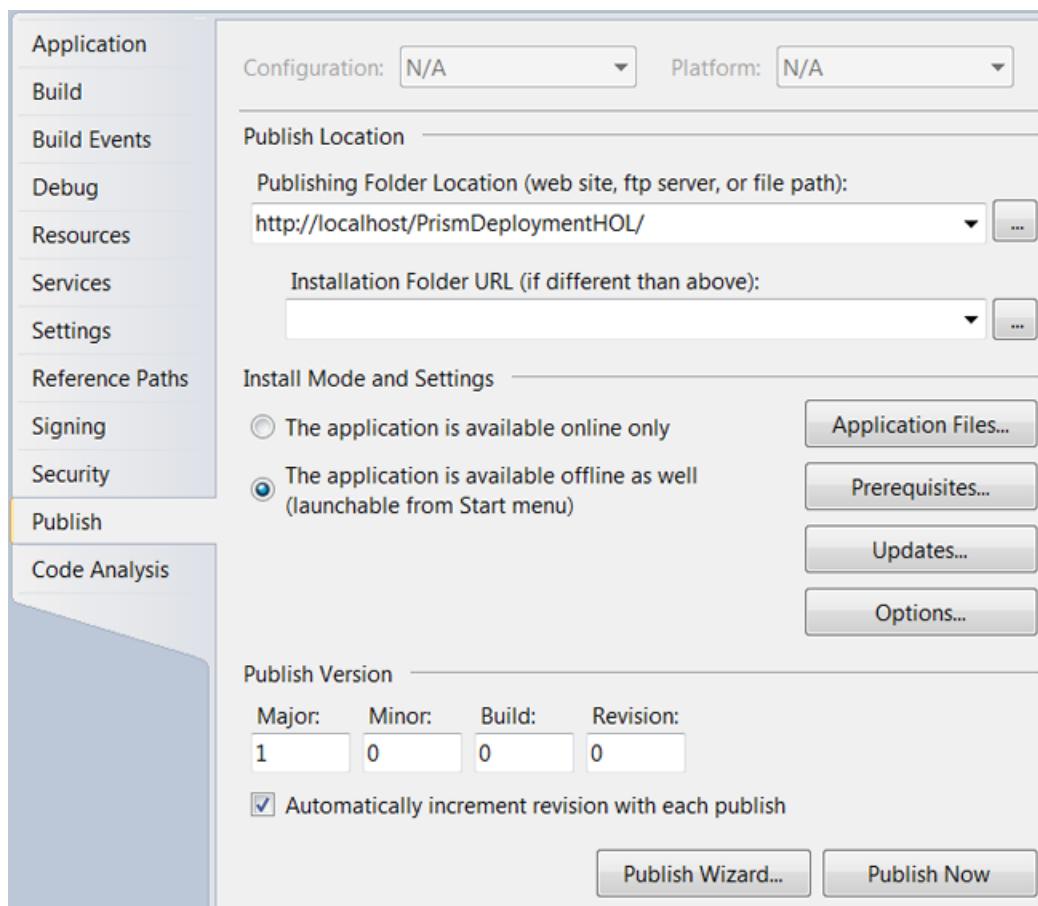
In this task, you will publish the initial version of the shell application project using Visual Studio. The following steps will be performed:

1. **Set the ClickOnce publish settings.** In this task, you review and configure the project settings for the shell project that determine the ClickOnce deployment behavior.
2. **Add a publisher certificate.** In this task, you create a test publisher certificate to enable ClickOnce publishing and associate it with the application.
3. **Publish the application.** In this task, you physically publish the shell application from Visual Studio to a target deployment directory.
4. **Verify the published output.** In this task, you verify the output of the publication in the target directory.

The following procedure describes how to configure the ClickOnce publish settings within the shell project. These settings alter the behavior of ClickOnce, both at initial installation time and when setting the update policies for the application. The publish settings are only relevant for the shell project itself because it is the launch application executable, which determines the deployment behavior of the application as a whole in a ClickOnce deployed application.

To set the ClickOnce publish settings

5. In Visual Studio, open the project properties for the ModularityWithUnity.Desktop WPF project. To do this, right-click the project in Solution Explorer, and then click **Properties**. In the project settings, click the **Publish** tab. The ClickOnce publishing settings will be shown, as in the following illustration.

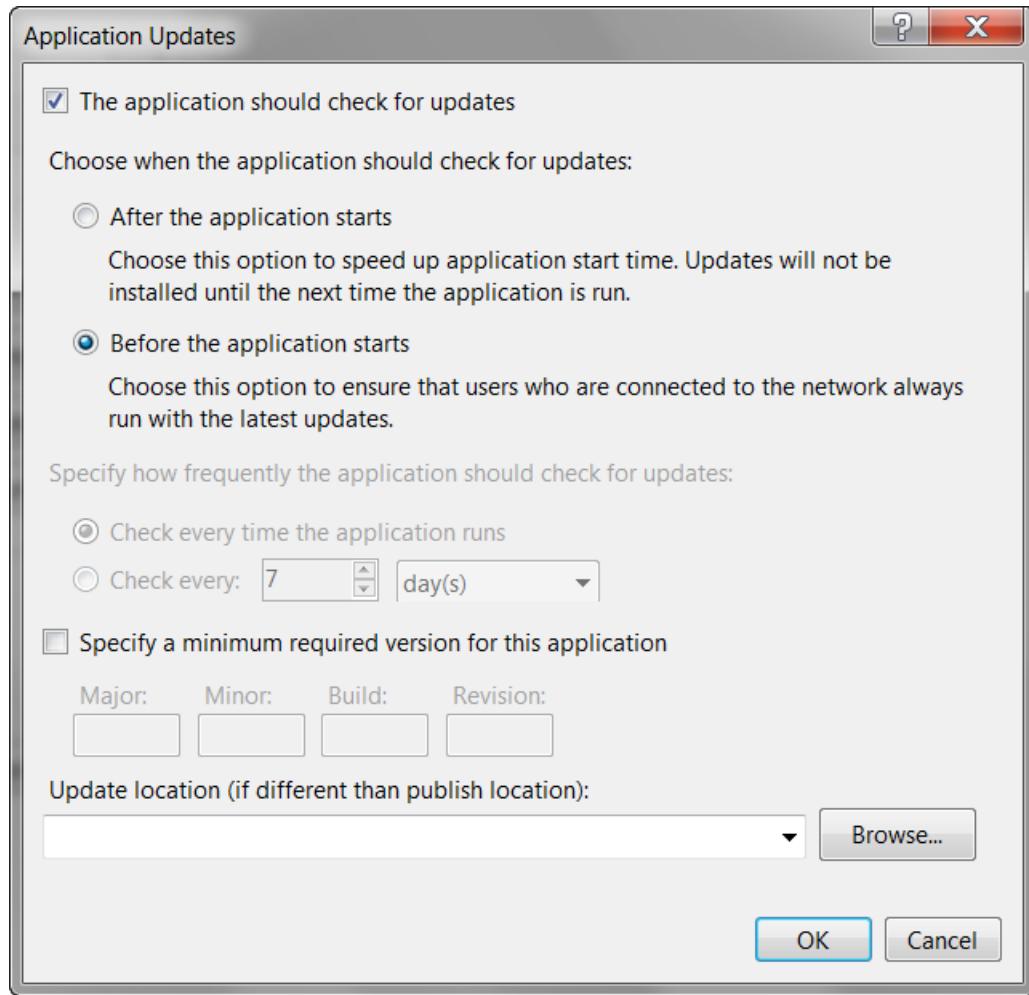


ClickOnce publish settings

6. Change the publishing folder location to **http://localhost/PrismDeploymentHOL** if you have IIS on your local computer. If you do not, you can publish to another IIS computer for which you have administrator permissions to create a new virtual directory, or you can use a fully qualified Universal Naming Convention (UNC) file path (such as \\mymachinename\c\$\PrismDeploymentHOL) if you first create that directory. The address used is the one that will be used to install the application later in the lab, so make sure you note

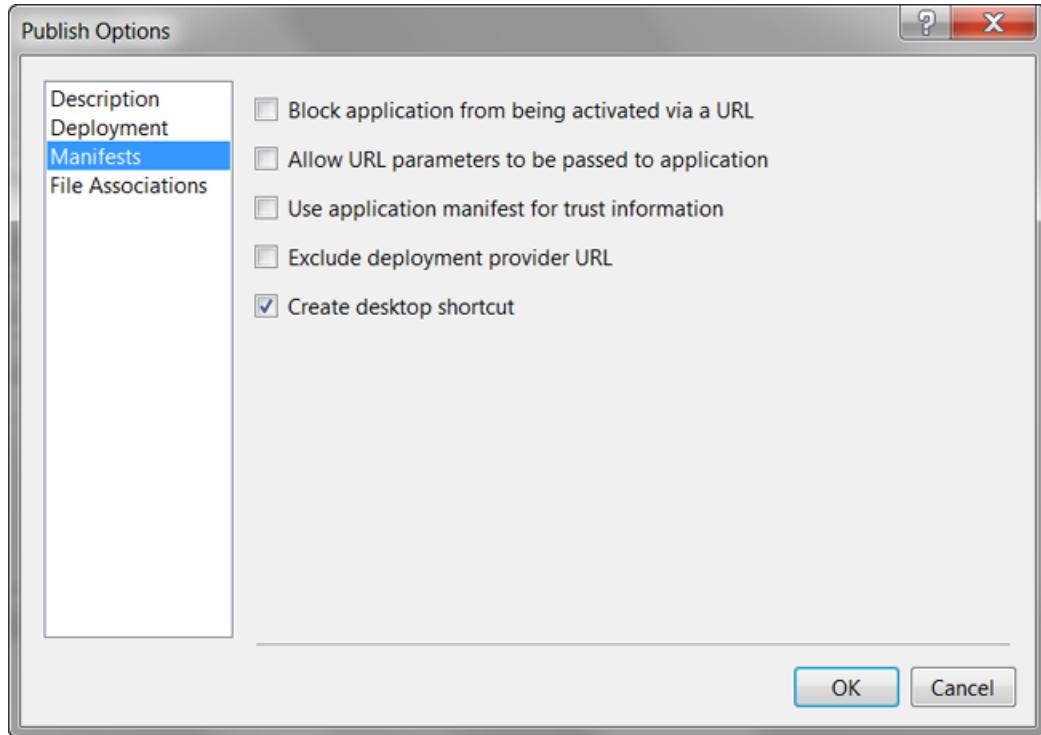
the address. This address is the physical address you use to push the ClickOnce manifests and application files to the deployment server when you publish.

7. The installation folder URL can be used if the externally exposed address used for installation of the application will be different from the one used for publishing (for example, if you are publishing via FTP to one of your servers, but users will install the application based on an externally visible HTTP path to that server). This path represents the installation address on the deployment server to the users. If the path is not supplied, it is assumed you can launch the application using the same address you used to publish.
8. The install mode and settings provide you fine-grained options for configuring the way the application installs, what files it is composed of, if there are prerequisite installations that need to happen first (such as installing the .NET Framework 4.5), how updates are performed, and a number of other options. For this lab, you will use the default settings, which configures the application to install for offline use (meaning it can at least be launched even if you are not connected to the deployment server, but it depends on what your application does after launch as to whether it will function properly). The default settings also set the application to automatically update before launch if a new version is detected on the deployment server.
9. Click the **Updates** button, and then select check box labeled **The application should check for updates**, as shown in the following illustration.



Application Updates dialog box

10. Click the **Options** button. In the **Options** dialog box, click **Manifests** in the left pane, and then select the check box labeled **Create desktop shortcut**.



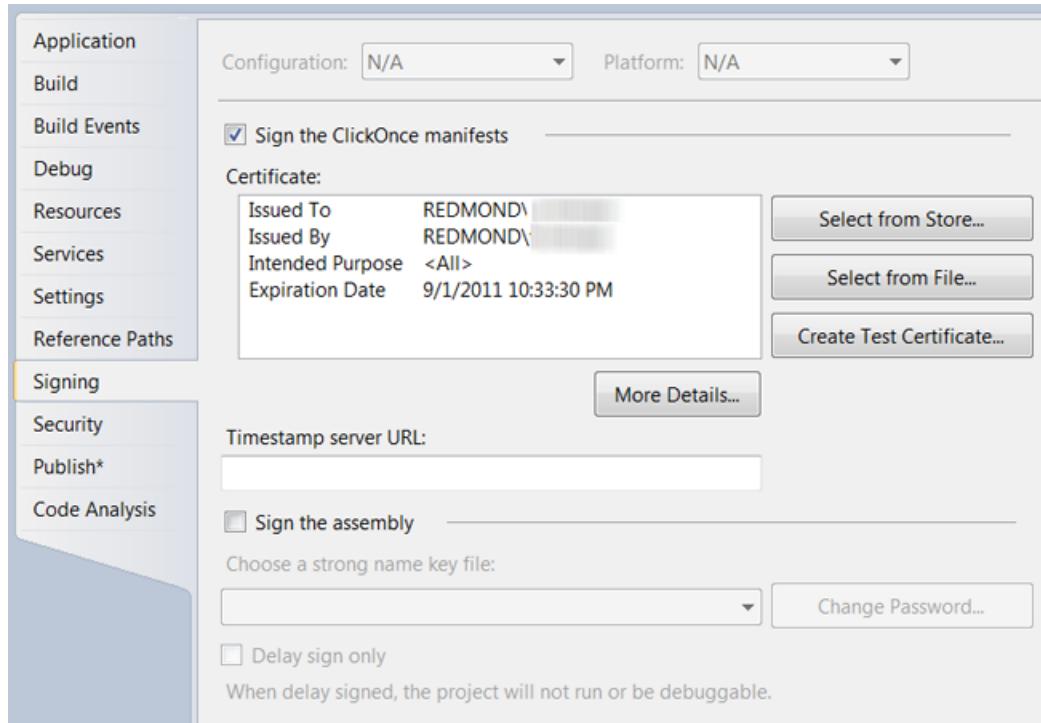
Publish Options dialog box

11. The publish version drives detection of updates for installed ClickOnce applications. Generally, you want to be in explicit control over this version in a real deployment. For this lab, you will allow Visual Studio to automatically increment this version number each time you publish.

The following procedure explains how to set up the certificate used for signing the published application. To ensure that your application cannot be replaced on the deployment server with a tampered version, ClickOnce requires you to digitally sign the ClickOnce manifests using an X509 code signing certificate. For development purposes, Visual Studio can generate a test certificate for your use. For putting your application into production, it is not recommended to use a test certificate. You should either obtain a certificate from a well-known (Trusted Root) certificate authority for public deployments or obtain one from your domain administrators for an internal deployment. In this lab, you will simply use the Visual Studio-generated test certificate.

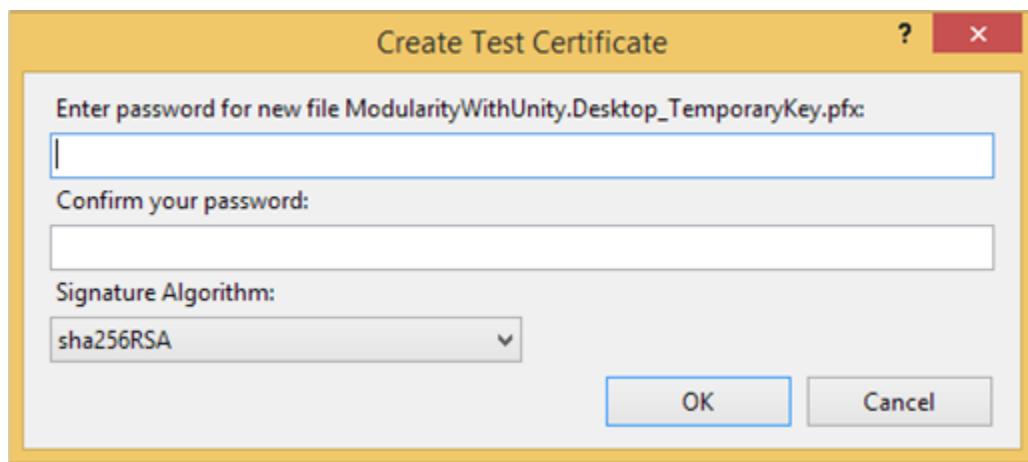
To add a publisher certificate

1. In the shell project properties, click the **Signing** tab.
2. Select the check box labeled **Sign the ClickOnce manifests**. The certificate information will initially be blank if you have not previously created or associated a certificate with the project.



ClickOnce Publish project property settings

3. Click the **Create Test Certificate** button. This opens the Create Test Certificate dialog box, as shown in the following illustration.

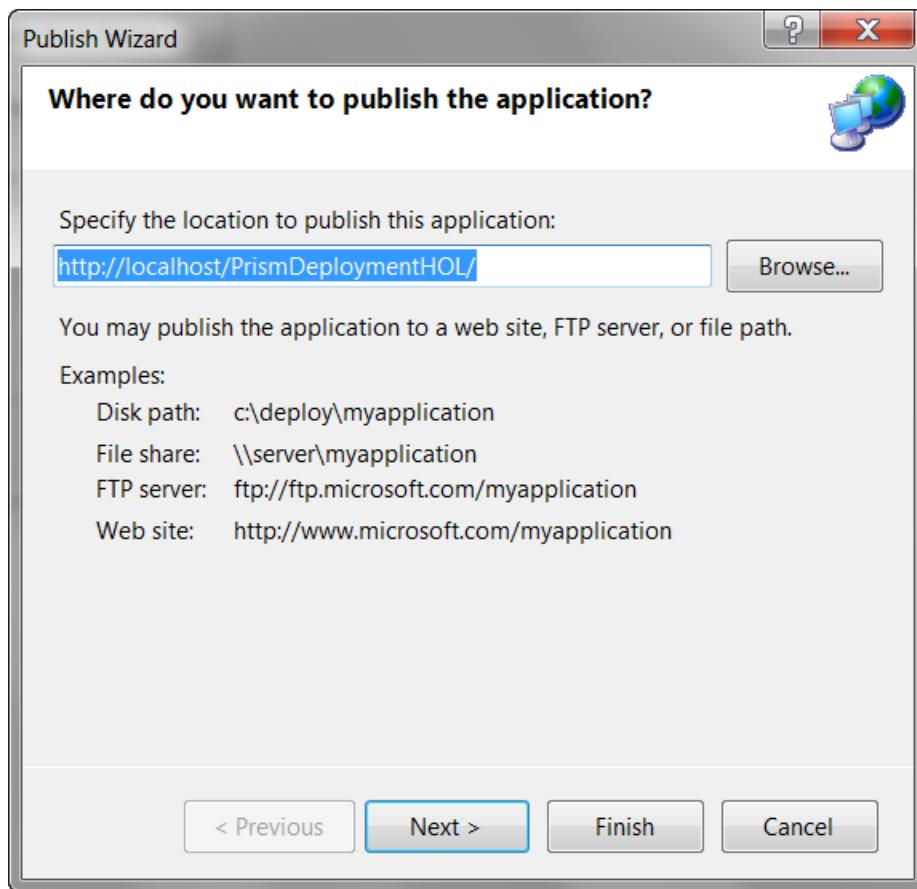


Create Test Certificate password dialog box

4. Click **OK** to leave the test certificate without a password.
5. The certificate information should now be populated, and the certificate name and issuer will be based on your logged-on Windows account information. If you have an existing certificate as a file or already installed in your certificate stores, you can select the certificate using one of the buttons next to the certificate information instead.

To publish the shell application

1. Build the application and make sure it builds as expected. Publishing the application will cause the application to build, but it is easier to resolve any build errors with a normal build before publishing.
2. In Visual Studio, click **Publish ModularityWithUnity.Desktop** on the **Build** menu.
3. The **Publish Wizard** dialog box displays the publish folder location address that you entered in step 2, as shown in the following illustration. Click **Finish** to publish the application.



Publish Wizard dialog box

Note: Depending on the computer you publish to and the security settings, you may get a warning that Visual Studio is unable to view the published application. This simply means it was unable to launch a browser and navigate to the publish location URL. However, the application is not really ready to install yet at this point because you need to add the dynamic modules to the manifests in the next task.

Task 2: Updating the Manifests to Include Dynamically Loaded Module Assemblies

In this task, you will edit the ClickOnce manifests of your deployed application to add the dynamic module assemblies. This involves editing the application files list in the application manifest, saving and re-signing the application manifest, updating the application manifest reference within the deployment manifest, and saving and re-signing the deployment manifest. These steps can all be performed individually using the .NET Framework SDK tool named the Manifest Generating and Editing tool (Mage). However, the Manifest Manager Utility that you can download from the Prism CodePlex site automates these steps into a single easy editor. To accomplish them, you will do the following:

1. **Open the deployment manifest in the Manifest Manager Utility.** In this task, you run the utility to simplify editing of the manifests.

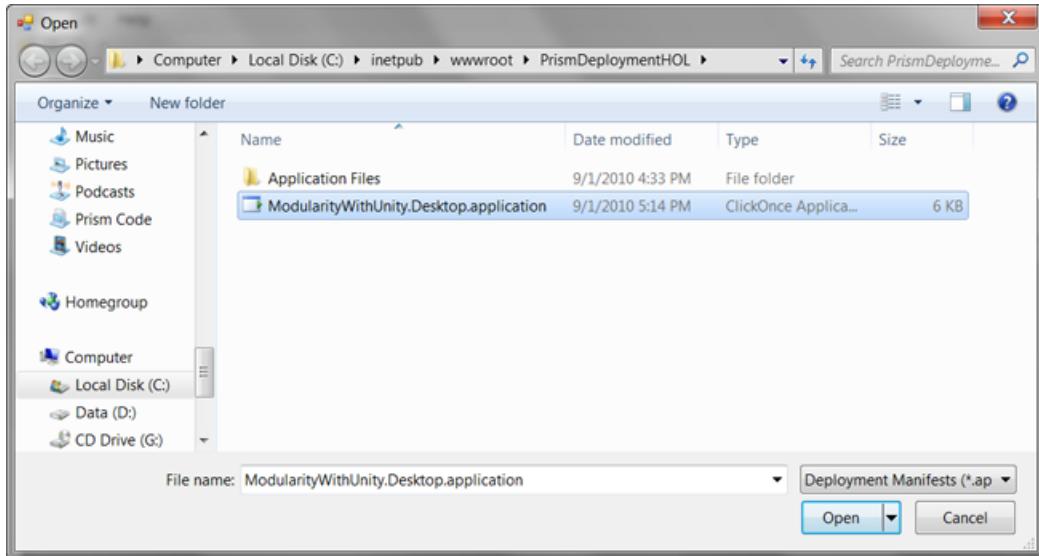
Important: You must run this utility as an administrator.
2. **Add the dynamically loaded modules to the manifests.** In this task, you locate and add the dynamic module assemblies to the manifest and get them deployed to the publish location.
3. **Save and sign the manifests.** In this task, you select the publisher certificate used for signing the ClickOnce manifests to save and re-sign the manifests.

The following procedure describes how to add the dynamic module assemblies to the ClickOnce manifests.

To open the deployment manifest in the Manifest Manager Utility

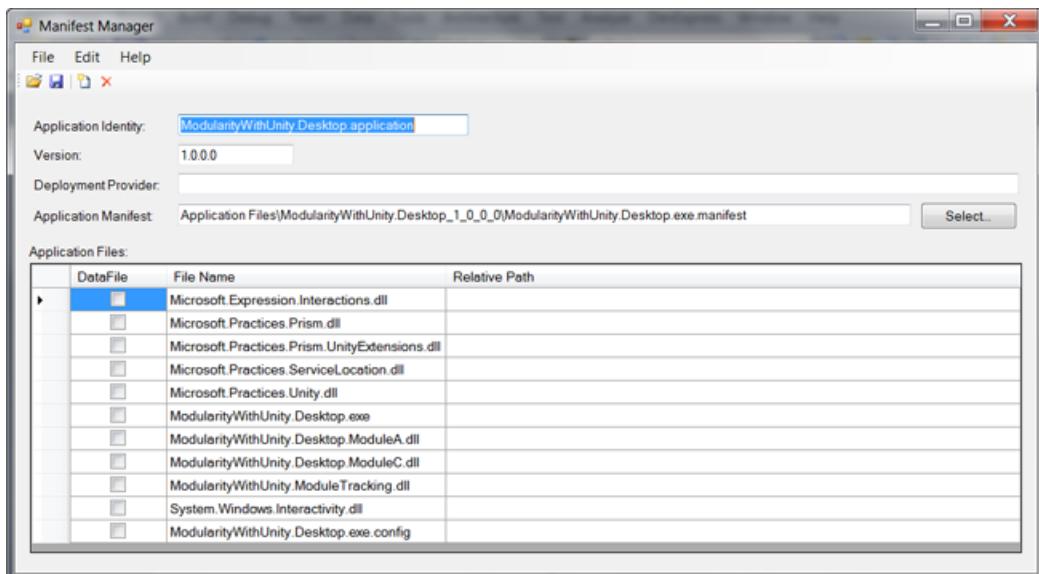
1. If you have not already done so, download the most recent Manifest Manager Utility from the **Download** section of the Prism CodePlex site at <http://compositewpf.codeplex.com/releases/view/14771> and unzip it to a working directory on your computer.
2. In Visual Studio 2013, open the file ManifestManagerUtility.sln, build it, and run it.

Important: You must run this utility as an administrator. If you are running this from Visual Studio, you must start Visual Studio as an administrator.
3. On the **File** menu, click **Open**, and then navigate to the publish folder location where you published the QuickStart in the previous task. In that folder, select the deployment manifest file ModularityWithUnity.Desktop.application, and then click **Open**.



Open dialog box from Manifest Manager Utility in publish folder location

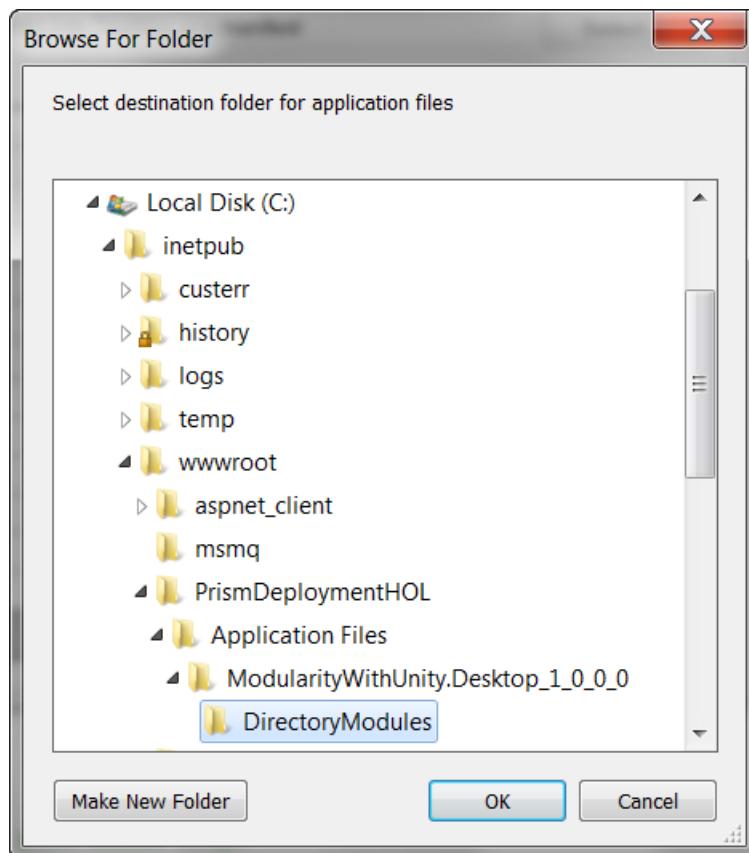
4. The deployment and linked application manifest files will be opened by the utility and will be presented in the unified view of the utility, as shown in the following illustration. You can see that the shell executable file and all referenced assemblies that are not part of the framework are automatically included. Note that Modules A and C are included because they were referenced for static loading by the QuickStart, but you will need to add the additional modules using the utility.



Manifest Manager utility

To add the dynamically loaded modules to the manifest

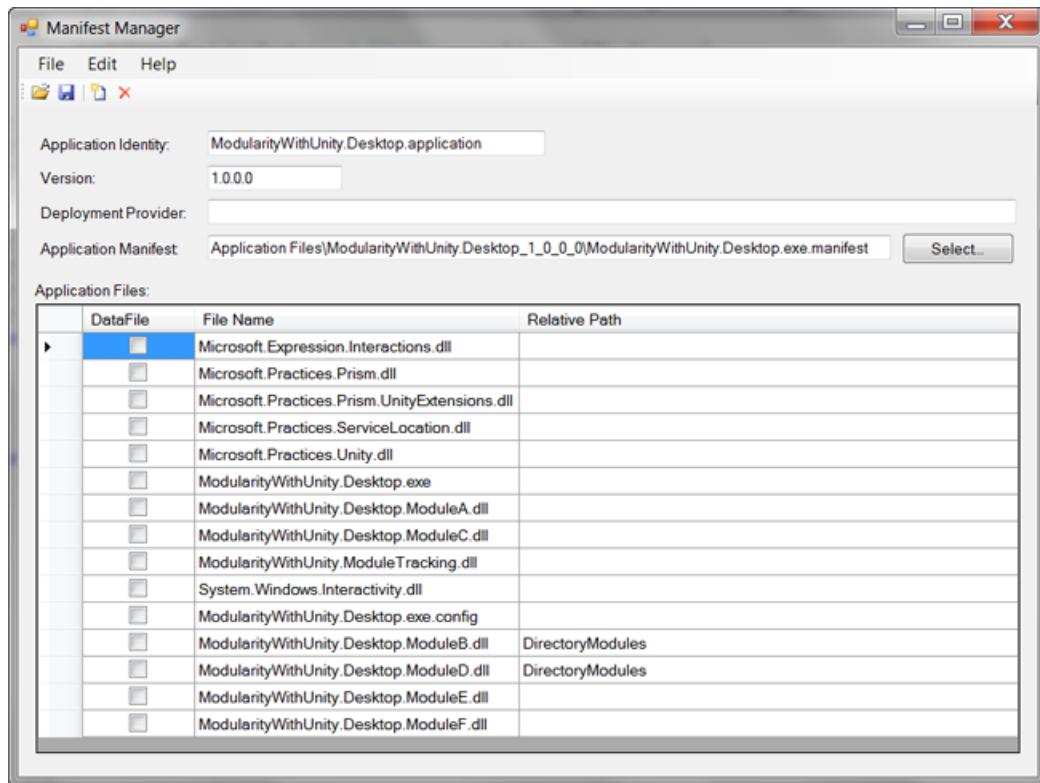
1. On the **Edit** menu, click **Add Files**. In the **Add Application Files** dialog box, navigate to the build output folder for Module B (such as C:\temp\ModularityWithUnity\ModuleB\bin\Debug\), and select the module DLL (such as ModularityWithUnity.Desktop.ModuleB.dll). In the **Add Application Files** dialog box, click **Open** to add the module DLL to the manifest.
2. When you click **Open**, a **Browse For Folder** dialog box appears. In this dialog box, you can specify the destination folder to copy the module file to the publish folder. Modules B and D are loaded in the QuickStart through directory scan, and the bootstrapper sets the folder it scans to a relative path of .\DirectoryModules from the executable file. This means the files need to be in that same relative path in the published application.
3. Select the version-specific Application Files folder, and then click the **Make New Folder** button at the bottom of the dialog box.
4. Name the new folder **DirectoryModules**.
5. Make sure the new folder is selected, and then click **OK**. This copies the Module B DLL into the **DirectoryModules** subfolder of the application files, as shown in the following illustration.



Browse For Folder dialog box with DirectoryModules subfolder selected

6. Repeat the preceding steps to add Module D to the manifest and place it in the **DirectoryModules** subfolder.

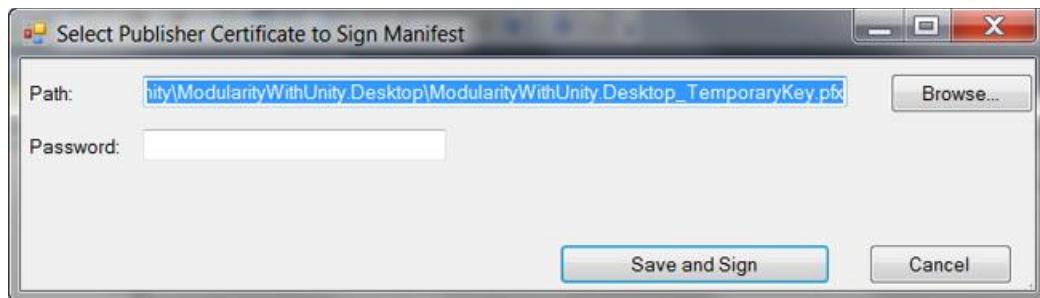
7. Repeat the preceding steps to add Modules E and F to the manifest, but those both go in the root Application Files folder (ModularityWithUnity.Desktop_1_0_0_0).
8. At this point, the additional modules should be listed in Manifest Manager Utility with the relative path shown for Modules B and D, as shown in the following illustration (order does not matter).



Manifest Manager utility with Modules B, D, E, and F added

To sign and save the manifests

1. Click the **Save** button on the toolbar of the utility. This opens the **Select Publisher Certificate to Sign Manifest** dialog box.



Select Publisher Certificate to Sign Manifest dialog box

2. Click the **Browse** button, and then locate and select the ModularityWithUnity.Desktop_TemporaryKey.pfx file that was generated when you added the test certificate to the project in the first task of this lab.
3. Click the **Save and Sign** button, leaving the password blank again.

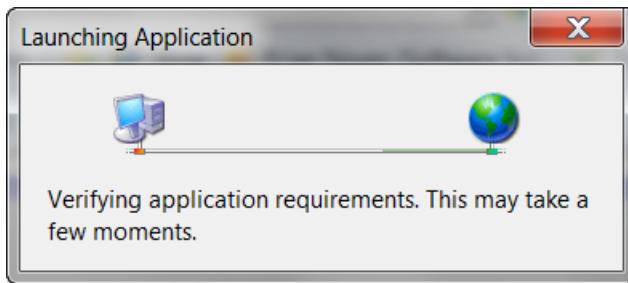
At this point, you have successfully published the application with modified manifest files and it is ready to install.

Task 3: Deploying the Initial Version to a Client Computer

In this task, you will launch and install the application.

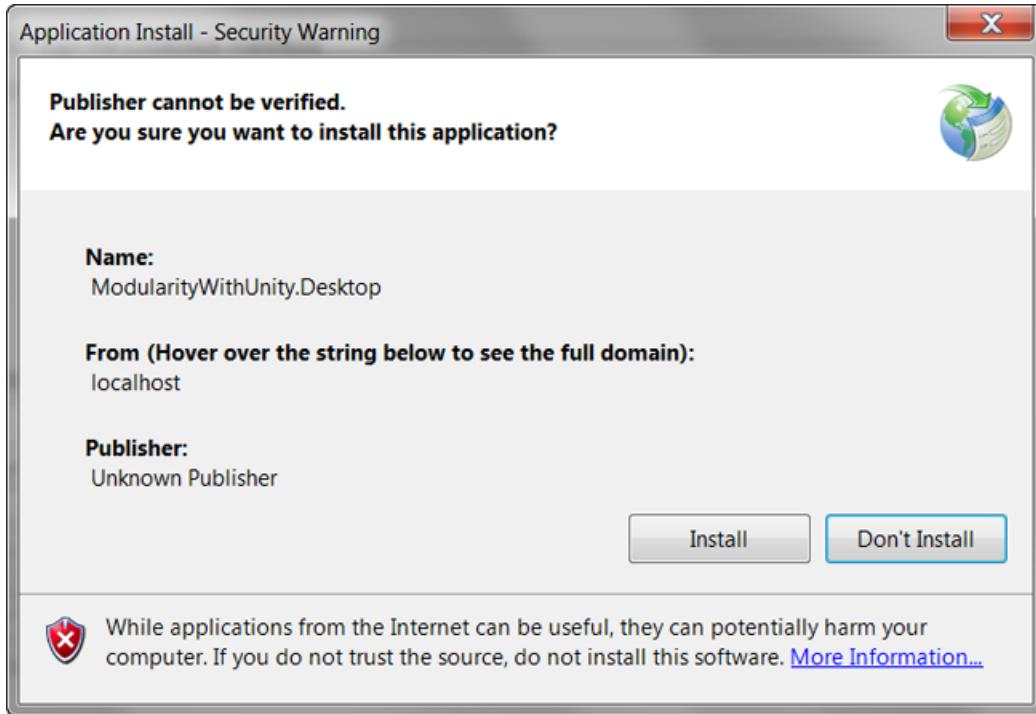
To launch and install the application

1. Open an Internet Explorer browser window and enter the address you used as the publish folder location with the deployment manifest (.application file) path added to the end of it (such as <http://localhost/PrismDeploymentHOL/ModularityWithUnity.Desktop.application>).
2. A **Launching Application** dialog box briefly appears as ClickOnce downloads the manifests for the application, as shown in the following illustration.



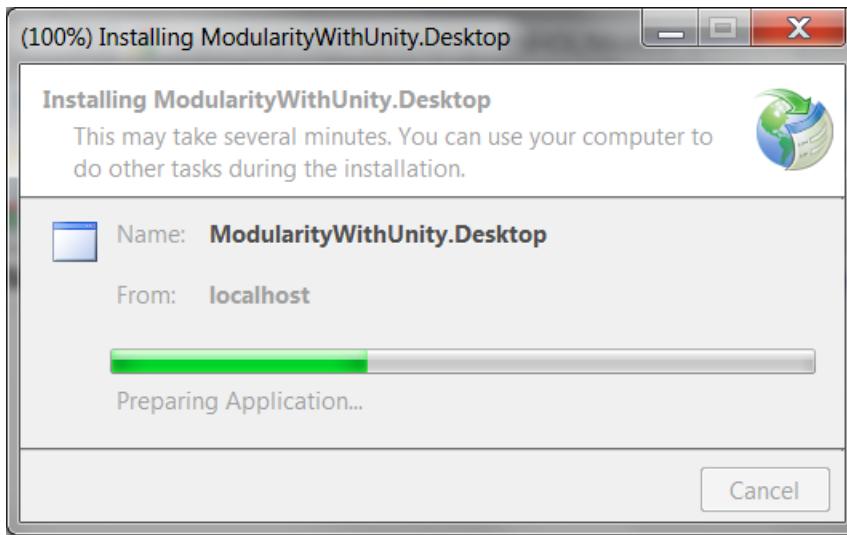
ClickOnce Launching Application dialog box

3. A security warning appears, as shown in the following illustration. It notifies the user of who the publisher of this application is. Because you are using a test certificate, it will show an unknown publisher. To get a more friendly security warning, you will need a certificate issued from a trusted root certification authority.



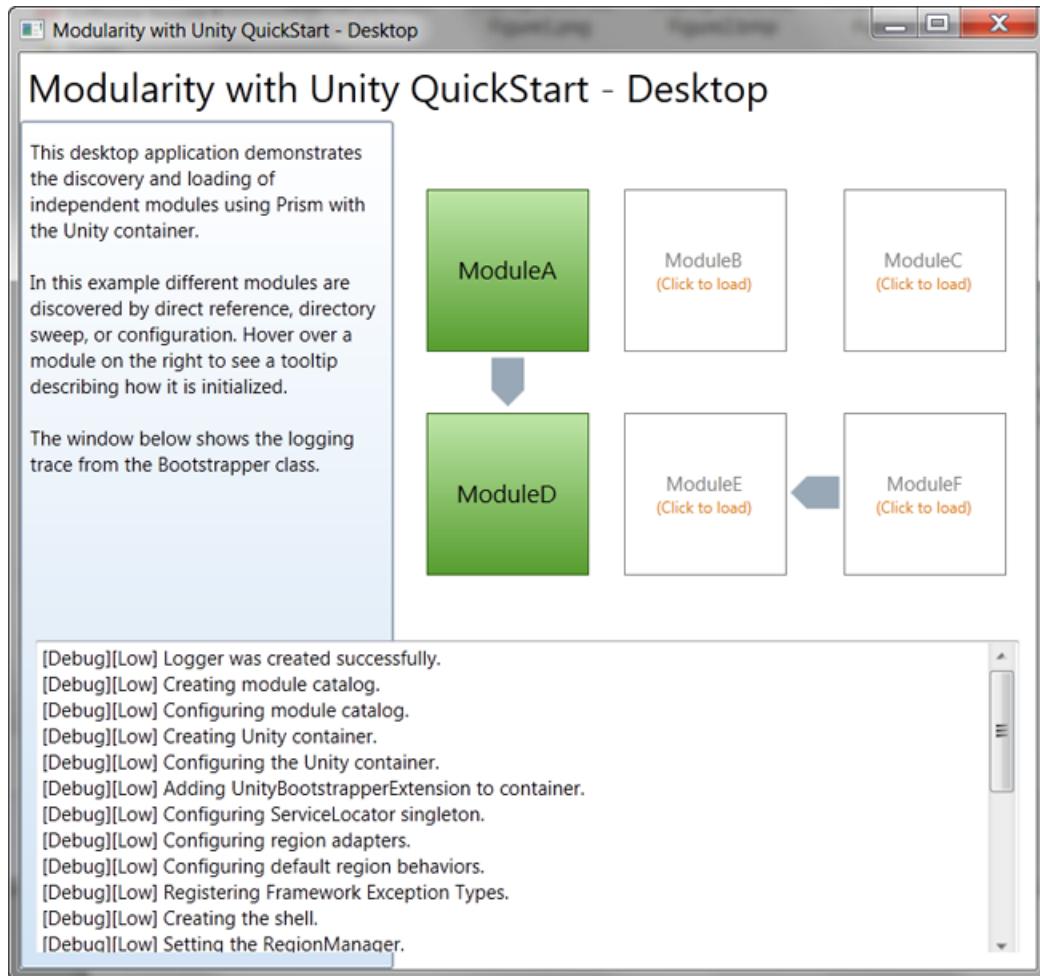
ClickOnce security warning

4. Click **Install**. While the rest of the application files are downloaded and launched, you will briefly see a dialog box with a progress bar, as shown in the following illustration.



Install progress bar

5. The QuickStart should launch and you should see Modules A and D load when it starts. You can click on the other squares to get the other modules to load on demand.



Modularity with Unity QuickStart running

Task 4: Publishing an Updated Version of the Application and Updating the Manifests

In this task, you will make a simple visible change to the application and publish the new version. To accomplish this, you will do the following:

1. **Modify the title of the application.** In this task, you will modify the large text at the top of the application to indicate a modified version. This gives a simple visible change to the application so you can verify the updated application launches in the next task.
2. **Publish the new version of the application.** In this task, you publish the application again with a new publish version so that the ClickOnce update checking will see that there is a new version of the application on the deployment server.

3. **Update the manifests.** In this task, you will use the manifest manager utility again to re-add Modules B, D, E, and F to the deployment because each time you re-publish, the manifest is re-generated by Visual Studio based on the referenced assemblies from the shell.
-

The following procedure describes how to publish the updated version.

To modify the title of the application

1. With the ModularityWithUnity.Desktop project open, open the Shell.xaml file in the designer.
 2. Modify the **Title** property of the window to read **Modularity with Unity QuickStart – Desktop – Modified**.
 3. Save and build the solution.
-

To publish the new version of the application

1. On the **Build** menu, click **Publish ModularityWithUnity.Desktop**.
 2. In the Publish Wizard, click **Finish**.
 3. The new version will be published and the publish version number will be 1.0.0.1 because Visual Studio auto-incremented the publish version when you first published in Task 1.
-

To update the manifests for the new version

1. Open Manifest Manager Utility again.
 2. On the **File** menu, click **Open**, and then locate and open the ModularityWithUnity.Desktop.application deployment manifest again (Manifest Manager Utility should remember the location from the last time you opened a manifest). You should see that the manifest version is now 1.0.0.1 and Modules B, D, E, and F are missing again.
 3. On the **Edit** menu, click **Add Files** to select the Modules B assembly.
 4. When the **Browse For Folder** dialog box appears, go to the new published version's Application Files folder (ModularityWithUnity.Desktop_1_0_0_1), create a DirectoryModules subfolder, and then select it to place Module B in that relative path.
 5. Repeat steps 3 and 4 for Module D, also putting it into the DirectoryModules subfolder.
 6. Repeat steps 3 and 4 for Modules E and F, but place them in the ModularityWithUnity.Desktop_1_0_0_1 directory, not the DirectoryModules subfolder.
 7. On the toolbar, click the **Save** button.
 8. Manifest Manager Utility should remember the path to the publisher certificate file you used to publish the first version, so you can just click the **Save and Sign** button.
-

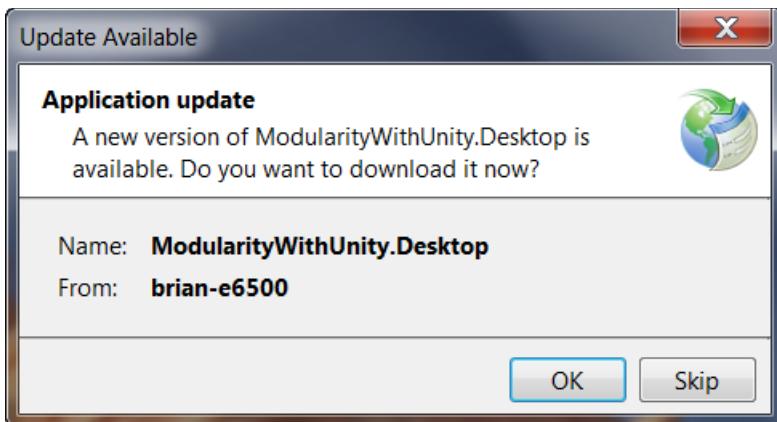
The new version is published and ready to deploy.

Task 5: Deploying the Updated Version to a Client Computer

In this task, you will launch the application as the client computer and see that it automatically updates.

To deploy the updated version to a client computer

1. Locate the shortcut on your desktop that was created when you installed the initial version of the application (ModularityWithUnity.Desktop), and then click it to launch the application from the client computer.
2. The Update Available dialog box appears, as shown in the following illustration. Click **OK** to accept the update.



Update Available dialog box

You should see the modified title on the application after it has launched.

Bibliography

General Links

To download Prism binaries, source code, and documentation, see the Prism home page on MSDN at <http://www.microsoft.com/Prism>.

If you have comments on this guide, visit the Prism community site at <http://www.codeplex.com/Prism>.

1: Introduction

Prism assumes you have hands-on experience with WPF. If you need general information about WPF see the following resources:

- [Windows Presentation Foundation](#) on MSDN.
 - MacDonald, Matthew. *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4*, Apress, 2010.
 - Nathan, Adam. *WPF 4 Unleashed*. Sams Publishing, 2010.
-

2: Initializing Prism Applications

For more information about MEF, **AggregateCatalog**, and **AssemblyCatalog**, see [Managed Extensibility Framework Overview](#) on MSDN.

3: Managing Dependencies Between Components

For information related to containers, see the following:

- [Unity Application Block](#) on MSDN.
- [Unity community site](#) on CodePlex.
- [Managed Extensibility Framework Overview](#) on MSDN.
- [MEF community site](#) on CodePlex
- [Inversion of Control containers and the Dependency Injection pattern](#) on Martin Fowler's website.
- [Design Patterns: Dependency Injection](#) in *MSDN Magazine*.
- [Loosen Up: Tame Your Software Dependencies for More Flexible Apps](#) in *MSDN Magazine*.
- [Castle Project](#)
- [StructureMap](#)
- [Spring.NET](#)

4: Modular Application Development

To learn more about modularity in Prism, see the Modularity with MEF for WPF QuickStart or the Modularity with Unity for WPF QuickStart. For more information about the QuickStarts, see [Modularity QuickStarts](#).

For information about the modularity features that can be extended in the Prism Library, see [Modules](#) in [Extending the Prism Library](#).

5: Implementing the MVVM Pattern

For more information about data binding in WPF, see [Data Binding](#) on MSDN.

For more information about binding to collections in WPF, see [Binding to Collections](#) in [Data Binding Overview](#) on MSDN.

For more information about the Presentation Model pattern, see [Presentation Model](#) on Martin Fowler's website.

For more information about data templates, see [Data Templating Overview](#) on MSDN.

For more information about MEF, see [Managed Extensibility Framework Overview](#) on MSDN.

For more information about Unity, see [Unity Application Block](#) on MSDN.

For more information about **DelegateCommand** and **CompositeCommand**, see [Communicating Between Loosely Coupled Components](#).

For more information about using MVVM in Windows Store Apps see [Using the Model-View-ViewModel \(MVVM\) pattern in a Windows Store business app using C#, XAML, and Prism](#).

6: Advanced MVVM Scenarios

For more information about the logical tree, see [Trees in WPF](#) on MSDN.

For more information about attached properties, see [Attached Properties Overview](#) on MSDN.

For more information about MEF, see [Managed Extensibility Framework Overview](#) on MSDN.

For more information about Unity, see [Unity Application Block](#) on MSDN.

For more information about **DelegateCommand**, see [Implementing the MVVM Pattern](#).

For more information about using Microsoft Expression Blend behaviors, see [Working with built-in behaviors](#) on MSDN.

For more information about creating custom behaviors with Microsoft Expression Blend, see [Creating Custom Behaviors](#) on MSDN.

For more information about creating custom triggers and actions with Microsoft Expression Blend, see [Creating Custom Triggers and Actions](#) on MSDN.

For more information about using the dispatcher in WPF , see [Threading Model](#) and [The Dispatcher Class](#) on MSDN.

For more information about region navigation, see the section, [View-Based Navigation](#) in [Navigation](#).

For more information about the Event-based Asynchronous pattern, see [Event-based Asynchronous Pattern Overview](#) on MSDN.

For more information about the IAsyncResult design pattern, see [Asynchronous Programming Overview](#) on MSDN.

7: Composing the User Interface

For more information about extending the Prism Library, see [Extending the Prism Library](#).

For more information about commands, see [Commands](#) in [Implementing the MVVM Pattern](#).

For more information about data binding, see [Data Binding](#) in [Implementing the MVVM Pattern](#).

For more information about region navigation, see [Navigation](#).

For more information about the guidelines discussed in this topic, see the following:

- [Dependency Properties Overview](#) on MSDN.
- Data binding; see:
 - [Data Binding Overview](#) on MSDN.
 - [Data Binding in WPF](#) in *MSDN Magazine*.
- [Data Templating Overview](#) on MSDN.
- [Resources Overview](#) on MSDN.
- [UserControl Class](#) on MSDN.
- [VisualStateManager Class](#) on MSDN.
- [Customizing Controls For Windows Presentation Foundation](#) in *MSDN Magazine*.
- [ComponentResourceKey Markup Extension](#) on MSDN.
- [Design-Time Attributes in the WPF Designer](#) on MSDN.
- [Markup Extensions and WPF XAML](#) on MSDN.
- [Sample Data in the WPF and Silverlight Designer](#) on MSDN.
- [Learning the Visual Studio WPF and Silverlight Designer](#). This contains tutorials and articles on layout, resources, data binding, sample data, debugging data bindings, object data sources, and master-detail forms.

8: Navigation

For more information about Prism regions, see [Composing the User Interface](#).

For more information about the MVVM pattern and Interaction Request pattern, see [Implementing the MVVM Pattern](#) and [Advanced MVVM Scenarios](#).

For more information about the **Interaction Request** object, see [Using Interaction Request Objects](#) in [Advanced MVVM Scenarios](#).

For more information about the Visual State Manager, see [VisualStateManager Class](#) on MSDN.

For more information about using Microsoft Blend behaviors, see [Working with built-in behaviors](#) on MSDN.

For more information about creating custom behaviors with Microsoft Blend, see [Creating Custom Behaviors](#) on MSDN.

9: Communicating Between Loosely Coupled Components

For more information about weak references, see [Weak References](#) on MSDN.

10: Deploying Prism Applications

Download the [Manifest Manager Utility](#) from the Prism community site on Codeplex.

To learn the specific steps involved in publishing and updating a WPF Prism application that uses dynamic module loading, see the [Publishing and Updating Applications Using the Prism Library Hands-on Lab](#).

12: Patterns in the Prism Library

The following are references and links to the patterns found in the Stock Trader RI and in the Prism Library:

- Composite pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Adapter pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Façade pattern in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Template Method pattern in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- Observer pattern in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software* (1).
- [Exploring the Observer Design Pattern](#) on MSDN.

- [Repository pattern](#) in *Patterns of Enterprise Application Architecture* by Martin Fowler or the abbreviated version on his website.
 - Inversion of Control containers and the [Dependency Injection](#) pattern on Martin Fowler's website.
 - [Plugin pattern](#) on Martin Fowler's website.
 - [Registry pattern](#) on Martin Fowler's website.
 - [Presentation Model pattern](#) on Martin Fowler's website.
 - [Event Aggregator pattern](#) on Martin Fowler's website.
 - [Separated Interface pattern](#) on Martin Fowler's website.
 - [MVC and MVP variants](#) on Martin Fowler's website.
 - [Design Patterns: Dependency Injection](#) by Griffin Caprio on MSDN.
 - [Model-View-ViewModel pattern](#) on John Gossman's blog.
-

For more information about the Unity Application Block, see "[Unity Application Block](#)" on MSDN.

(1) Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

13: Prism Library

Prism's community sites are:

- [Prism](#)
 - [PubSubEvents \(Event Aggregator\)](#)
 - [MVVM](#)
-

For more information about Unity, see the following:

- "[Unity Application Block](#)" on MSDN.
 - [Unity community site](#) on CodePlex.
-

For more information about MEF, see the following:

- "[Managed Extensibility Framework Overview](#)" on MSDN.
 - [MEF community site on](#) CodePlex.
-

For more information about service locator, see the [Common Service Locator](#) on CodePlex.