Microsoft

Composite Application Guidance for WPF

patterns & practices

Composite Application Guidance for WPF

patterns & practices

Part # X15-19255

Contents

Forewords ix
Foreword by Ian Ellison-Taylor . . . ix
Foreword by Brian Noyes . . . x

Authors and Contributors xi

Chapter 1: Introduction 1
Welcome . . . 1
Intended Audience . . . 1
Composite Application Guidance Assets . . . 2
Exploring the Documentation . . . 2
Community . . . 4
Overview of the Composite Application Guidance for WPF . . . 4
Application Development Challenges . . . 4
The Composite Approach . . . 5
Architectural Goals and Principles . . . 7
Adoption Experience . . . 8
Concerns Not Addressed by the Composite Application Guidance . . . 9
Considerations for Choosing the Composite Application Guidance . . . 9
Guidelines for Choosing Composite UI Deliverables from patterns & practices . . . 10
Comparing to the Composite UI Application Block . . . 11

Chapter 2: Design Concepts 13
UI Composition . . . 13
Layout . . . 14
Commanding . . . 17
Eventing . . . 18
Modularity . . . 20
Designing a Modular System . . . 21
Container . . . 22
Using the Container . . . 24
Considerations for Using the Container . . . 25

Chapter 3: Patterns in the Composite Application Library 27

Pattern Overview 28
Composite and Composite View 28
Separated Interface and Plug-In 30
Inversion of Control 30
Service Locator 31
Command 31
Adapter 31
Event Aggregator 32
Separated Presentation 32
Façade 32
More Information 33
Dependency Injection Pattern 34
Problem 34
Forces 35
Solution 35
Liabilities 37
Related Patterns 37
More Information 37
Inversion of Control Pattern 38
Problem 38
Forces 39
Solution 39
Implementation Details 39
Liabilities 41
Related Patterns 41
More Information 41
Service Locator Pattern 42
Problem 42
Forces 42
Solution 43
Liabilities 44
Related Patterns 44
More Information 45
Separated Presentation Pattern 45
Problem 45
Forces 45
Solution 45
Liabilities 46
Related Patterns 46
More Information 47

Supervising Controller Pattern 47
Problem 47
Forces 47
Solution 47
Liabilities 49
Related Patterns 49
More Information. 49
Presentation Model Pattern 50
Problem 50
Forces 50
Solution 50
Liabilities 52
Related Patterns 52
More Information. 52

Chapter 4: Composite Application Library 53

System Requirements 54
Composite Application Library Baseline Architecture 55
When to Use the Composite Application Library 57
A New Application Based on the Composite Application Library 58
Define the Shell 59
Create the Bootstrapper 59
Create the Module 61
Add a Module View to the Shell 61
Goals and Benefits. 61
Architectural Goals 61
Design Goals 62
Benefits 63
Organization of the Composite Application Library 64
Technical Concepts 65
Development Activities 66
Deploying Your Application 66
Customizing the Composite Application Library. 66
Guidelines for Extensibility 67
Recommendations for Modifying the Composite Application Library 68
Extensibility Points in the Composite Application Library 68

Chapter 5: Stock Trader Reference Implementation 71

The Scenario . . . 73
Operating Environment . . . 73
Operational Challenges . . . 74
Emerging Requirements . . . 74
Meeting the Business and IT Objectives . . . 75
Development Challenges . . . 76
The Solution: Composite Application Guidance for WPF . . . 76
Stock Trader RI Features . . . 77
Logical Architecture . . . 77
Implementation View . . . 79
How the Stock Trader RI Works . . . 80
Modules . . . 81
Services and Containers . . . 81
Bootstrapping the Application . . . 81
Module Enumeration . . . 82
Module Loading . . . 82
Views . . . 82
Regions and the RegionManager . . . 84
Service Registration . . . 85
Commands . . . 85
Event Aggregator . . . 87
Technical Challenges . . . 87

Chapter 6: Technical Concepts 91

Bootstrapper . . . 92
Configuring the Container . . . 93
Configuring the Region Mappings . . . 94
Creating the Shell . . . 95
Initializing the Modules . . . 96
More Information . . . 97
Container and Services . . . 97
IContainerFacade . . . 97
UnityContainerAdapter . . . 98
Considerations for Using IContainerFacade . . . 98
Composite Application Library Services . . . 99
More Information . . . 100
Module . . . 100
Team Development Using Modules . . . 101
Module Design . . . 103
IModule . . . 104
Considerations for Modules . . . 105
Module Loading . . . 105
More Information . . . 111

Region 111
Template Layout 112
Multiple View Layout 112
Working with Regions 114
Scoped Regions 117
More Information 118
Shell and View 119
Implementing a Shell 119
Stock Trader RI Shell 120
Implementing a View 121
Composite Views 122
Views in the Stock Trader RI 122
Views and Design Patterns 123
More Information 123
Event Aggregator 123
IEventAggregator 124
CompositeWpfEvent 124
More Information 128
Commands 128
DelegateCommand<T> 129
CompositeCommand 130
Frequently Asked Questions About Commands 134
More Information 135
Communication 135
Commanding 135
Event Aggregator 136
Shared Services 136

Chapter 7: Designer Guidance 139

Overview 139
Layout 140
Container Composition 140
Region 142
Visual Representation 146
User Controls 147
Custom Controls 147
Data Templates 147
Data Binding 147
Resources 149
Application Resources 150
Module Resources 151
Control Resources 151

Chapter 8: Development, Customization, and Deployment Information 153

Composite Application Guidance for WPF Hands-On Lab 153
QuickStarts 153
Development Activities 154
Creating Your Solution 155
Bootstrapper 155
Modules 156
Regions 156
Views 157
Services 157
Commands 157
Events 158
Customization Activities 158
Deployment Activities 159

Bibliography 161

Chapter 1: Introduction 161
Chapter 2: Design Concepts 162
Chapter 3: Patterns in the Composite Application Library 163
Chapter 4: Composite Application Library 164
Solution 164
Bootstrapper 164
Container and Services 165
Modules 165
Regions 165
Shell and Views 166
Events 166
Commands 166
Deployment 167
Chapter 5: Stock Trader Reference Implementation 167
Chapter 6: Technical Concepts 168
Bootstrapper 168
Container and Services 168
Modules 168
Regions 169
Shell and Views 170
Events 170
Commands 170
Chapter 7: Designer Guidance 170
Chapter 8: Development, Customization, and Deployment Information 171
Getting Started 171
Development Topics 171
Customization Topics 172
Deployment Topics 173

Forewords

Windows Presentation Foundation, WPF, is a powerful platform for writing compelling, next generation Windows-based client applications. As WPF has grown in popularity, we've seen WPF used to build an incredible range of applications, including rich media applications, social networking applications, collaboration and productivity applications, line-of-business applications, and many, many more.

Despite the wide variety of these applications, folks embarking on these frequently large, complex projects often ask a similar set of questions:

- "How should I structure my application to allow multiple development teams to work in parallel as independently as possible?"
- "How do I make sure that I can unit test my application effectively?"
- "How should I design my application so that new functionality can easily be added?"

In order to get the best out of WPF and the .NET Framework, as with any other software development technologies, finding and leveraging the techniques and patterns that have proven to be successful in the past or on other projects can bring huge benefits to a project. Unfortunately, this information is often not easy to come by.

Happily, the Microsoft patterns & practices team exists to make this kind of guidance widely available in an easy-to-understand way.

The Composite Application Guidance for WPF answers all of the above questions and more. It provides a solid foundation on which you can design and build modular WPF client applications by helping you to split the development of your application across multiple teams, so that each team can build, test, and deliver a piece of the application that can be easily integrated into the overall application. This can help you reduce the risk and complexity of your project and help get you to market faster with a better product.

This book tells you all you need to know about the patterns and underlying concepts behind the Composite Application Guidance for WPF. The patterns and techniques that this book contains will really help you get the most out of WPF. It is complemented by re-usable library code, a reference implementation, and QuickStart tutorials that provide ready-to-use implementations of the patterns and techniques that this book contains, so you can focus on your specific application functionality right from the start.

But remember, the important thing is to have fun with WPF!

Ian Ellison-Taylor
General Manager, Windows Presentation Foundation
Microsoft Corporation
September 2008

Learning WPF on its own is a daunting task because there are so many new capabilities and constructs in the WPF architecture. However, even once you master what all those new things are, you are still at square one in figuring out how to put them all together to design a robust, scalable, loosely coupled, testable, and maintainable UI application. In the first half of 2008, I had the pleasure of working with the Microsoft patterns & practices Composite Application Guidance for WPF team to come up with some guidance and code to help you do just that. Working with such a top notch team was its own reward, but setting aside my own bias from having contributed to the result, I truly believe the set of guidance that came out of the whole team's effort is a huge step forward for teams building complex applications with WPF.

The Composite Application Library that comes with the guidance gives you a great starting point for building modular, loosely coupled UI applications in WPF. It includes code that helps you to set up the execution environment for your application, dynamically load modules that your application is composed of, dynamically inject views into placeholder locations in your UI layout, hook up and handle commands in a way that goes well beyond the capabilities of WPF itself, and have a loosely coupled publish/subscribe events mechanism that supports thread dispatching and filtered subscriptions. You can follow and employ the UI design patterns demonstrated in the QuickStarts and Stock Trader Reference Implementation application resulting in a well-factored and testable application architecture with a strong focus on separation of concerns. The code alone that comes with the guidance should allow you to end up with a much better architecture than if you just stick with the standard XAML and code behind model of WPF. You can use the Composite WPF guidance to build an application from scratch utilizing all the features in the Composite Application Library, or you can pick and choose just the parts that you like—you can even easily integrate them into an existing application well into the development process.

Even though the code speaks for itself to a large degree, it is always important to have a written version of the story that you can consume to get the big picture—that way, you can see how everything flows together in a logical way. It is also useful to have a written reference to fall back on to quickly locate and refresh yourself on the concepts at points in the future. Even though the Help topics that accompany the guidance provide one form of that, this book should satisfy all those needs for you and help make the Composite Application Guidance for WPF quick and easy to understand and employ, whether or not you are sitting at your computer. The team has put together some great content and explanations in this book; I'm sure you will find it invaluable.

Brian Noyes
Chief Architect, IDesign
July 2008

Authors and Contributors

The Composite Application Guidance for WPF was produced by the following individuals:

patterns & practices Team:

Blaine Wastell, Bob Brumfield, David Hill, Erwin van der Valk, Francis Cheung, Glenn Block, Larry Brader, Nelly Delgado, Alex Homer (Microsoft Corporation)

Brian Noyes (iDesign)

Adam Calderon (Interknowlogy LLC)

Arun Subramonian Namboothiri, Gokul Janardhanan, Padmavathy Bharathy Jambunathan, Prashant Javiya, Prasad Paluri (Infosys Technologies Ltd)

Damian Schenkelman, Diego Poza, Ezequiel Jadib, Ignacio Baumann Fonay, Jonathan Cisneros, Julian Dominguez, Mariano Converti, Mariano Szklanny, Matias Woloski (Southworks)

Tina Burden McGrayne (TinaTech, Inc.)

Veronica Ruiz (CXR Design)

Many thanks to the following advisors who provided invaluable assistance and feedback:

Bil Simser, Brad Abrams (Microsoft Corporation), Chad Myers, Clifford Tiltman (Morgan Stanley), David S Platt (Rolling Thunder Computing, Inc.), Derek Greer, Ian Ellison-Taylor (Microsoft Corporation), Ivo Manolov (Microsoft Corporation), Jamie Rodriguez (Microsoft Corporation), Jeremy D. Miller (Dovetail Software), John Gossman (Microsoft Corporation), Josh Twist (Microsoft Corporation), Matt Smith (AltiMotion Corporation), Mark Feinholz (Microsoft Corporation), Mark Tucker (JDA Software Group, Inc.), Michael D. Brown (Software Engineering Professionals, Inc.), Michael Kenyon (IHS, Inc.), Michael Sparks (RDA Corp), Ohad Israeli (Hewlett-Packard), Oren Eini (aka Ayende Rahien), Peter Lindes (The Church of Jesus Christ of Latter-day Saints), Rob Eisenberg (Blue Spire Consulting, Inc.), Shanku Niyogi (Microsoft Corporation), Scott Bellware, Szymon Kobalczyk (InterKnowlogy), Udi Dahan (The Software Simplist), Varghese John (UBS), Ward Bell (IdeaBlade)

1

Introduction

Welcome

The Composite Application Guidance for WPF is a set of guidance designed to help you more easily manage the complexities you may face when building enterprise-level Windows Presentation Foundation (WPF) client applications. This guidance will help you design and build flexible WPF client applications using loosely coupled, independently evolvable pieces that work together and are integrated into the overall application. This type of application is known as a composite application.
The guidance includes a reference implementation, reusable library code (named the Composite Application Library), and supporting documentation. This documentation, as well as the source code, is available on the MSDN® developer network at *http://www.microsoft.com/CompositeWpf*.

Intended Audience

This guidance is intended for software architects and software developers building complex WPF applications with functionality separated across multiple teams. Such complex applications typically feature layered architectures that may be physically deployed across tiers, strong separation of concerns, and loosely coupled components. Simple applications that do not have some of these requirements may not benefit as much from the Composite Application Library.

The Composite Application Library is built on the Microsoft® .NET Framework and Windows Presentation Foundation, and it uses a number of software design patterns. Familiarity with these technologies and patterns is useful for evaluating and adopting the Composite Application Library. For more information about the patterns, see Chapter 3, "Patterns in the Composite Application Library."

Composite Application Guidance Assets

The Composite Application Guidance consists of the following:

- **Composite Application Library source code.** Developers can use the Composite Application Library to develop WPF applications that are composed of independent and collaborating modules.
- **Unity Extensions for Composite Application Library source code.** This provides components to use the Unity Application Block with the Composite Application Library for WPF.
- **Unity Application Block binaries.** The Composite Application Library itself is not container-specific; however, the Stock Trader Reference Implementation uses the Unity Application Block as the container.
- **Stock Trader Reference Implementation (Stock Trader RI).** This is a composite application that is based on a real world scenario. This intentionally incomplete application illustrates the composite application baseline architecture. This is a good reference to see how many of the challenges when building composite applications are addressed by this guidance.
- **QuickStarts and Hands-On Lab.** This includes the source code for several small, focused applications that illustrate user interface (UI) composition, dynamic modularity, commanding, and event aggregation. The Hello World Hands-On Lab provides a step-by-step hands-on lab to create your first application using the Composite Application Library.
- **Documentation.** This includes the architectural overview, Stock Trader RI overview, design and technical concepts for composite applications, applied patterns, How-to topics, QuickStart overviews, and deployment topics. Much of this guidance is applicable even if you are not using the Composite Application Library, but you just want to know best practices for creating composite applications.

Exploring the Documentation

The documentation spans a wide range of topics from the conceptual drivers for composite applications to step-by-step instructions for using pieces of the Composite Application Library. The documentation is intended to appeal to a broad technical audience to help you understand composite scenarios, evaluate the Composite Application Library, and use the Composite Application Library. Figure 1.1 maps the types of documentation available in this book and on MSDN.

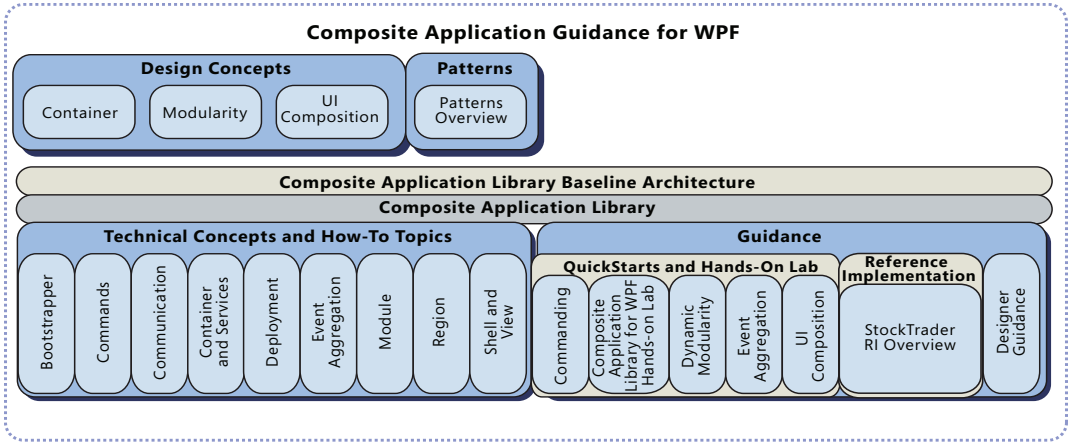


Figure 1.1 *Composite Application Guidance for WPF documentation*

As illustrated in Figure 1.1, the documentation consists of the following:

- **Chapter 2, "Design Concepts."** This chapter introduces the key challenges in building composite applications and guidance on the solutions designed to solve these challenges. It primarily targets architects and developers seeking a deeper understanding of the drivers for the Composite Application Library.
- **Chapter 3, "Patterns in the Composite Application Library."** This chapter describes the software design patterns applied in the Composite Application Library and Stock Trader RI. It primarily targets architects and developers looking to familiarize themselves with the patterns used to address the challenges in building composite applications.
- **Chapter 4, "Composite Application Library."** This chapter introduces a candidate composite architecture that you can use to create your own baseline architecture and describes the goals, benefits, development activities, and customization points of the library. It primarily targets architects and developers interested in using the library and seeking a starting point for their composite applications.
- **Chapter 5, "Stock Trader Reference Implementation."** This chapter describes the Stock Trader RI, which is an intentionally feature-incomplete application that demonstrates an implementation of the baseline architecture using the Composite Application Library. It primarily targets architects and developers wanting to see the library working as part of an application.
- **Chapter 6, "Technical Concepts."** This chapter introduces individual technical implementation details of the Composite Application Library. It primarily targets architects and developers seeking in-depth information about a particular aspect of the library.

- **Chapter 7, "Designer Guidance."** This chapter helps user interface designers understand composite applications. It provides helpful tips for designing composite user interfaces with the Composite Application Library. It primarily targets designers working on projects developing composite applications.
- **Chapter 8, "Development, Customization, and Deployment Information."** This chapter includes a brief introduction and pointers to the Hands-On Lab, Quick-Starts, and How-to topics on MSDN. These topics range from usage to customization and deployment of the library. These topics primarily target developers seeking to accomplish specific tasks or run working examples.
- **"Bibliography."** This section consolidates the references provided in each chapter.

Community

The Composite Application Guidance for WPF, like other Microsoft patterns & practices deliverables, is associated with a community site at *http://www.codeplex.com/CompositeWPF*. On this community site, 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.

Overview of the Composite Application Guidance for WPF

The goal of this section is to provide you with a high-level overview of the Microsoft patterns & practices Composite Application Guidance for WPF and the development challenges it addresses. This section describes some of the problems you might encounter when building complex WPF client applications, how the Composite Application Guidance helps you to address those problems, and how the Composite Application Guidance compares to alternative approaches.

Application Development Challenges

Typically, developers of client applications face a number of challenges. Most enterprise applications are sufficiently complex that they require more than one developer, maybe even a large team of developers that includes 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 sub-teams can work effectively on different pieces of the application independently, yet ensuring that the pieces come together seamlessly when integrated into the application.

In addition, 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 maintainability 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.

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 a number of 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 named composite applications.

Composite applications provide many benefits, including the following:

- They allow modules to be individually developed, tested, and deployed by different individuals or sub-teams. Modules can be modified or extended with new functionality more easily, thereby allowing the application to be more easily extended and maintained.
- 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 re-use 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.
- They help maintain a separation of roles by allowing different individuals or sub-teams 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 user interface 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 a number of disparate back-end systems. Figure 1.2 shows an example of this type of a composite application.

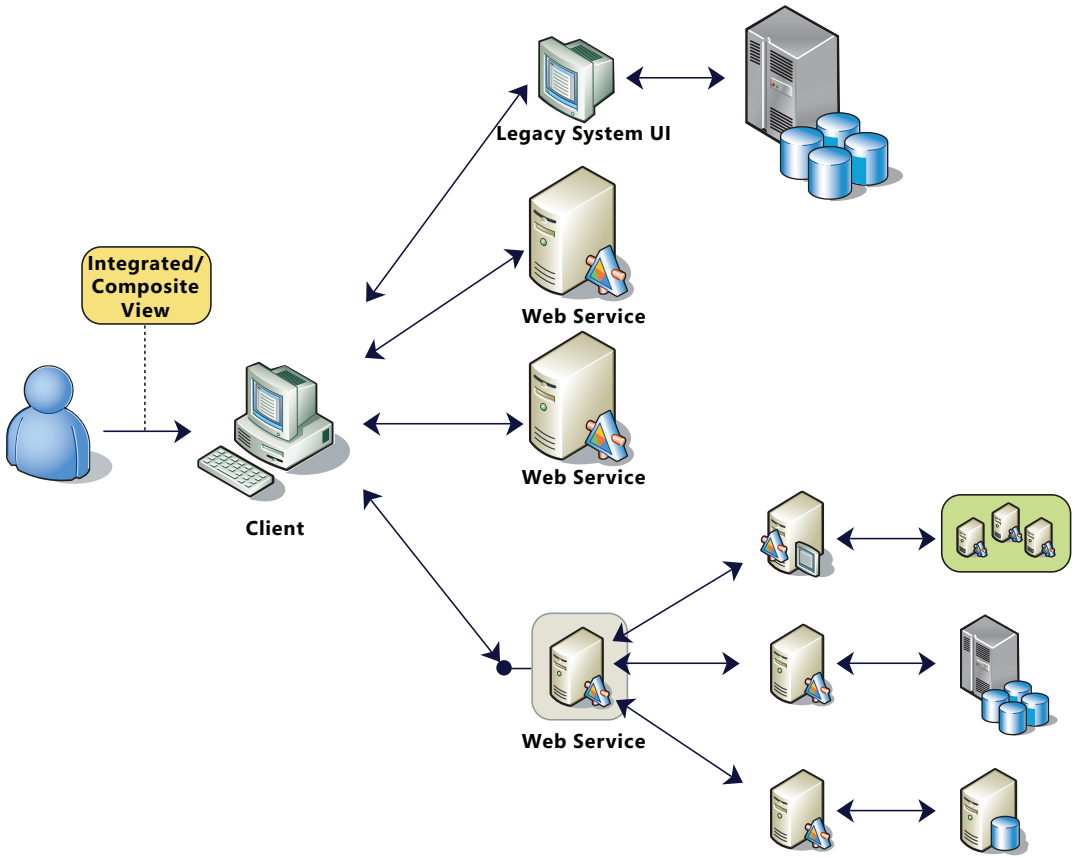


Figure 1.2 *Composite application with multiple back-end systems*

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 user interface allows the application to provide a consistent and differentiated appearance (look and feel) 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. Figure 1.3 shows a screen shot of this type of application. Each of the areas highlighted represent independent components that are composed into the UI.

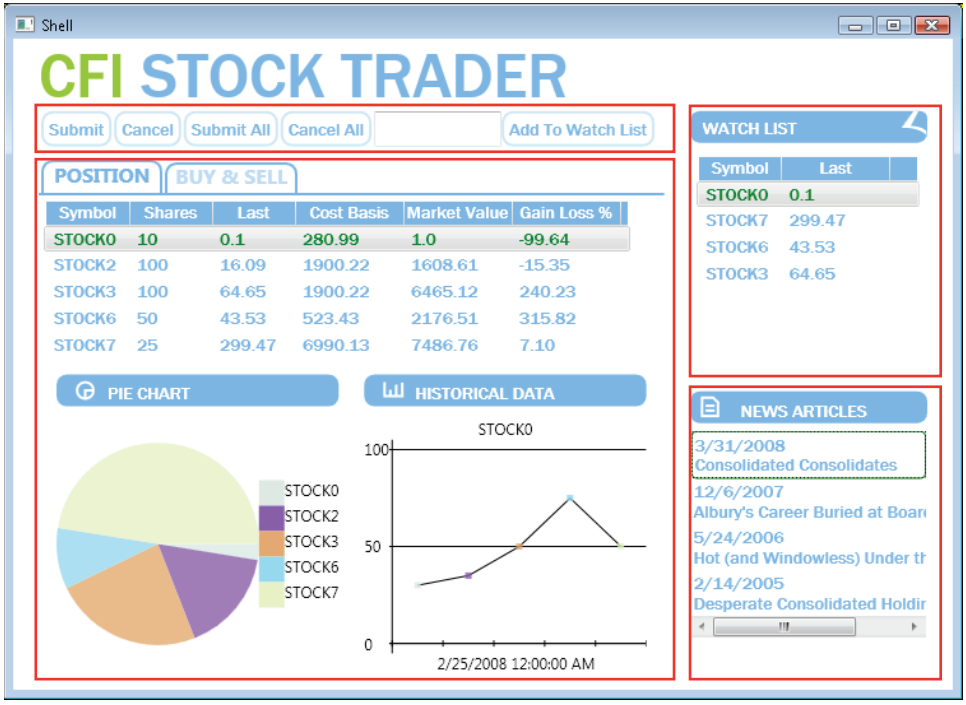


Figure 1.3 *Stock Trader RI composite application*

In this case, the composite allows the application's user interface to be dynamically 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.

Architectural Goals and Principles

The following table lists, approximately in priority order, the architectural principles that have been prioritized by the Composite Application Guidance team's customer advisory board. These principles determine how the guidance was developed and what the focus areas were.

Quality	Definition
Subsetability	This is the ability to adopt a portion of the Composite Application Library. You can choose only certain capabilities, incrementally adopt capabilities, and enable or disable features.
Learnability	This is the ability to quickly learn how to build WPF composite applications using the Composite Application Library. With small digestible and independent capabilities, you can start building your applications faster.
Extensibility	This is the ability to enhance, extend, or replace pieces of the Composite Application Library without requiring you to redesign the library or your application.
Compatibility	This means you can adopt the Composite Application Library for an existing application and you can use it with other existing infrastructure. Core services are swappable.
Simplicity	This means the Composite Application Library is designed in a minimalist way to reduce the amount of complexity. The Composite Application Library strives for the simplest approach to get the job done.
Testability	The reference implementation in the Composite Application Guidance provides an implementation for Separated Presentation patterns. These patterns allow UI logic to be tested.
Performance	The Composite Application Library minimizes overhead while the application is running.
Scalability	The application can scale to support increased load.
Upgradeability	Existing WPF applications can be upgraded to use the Composite Application Library.

Adoption Experience

The Composite Application Guidance has an explicit goal to provide a good adoption experience. To deliver on this goal, the Composite Application Guidance provides the following:

- You can "opt in" and "opt out" of the Composite Application Library capabilities. For example, you can consume only services you need.
- You can incrementally add the Composite Application Library capabilities to your existing WPF applications.
- It is non-invasive because of the following:
  - It limits the Composite Application Library footprint in the code.
  - It limits reliance on custom Composite Application Library attributes. You can integrate existing libraries with the Composite Application Library through a design that favors composition over inheritance (this avoids forcing you to inherit from the classes in the Composite Application Library).

Concerns Not Addressed by the Composite Application Guidance

Please note that the Composite Application Guidance does not directly address the following:

- Occasional connectivity
- Messaging infrastructure, including the following:
  - Client/server communication
  - Asynchronous communication
  - Encryption
- Application performance
- Authentication and authorization
- Handling thread-safety from background updates, including the following:
  - Data races
  - Data synchronization
  - Handling UI updates from multiple threads (the Composite Application Library addresses some aspects of this)
- Versioning
- Error handling and fault tolerance

For more information about these topics, see the following resources:

- "Smart Client Architecture and Design Guide"
- "Occasionally Connected Systems Architecture: The Client"
- "Occasionally Connected Systems Architecture: Concurrency"

Considerations for Choosing the Composite Application Guidance

The Composite Application Guidance is for designing complex WPF applications. The scenarios where you should consider using the Composite Application Guidance include the following:

- You are building a composite application that presents information from multiple sources through an integrated user interface.
- You are developing, testing, and deploying modules independently of the other modules.
- Your application is being developed by multiple collaborating teams.

If your applications do not require one or more of these scenarios, the Composite Application Guidance may not be right for you. The Composite Application Guidance requires you to have hands-on experience with WPF. If you need general information about WPF, see the following sources:

- "Windows Presentation Foundation" on MSDN
- Sells, Chris and Ian Griffiths. *Programming WPF: Building Windows UI with Windows Presentation Foundation*. Second Edition. O'Reilly Media, Inc., 2007.
- Nathan, Adam. *Windows Presentation Foundation Unleashed*. Indianapolis, IN: Sams Publishing, 2006.

Guidelines for Choosing Composite UI Deliverables from patterns & practices

The patterns & practices team has several deliverables for building composite applications. The following list highlights the scenarios where you should consider using each, along with advantages and disadvantages to each approach:

- **If you need to develop a composite Windows Forms application, consider using the Smart Client Software Factory:**
  - It leverages the most mature platform.
  - It offers a good development and debugging experience (including tooling, control support, drag and drop, and rapid application development experience).
  - It includes code generation (this is provided with guidance package support and recipes).
  - It includes capabilities for occasionally connected clients.
- **If you need to develop a composite Windows Forms application with islands of WPF content, consider using the Smart Client Software Factory's WPF interop support:**
  - It can be leveraged with an existing Smart Client Software Factory infrastructure.
  - It includes guidance automation (in the form of guidance package support and recipes).
- **If you want to upgrade an existing Composite UI Application Block application to WPF, consider using the WPF/CAB layer in the Smart Client Contrib CodePlex project:**
  - It allows building pure WPF-based applications with the Composite UI Application Block.
  - It can be leveraged with an existing Smart Client Software Factory infrastructure, except for views.
  - It is not maintained or owned by the patterns & practices team; instead, it is supported by community.

  - It is a literal port of Composite UI Application Block to WPF and is not optimized to take advantage of WPF.
  - It does not include guidance automation.
- **If you want to create a composite application with WPF or upgrade an existing WPF application to a composite application, use Composite Application Guidance for WPF:**
  - It targets WPF composite application development.
  - It is designed to best use WPF capabilities.
  - It includes a lightweight library—you can choose the library capabilities you want to include.
  - It integrates with existing applications and libraries.
  - It does not include guidance automation.

Comparing to the Composite UI Application Block

This is not simply a new version of the patterns & practices Composite UI Application Block. It is a new set of libraries and guidance, built from the ground up, that is designed to help you develop new WPF composite applications. Although it is not a new version of the Composite UI Application Block, it uses the core concepts of the Composite UI Application Block, such as modularity, UI composition, services, dependency injection, and event brokering. These concepts are essential for building composite applications and Composite Application Guidance uses them; however, the implementation differs from the Composite UI Application Block for several reasons:

- **Composite Application Guidance incorporates customer feedback**. Over the years, the patterns & practices team has received great feedback, positive and negative, about the Composite UI Application Block implementation. Some of the negative feedback includes that it is too heavy, too complicated, too tightly coupled, and too hard to get going. Additionally, customers expressed a need for an approach that allows incremental adoption of the library and to work with existing libraries. The patterns & practices team determined that the best way to address the concerns and tackle the new ideas was with a clean break.
- **The Composite UI Application Block was not built to support WPF**. Although you can port the Composite UI Application Block to WPF, the application block was not built to take advantage of WPF's core functionality. In many instances, the application block introduced mechanisms that are now native to WPF. For example, WPF introduces attached properties that aid in UI composition in a lighter-weight way than what existed in the application block. WPF is also an inherently different paradigm than Windows Forms. There are many flexible ways to compose your UI over the traditional way of using controls. WPF also introduces the idea of using templates to control the way your UI renders.

- **The Composite UI Application Block relies on the Windows Forms development experience**. Composite UI Application Block development scenarios depend on the tooling and productivity experience provided for Windows Forms by Microsoft Visual Studio® development system. Currently, the WPF developer experience is a very different paradigm. In Visual Studio 2008, a good portion of WPF application development still requires manually working with XAML. The built-in Visual Studio designers provide only a small subset of the capabilities Windows Forms developers are accustomed to. Tools such as Microsoft Expression Blend™ offer some of these capabilities, but they are not targeted for developers and do not integrate into the development environment. This experience will be improved in future versions of the platform. Based on the current state of WPF development, the transition from Windows Forms to WPF currently requires substantial effort and developers face a steep learning curve. For these reasons, this guidance is optimized for new composite application development in WPF.

For more information about the Composite Application UI Block, see "Composite UI Application Block" on MSDN.

2

Design Concepts

This chapter discusses the core design concepts for building composite applications. The design concepts are the following:

- UI Composition
- Modularity
- Container

These design concepts are important to your understanding of the technical concepts that the Composite Application Guidance for WPF (Windows Presentation Foundation) is based on.

UI Composition

Composite applications typically compose their user interfaces (UIs) from various loosely coupled visual components, otherwise known as views, to deliver an integrated application experience. To the user, the application appears as a seamless program that offers many capabilities. For example, the Stock Trader Reference Implementation (Stock Trader RI) has the views illustrated in Figure 2.1.

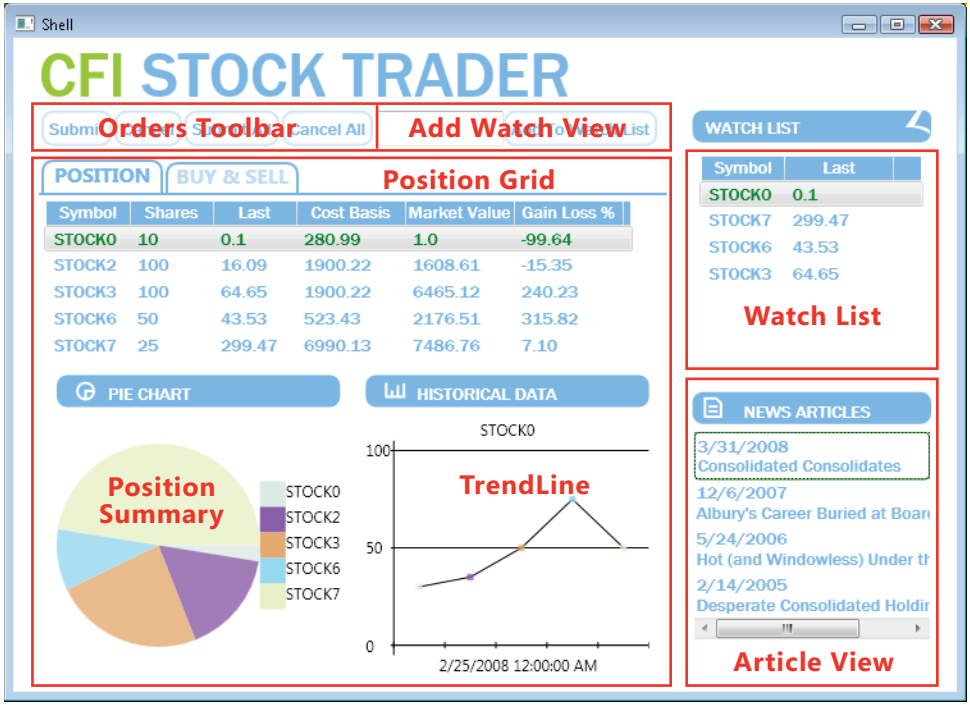


Figure 2.1 *Stock Trader RI views*

There are many different challenges for composing the UI. The following are common challenges that are illustrated in the Stock Trader RI and enabled through the Composite Application Library.

- Layout
- Commanding
- Eventing

The next sections describe each of these in greater detail.

Layout

In a composite application, views from multiple modules are displayed at run time. Because of this, the application needs a mechanism for specifying how they are laid out. There are several approaches to do this.

View Injection

In this approach, the application contains a registry of reserved locations in the UI. A module can access one of the locations in the registry and use it to inject views. The view being injected does not have any specific knowledge of how it will be displayed in that location. The place it is being injected is referred to by a moniker, such as the name. Each of the objects in the registry implements a specific interface that is used to inject the view. Figure 2.2 illustrates the View Injection approach to dynamic layout.

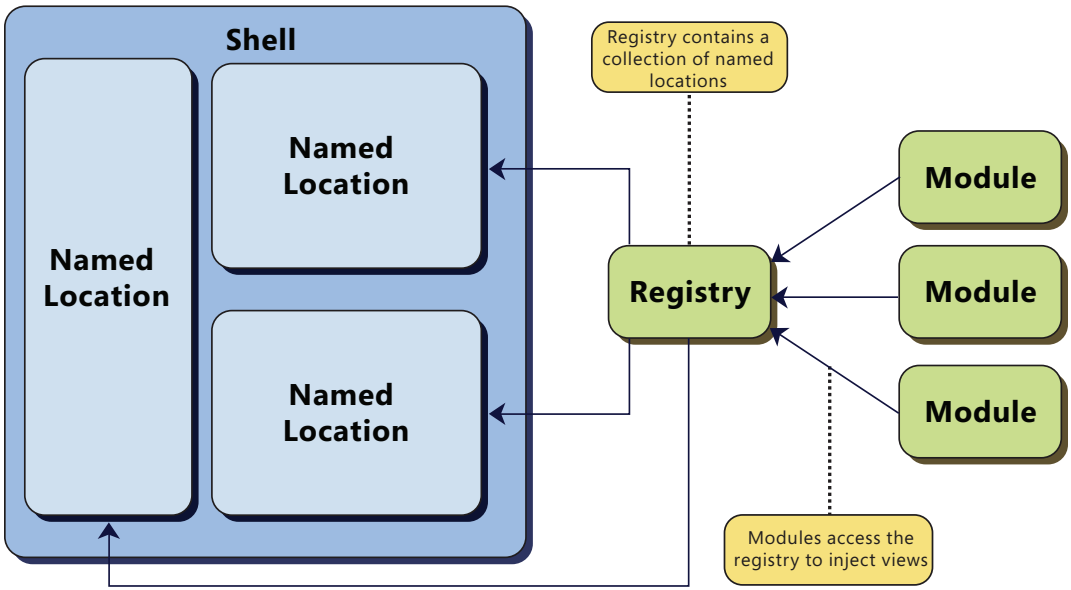


Figure 2.2 *View injection*

The Composite Application Library introduces the concept of a region. Regions are defined through attached properties in XAML. Together, regions and the region manager implement view injection. The Stock Trader RI uses regions for composing its UI. For more information about regions, see "Region" in Chapter 6, "Technical Concepts," and "UI Composition QuickStart" on MSDN.

View Discovery

With View Discovery, the modules register their views (or presenters) in a central location, such as in the container, using a well-known interface. A shell service or a composite view then queries the container to discover the views that were registered. After they are discovered, the service lays out those views on the screen as appropriate, such as adding them to a panel or an items control. If, after the application is loaded, additional views need to be displayed, such as a new order screen, the shell service or composite view should be notified to handle the display. Figure 2.3 illustrates the View Discovery approach to dynamic layout.

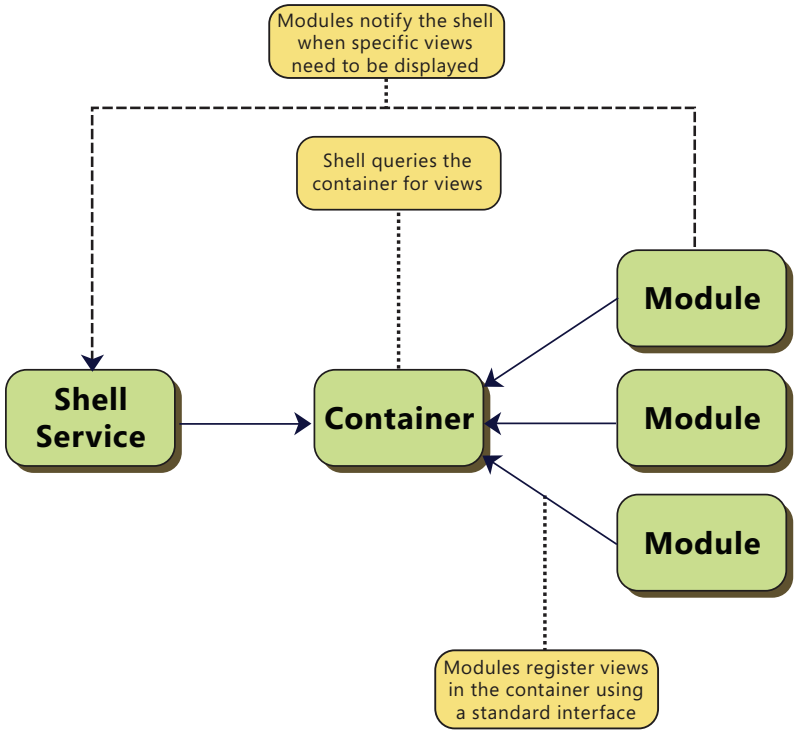


Figure 2.3 *View discovery*

In the Stock Trader RI, the **TrendLinePresenter** is registered in the container by the market module and retrieved from the container by the **PositionSummaryPresentationModel**. The **PositionSummaryPresentationModel** then calls a method on its view (which is a composite view) to handle the display of the trend line.

Commanding

In a composite WPF application, separated presentation patterns, such as Model-View-Presenter, PresentationModel, and Model-View-ViewModel, are used for decoupling the view from the business logic. How then, are actions within the view routed to the appropriate handlers outside of the view? How are the UI elements associated with those actions enabled or disabled based on state changes within the application?

WPF introduces the concept of commands to allow this to occur. UI elements can be bound to a command, which handles the execution logic. After it is bound, it can execute the command, and the element will be automatically enabled or disabled along with the command. The default **RoutedUICommand** mechanism requires event handlers to be defined in the receiving view. **RoutedUICommand**s can also only be received by UI elements in the visual tree, which is often not the case in a composite application. Additionally, there are complex scenarios in composites where the handling of the command is delegated to child commands.

To overcome this constraint, you can use WPF to create custom **ICommand**s so that you can directly route the handling logic. Two common approaches are delegation and composition.

Delegation

This method uses a command that delegates off its handling logic, either through events or delegates where it can be handled externally by a class such as a presenter, service, controller, and so on. This provides the benefit of not having to put any code in the code behind. The command requires two handlers: one for the **Execute** method and one for the **CanExecute** method. Whenever the **Execute** or **CanExecute** methods are called on the command, the handlers are called either through the event being raised or the delegate being invoked. Figure 2.4 illustrates the Delegation approach to commanding.

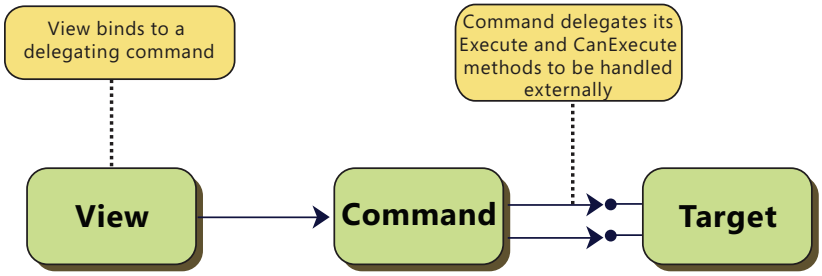


Figure 2.4 *Delegation*

Composition

Composition is a variation of delegation. In this approach, a composite command delegates its handling logic to a set of child commands, such as in a **Save All** scenario described earlier. The composite command needs to provide a way for the child commands to be registered. Executing the composite command executes the children. The composite commands **CanExecute** returns **false**, unless all the children return **true**.

The Composite Application Library introduces the **DelegateCommand<T>** and **CompositeCommand** classes as implementations of these approaches. For more information about commands, see "Commands" in Chapter 6, "Technical Concepts," and "Commanding QuickStart" on MSDN. Figure 2.5 illustrates the Composition approach to commanding.

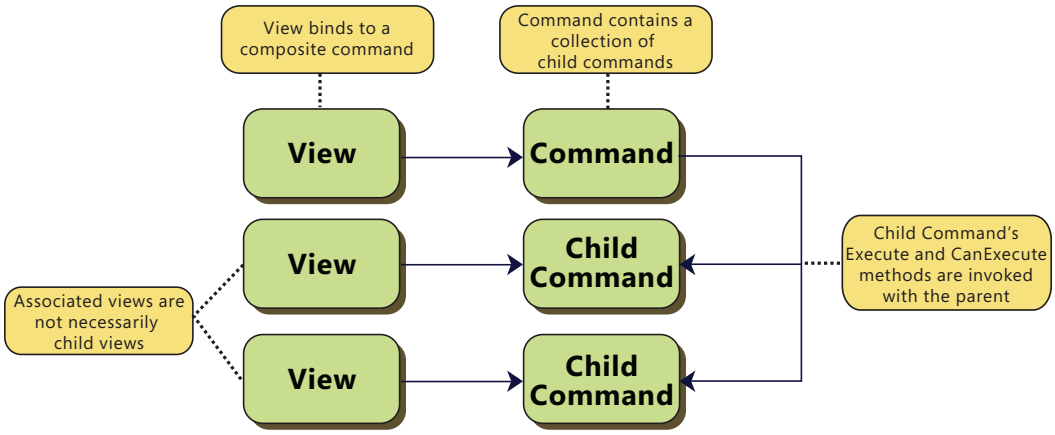


Figure 2.5 *Composition*

Eventing

In a composite application, components, such as presenters, services, and controllers, residing in different modules often need to communicate with one another based on state changes. This is a challenge due to the highly decoupled nature of a composite application because the publisher has no connection to the subscriber. Additionally, there may be threading issues because the publisher is on a different thread than the subscriber.

The Publish/Subscribe pattern addresses these challenges. There are several ways to implement the pattern. Two approaches used in the Composite Application Guidance are Event Services and Event Aggregation.

Event Services

In this method, an application-specific service raises standard .NET Framework events. To add new events, the service and service interface need to be modified. This service is registered in the container where it can be accessed by the different modules in the system. The publisher and the subscriber reference the service interface and do not depend on one another. Using this approach, the subscriber needs to manually handle any thread marshaling concerns and handle unregistering itself from the event so that it can be garbage collected. Figure 2.6 illustrates the Event Services approach.

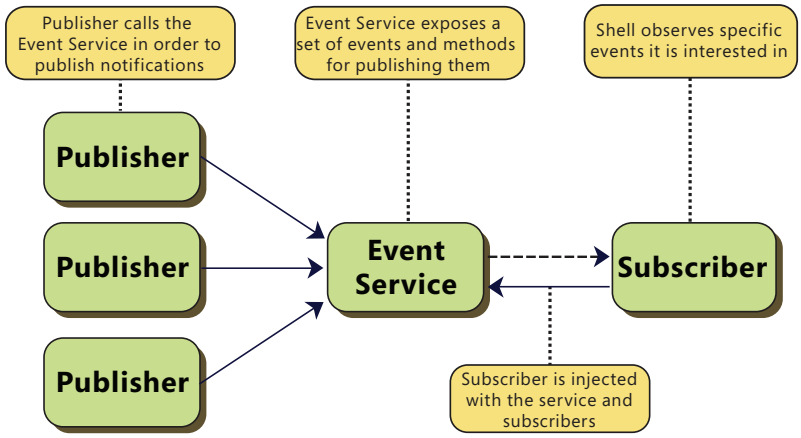


Figure 2.6 *Event Services*

Event Aggregation

This approach uses a generic event aggregator service registered in the container that holds a repository of event objects. The event object itself uses delegates instead of events. The advantage of this is that these delegates can be created at the time of publishing and immediately released, which does not prevent the subscribers from being garbage collected. Each event object contains a collection of subscribers it will publish to. New events can be added to the system without modifying the service. The event object can also automatically handle marshaling to the correct thread.

The **EventAggregator** service and **CompositeWpfEvent** class are implementations that exist in the Composite Application Library. For more information, see "Event Aggregator" in Chapter 6, "Technical Concepts," and "Event Aggregation QuickStart" on MSDN. Figure 2.7 on the next page illustrates the Event Aggregation approach.

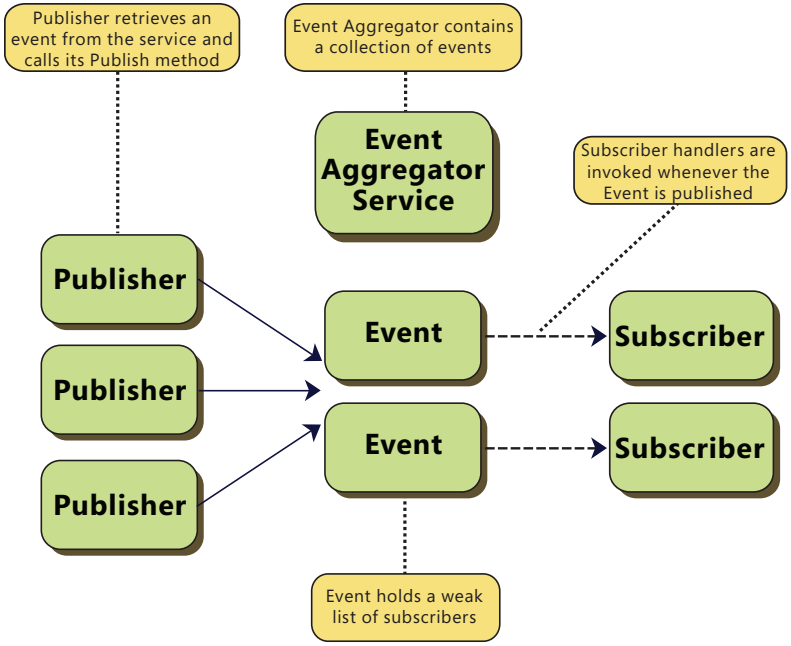


Figure 2.7 *Event Aggregation*

Modularity

Modularity is designing a system that is divided into a set of functional units (named modules) that can be composed into a larger application. A module represents a set of related concerns. It can include components, such as views or business logic, and pieces of infrastructure, such as services for logging or authenticating users. Modules are independent of one another but can communicate with each other in a loosely coupled way.

A composite application exhibits modularity. Imagine an online banking program. The user can access a variety of functions, such as transferring money between accounts, paying bills, and updating personal information from a single UI. However, behind the scenes, each of these functions is a discrete module. These modules communicate with each other and with back-end systems such as database servers. Application services integrate components within the different modules and handle the communication with the user. The user sees an integrated view that looks like a single application.

Figure 2.8 illustrates a design of a composite application with multiple modules.

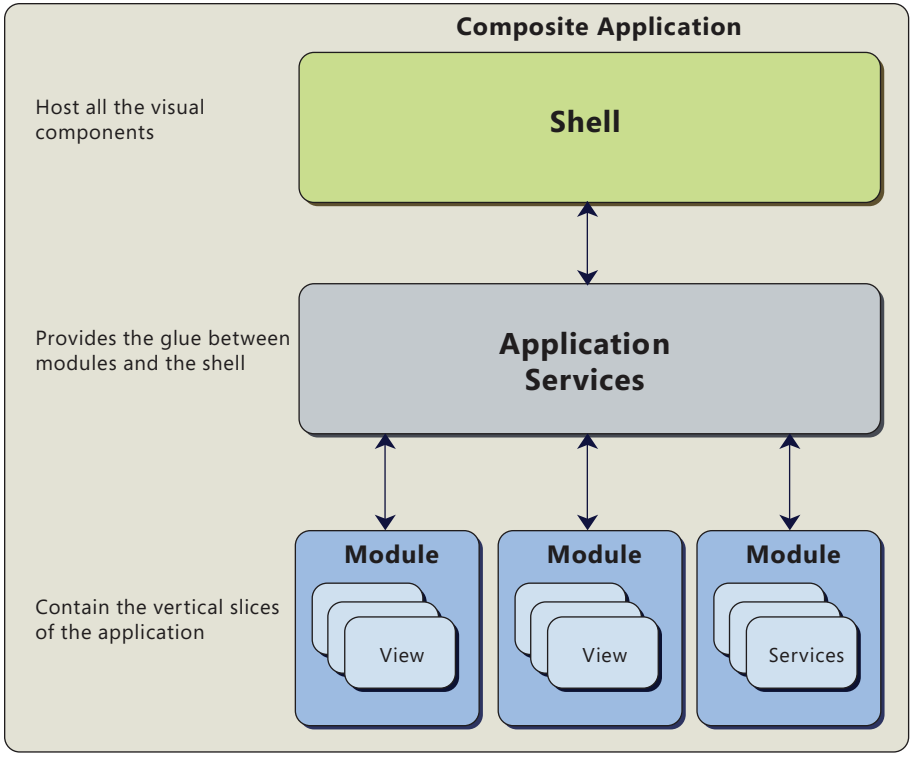


Figure 2.8 *Module composition*

Designing a Modular System

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. Modules can enforce separation of concerns by vertically partitioning the system and keeping a clean separation between the UI and business functionality. Not having modularity makes it difficult for the team to introduce new features and makes the system difficult to test and to deploy.

The following are specific guidelines for developing a modular system:

- Modules should be opaque to the rest of the system and initialized through a well-known interface.
- Modules should not directly reference one another or the application that loaded them.
- Modules should use services to communicate with the application or with other modules.

- Modules should not be responsible for managing their dependencies. These dependencies should be provided externally, for example, through dependency injection.
- Modules should not rely on static methods that can inhibit testability.
- Modules should support being added and removed from the system in a pluggable fashion.

For more information about modules, see "Module" in Chapter 6, "Technical Concepts," and "Dynamic Modularity QuickStarts" on MSDN.

Container

Applications based on the Composite Application Library are composites that potentially consist of many loosely coupled modules. They need to interact with the shell 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 modules, applications based on the Composite Application Library rely on a dependency injection container. The container offers a collection of services. A service provides functionality to other modules in a loosely coupled way through an interface and is often a singleton. The container creates instances of components that have service dependencies. During the component's creation, the container injects any dependencies that the component has requested into it. If those dependencies have not yet been created, the container creates and injects them first. In some cases, the container itself is resolved as a dependency. For example, modules will often register views of the container by having the container injected.

There are several advantages of using a container:

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

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.

In the context of an application based on the Composite Application 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 presenters and views.
- A container creates presenters and presentation 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: The Stock Trader RI and QuickStarts rely on the Unity Application Block as the container; this is referred to as a "Unity container." The Composite Application 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.

The following code shows how injection works. When the **PositionModule** is created by the container, it is injected with the **regionManager** and the container. In the **RegisterViewsAndServices** method, which is called from the module's **Initialize** method, when the module is loaded, various services, views, and presenters are registered.

C# PositionModule.cs

```
public PositionModule(IUnityContainer container, IRegionManager regionManager)
{
    _container = container;
    _regionManagerService = regionManager;
}

public void Initialize()
{
    RegisterViewsAndServices();
    ...
}

protected void RegisterViewsAndServices()
{
    _container.RegisterType<IAccountPositionService, AccountPositionService>_
        (new ContainerControlledLifetimeManager());
    _container.RegisterType<IPositionSummaryView, PositionSummaryView>();
    _container.RegisterType<IPositionSummaryPresentationModel, PositionSummary_
        PresentationModel>();
    ...
}
```

Using the Container

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

Registering

For services to be available to be injected, they need to be registered with the container. Registering a service involves passing the container a service interface and a concrete type that implements that service. There are primarily two means for registering services: through code or through configuration. The specific means vary from container to container. In the previous code example, you can see how the **AccountPositionService** is registered with the **IAccountPositionService**. For an example of entering configuration information through the Unity container, see "Entering Configuration Information" on MSDN.

Services have a lifetime. This can be either a singleton (a single instance for the container) or an instance. The lifetime can be specified at registration or through configuration. The default lifetime depends on the container implementation. For example, the Unity container will register services as instance by default.

Resolving

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

In general, when a service is resolved, one of three things will happen:

- If the service has not been registered, the container will throw an exception.

> **Note:** Some containers will allow you to resolve a concrete type that has not been registered.

- If the service has been registered as a singleton, the container will return the singleton instance. If this is the first time the service was called for, the container will create it and hold on to it for future calls.
- If the service has not been registered as a singleton, the container will return a new instance and will not hold on to it.

The following code example shows where the **IPositionSummaryPresentationModel** is being resolved by the container.

C# PositionModule.cs

```
IPositionSummaryPresentationModel presentationModel = _container.Resolve_
    <IPositionSummaryPresentationModel>();
```

The **PositionSummaryPresentationModel** constructor contains the following dependencies which are injected when it is resolved.

C# PositionSummaryPresentationModel.cs

```
public PositionSummaryPresentationModel(
            IPositionSummaryView view,
            IAccountPositionService accountPositionService,
            IMarketFeedService marketFeedSvc,
            IMarketHistoryService marketHistorySvc,
            ITrendLinePresenter trendLinePresenter,
            IOrdersController ordersController,
            IEventAggregator eventAggregator)
```

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 in 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 the container uses 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.
- Consider whether a component's lifetime should be registered as a singleton or an 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 Presentation Model needs a new instance of a view.
- 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.

3

Patterns in the Composite Application Library

Building composite applications is a complex endeavor that involves applying different patterns. This section describes design patterns that the Composite Application Guidance for WPF (Windows Presentation Foundation) team encountered when building composite user interface (UI) applications. Figure 3.1 shows a typical composite application architecture using the Composite Application Library and some of the common patterns.

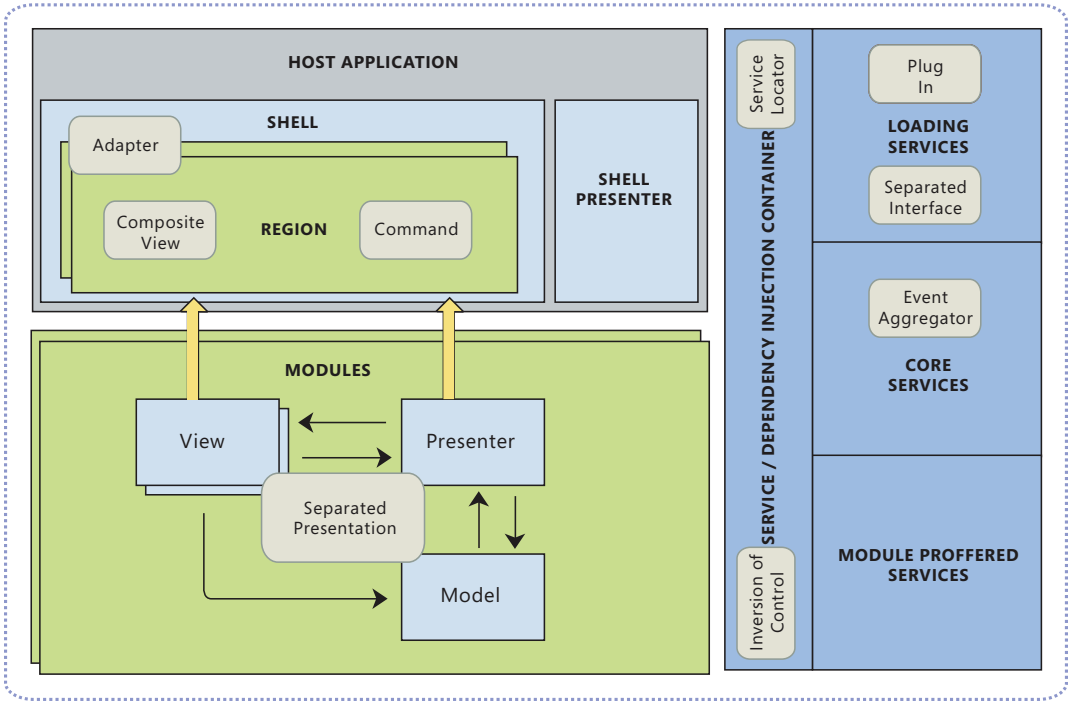


Figure 3.1 *Composite application patterns*

Pattern Overview

This section provides a brief overview of each listed pattern and an area in the Composite Application Guidance code to see an example of this pattern. The following patterns are described:

- Composite User Interface patterns:
  - Composite and Composite View
  - Command
  - Adapter
- Modularity patterns:
  - Separated Interface and Plug In
  - Service Locator
  - Event Aggregator
  - Façade
- Testability patterns:
  - Inversion of Control
  - Separated Presentation

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 Figure 3.2 on the next page.

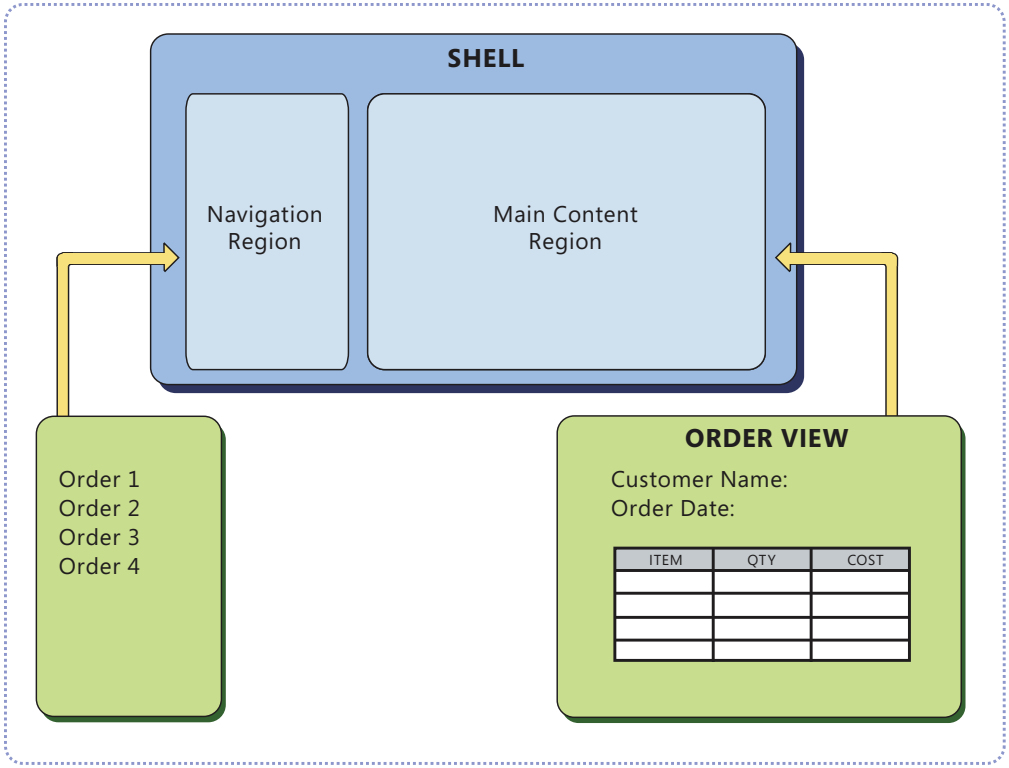


Figure 3.2 *Composition example*

For more information about the Composite pattern, see Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]. The Composite View pattern is a variation of the Composite pattern.

In the Stock Trader Reference Implementation (Stock Trader RI) application, 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.

Separated 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 as the implementation. The **IModule** definition is part of the Composite Application Library, and each module implements the **IModule** interface. For an example of implementing a module in the Stock Trader RI, see the file NewsModule.cs.
- Plug-In. This pattern allows the concrete implementation of a class to be determined at run time to avoid requiring recompilation due to which concrete implementation is used or due to changes in the concrete implementation. In the Composite Application Library, this is handled through the **DirectoryLookupModuleEnumerator**, **ConfigurationModuleEnumerator**, and the **ModuleLoader**. For examples of plug-ins, see the files DirectoryLookupModuleEnumerator.cs, ConfigurationModuleEnumerator.cs, and ModuleLoader.cs in the Composite Application Library.

Inversion of Control

Frequently, the Inversion of Control (IoC) pattern is used to enable extensibility in a class or library. 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.

A form of the IoC pattern, Dependency Injection (DI), is used throughout the Stock Trader RI and the Composite Application Library. During a class's construction phase, DI provides any dependent classes to the class. 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. The Stock Trader RI uses Unity, a DI container. However, the Composite Application Library itself is not tied to a specific DI container; you are free to choose whichever DI container you want, but you must provide an adapter for it to the **IContainerFacade** interface. To see an example of using the container in the Stock Trader RI, see the NewsModule.cs file. For more details about IoC and DI patterns, see "Inversion of Control Pattern" and "Dependency Injection Pattern" later in this chapter.

Another form of IoC demonstrated in the Stock Trader RI is the Template Method pattern. 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 Composite Application Library, this is shown in the **UnityContainerAdapter** class. For more information about the Template Method pattern, see Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]. To see an example of using the Template pattern in the Stock Trader RI, see the file StockTraderRIBootstrapper.cs.

Service Locator

The Service Locator pattern is a form of the Inversion of Control pattern (as described earlier in this section). 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. For more information about the Service Locator pattern, see "Service Locator Pattern" later in this chapter. In the Composite Application Library, this can be seen when **ModuleLoader** service resolves individual **IModules**.

For an example of using the **UnityContainer** to locate a service in the Stock Trader RI, see the file NewsModule.cs.

Command

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 Composite Application Library provides a **CompositeCommand** that allows combining of multiple **ICommand** items and a **DelegateCommand** that allows a presenter or model 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 StockTraderRICommands.cs and OrderDetailsPresentationModel.cs.

For more details about the Command pattern, see Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1].

Adapter

The Adapter pattern, as the name implies, adapts the interface of one class to match the interface expected by another class. For more details, see Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]. In the Composite Application Library, the Adapter pattern is used to adapt regions to the WPF **ItemsControl**, **ContentControl**, and **Selector**. To see the Adapters pattern applied, see the file ItemsControlRegionAdatper.cs in the Composite Application Library.

Event Aggregator

The Event Aggregator pattern channels events from multiple objects through a single object to simplify registration for clients. For more information about this pattern, see "Event Aggregator" on Martin Fowler's Web site. In the Composite Application 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 the **EventAggregator** and the **CompositeWpfEvent** in the Composite Application Library.

To see the usage of the EventAggregator in the Stock Trader RI, see the file WatchListPresentationModel.cs.

Separated Presentation

Separated Presentation patterns are a category of patterns that focus on keeping the logic for the presentation separate from the visual representation. Primarily, this is done to allow testing of your presentation logic without the need to involve a visual representation. There are a number of specific implementations of these patterns, such as Model-View-Controller (MVC), Model-View-Presenter (MVP) variants, and Model-View-ViewModel (MVVM). For more information about MVC and MVP variants, see "GUI Architectures" on Martin Fowler's Web site. For more information about MVVM, see "Introduction to Model/View/ViewModel pattern for building WPF apps" on MSDN. This guidance includes a description of the Supervising Controller and Presentation Model patterns. For more information, see "Supervising Controller Pattern" and "Presentation Model Pattern" later in this chapter.

The Composite Application Library itself is intended to be neutral with respect to choice of Separated Presentation pattern. You can be successful with any of the patterns, although considering WPF's highly binding-oriented nature, patterns that support data-binding lend themselves better to WPF applications. The Stock Trader RI demonstrates the use of the Presentation Model pattern, an MVP variant that encapsulates the presentation logic and constructs a model to which the view can bind.

To see examples of the PresentationModel in the Stock Trader RI, see the files WatchListPresentationModel.cs, WatchListView.xaml, and WatchListService.cs.

Façade

The Façade pattern simplifies a more complex interface, or set of interfaces, to ease their use or to isolate access to those interfaces. For more details, see Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*. The Composite Application Library interfaces with 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 their own services that will work with the Composite Application Library. The **IContainerFacade** and **ILoggerFacade** are the interfaces the Composite Application Library expects when communicating with a container or logging service respectively.

More Information

The following are references and links to the patterns found in the Composite Application Library and Stock Trader RI:

- "Composite" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Adapter" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Façade" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Template Method" in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Plugin" on Martin Fowler's Web site
- "Dependency Injection Pattern" later in this chapter
- "Inversion of Control Pattern" later in this chapter
- "Service Locator Pattern" later in this chapter
- "Event Aggregator" on Martin Fowler's Web site
- "Separated Interface" on Martin Fowler's Web site
- "Separated Presentation Pattern" later in this chapter
- "Supervising Controller Pattern" later in this chapter
- "Presentation Model Pattern" later in this chapter
- "Model View Controller" and "Model-View-Presenter (MVP)" sections in "GUI Architectures" on Martin Fowler's Web site
- "Introduction to Model/View/ViewModel pattern for building WPF apps" on John Gossman's blog

1 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

Dependency Injection Pattern

Problem

You have classes that have dependencies on services or components whose concrete type is specified at design time. In this example, **ClassA** has dependencies on **ServiceA** and **ServiceB**. Figure 3.3 illustrates this.

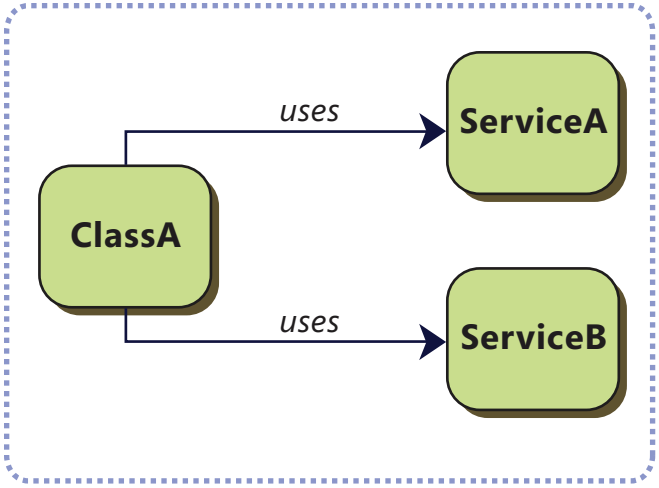


Figure 3.3 *ClassA has dependencies on ServiceA and ServiceB*

This situation has the following constraints:

- To replace or update the dependencies, you must change your classes' source code.
- The concrete implementation of the dependencies must be available at compile time.
- Your classes are difficult to test in isolation because they have a direct reference to their dependencies. This means that these dependencies cannot be replaced with stubs or mocks.
- Your classes contain repetitive code for creating, locating, and managing their dependencies.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to decouple your classes from their dependencies so that these dependencies can be replaced or updated with minimal changes or no changes to your classes' source code.
- You want to be able to write classes that depend on classes whose concrete implementation is not known at compile time.
- You want to be able to test your classes in isolation, without using the dependencies.
- You want to decouple your classes from being responsible for locating and managing the lifetime of dependencies.

Solution

Do not instantiate the dependencies explicitly in your class. Instead, declaratively express dependencies in your class definition. Use a **Builder** object to obtain valid instances of your object's dependencies and pass them to your object during the object's creation and/or initialization. Figure 3.4 illustrates this.

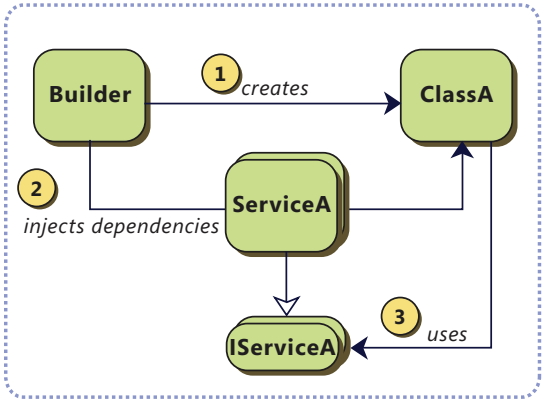


Figure 3.4 *Conceptual view of the Dependency Injection pattern*

Note: Typically, you express dependencies on interfaces instead of concrete classes. This enables easy replacement of the dependency concrete implementation without modifying your classes' source code.

The following are the two main forms of dependency injection:

- Constructor injection
- Setter injection

In *constructor injection*, you use parameters of the object's constructor method to express dependencies and to have the builder inject it with its dependencies. In *setter injection*, the dependencies are expressed through setter properties that the builder uses to pass the dependencies to it during object initialization.

Implementation Using the Unity Application Block

The Dependency Injection pattern can be implemented in several ways. The Unity Application Block (Unity) provides a container that can be used for dependency injection. For more information about the Unity Application Block, see "Unity Application Block" on MSDN.

Example

The **NewsReaderPresenter** class of the Stock Trader RI (located at StockTraderRI.Modules.News\Article\NewsReaderPresenter.cs) uses constructor dependency injection to obtain a valid instance of a view that implements the **INewsReaderView** interface. The class definition is shown in the following code.

C# NewsReaderPresenter.cs

```
public class NewsReaderPresenter : INewsReaderPresenter
{
    private INewsReaderView readerView;

    public NewsReaderPresenter(INewsReaderView view)
    {
        this.readerView = view;
    }

    public void SetNewsArticle(NewsArticle article)
    {
        readerView.Model = article;
    }

    public void Show()
    {
        readerView.ShowView();
    }
}
```

Because the **NewsReaderPresenter** class uses dependency injection to obtain its dependencies, its dependencies can be replaced with mock implementations when testing. The following test methods, taken from the fixture file StockTraderRI.Modules.News.Tests\NewsView\NewsReaderPresenterFixture.cs, show how the **NewsReaderPresenter** class can be tested in isolation using mock implementations to replace the classes' dependencies and verify behavior.

C# NewsReaderPresenterFixture.cs

```
[TestMethod]
public void ShowInformsViewToShow()
{
    var view = new MockNewsReaderView();
    var presenter = new NewsReaderPresenter(view);

    presenter.Show();

    Assert.IsTrue(view.ShowViewWasCalled);
}

[TestMethod]
public void SetNewsArticlesSetsViewModel()
{
    var view = new MockNewsReaderView();
    var presenter = new NewsReaderPresenter(view);

    NewsArticle article = new NewsArticle() { Title = "My Title", Body = "My Body"};
    presenter.SetNewsArticle(article);

    Assert.AreSame(article,view.Model);
}
```

Liabilities

The dependency injection pattern has the following liabilities:

- There are more solution elements to manage.
- You have to ensure that, before initializing an object, the dependency injection infrastructure can resolve the dependencies that are required by the object.
- There is added complexity to the source code; therefore, it is harder to understand.

Related Patterns

The following patterns are related to the Dependency Injection pattern:

- Inversion of Control. The Dependency Injection pattern is a specialized version of the Inversion of Control pattern where the concern being inverted is the process of obtaining the needed dependency.
- Service Locator. The Service Locator pattern solves the same problems that the Dependency Injection pattern solves, but it uses a different approach.

More Information

For more information about the Dependency Injection pattern, see the following:

- "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site
- "Design Patterns: Dependency Injection" by Griffin Caprio on MSDN

Inversion of Control Pattern

Problem

You have classes that have dependencies on services or components whose concrete type is specified at design time. In this example, **ClassA** has dependencies on **ServiceA** and **ServiceB**. Figure 3.5 illustrates this.

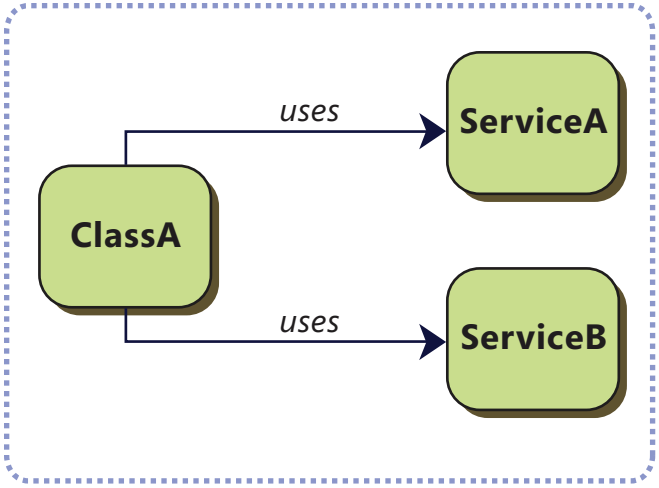


Figure 3.5 *ClassA has dependencies on ServiceA and ServiceB*

This situation has the following problems:

- To replace or update the dependencies, you need to change your classes' source code.
- The concrete implementations of the dependencies have to be available at compile time.
- Your classes are difficult to test in isolation because they have direct references to dependencies. This means that these dependencies cannot be replaced with stubs or mocks.
- Your classes contain repetitive code for creating, locating, and managing their dependencies.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to decouple your classes from their dependencies so that the dependencies can be replaced or updated with minimal or no changes to your classes' source code.
- You want to write classes that depend on classes whose concrete implementations are not known at compile time.
- You want to test your classes in isolation, without using the dependencies.
- You want to decouple your classes from being responsible for locating and managing the lifetime of dependencies.

Solution

Delegate the function of selecting a concrete implementation type for the classes' dependencies to an external component or source.

Implementation Details

The Inversion of Control pattern can be implemented in several ways. The Dependency Injection pattern and the Service Locator pattern are specialized versions of this pattern that delineate different implementations. Figure 3.6 illustrates the conceptual view of both patterns.

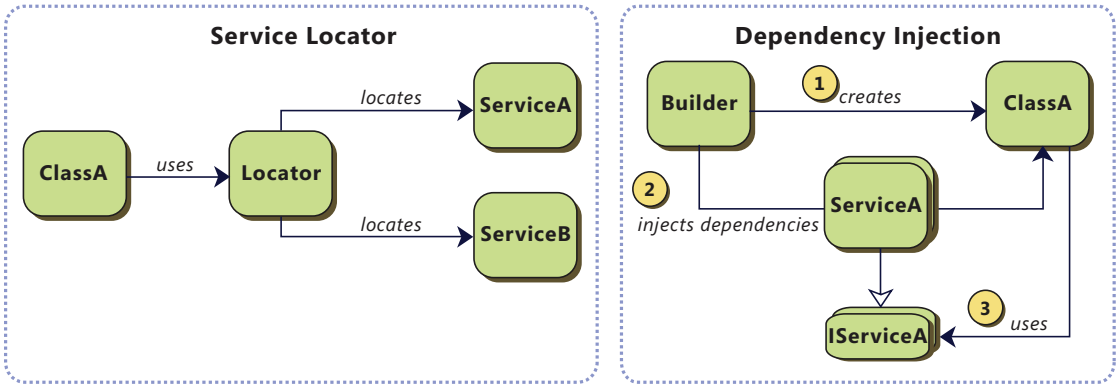


Figure 3.6 *Conceptual view of the Service Locator and Dependency Injection patterns*

For more information about these patterns, see "Dependency Injection Pattern" and "Service Locator Pattern" later in this chapter.

Examples

The following are example implementations of the Inversion of Control pattern:

- In the Configuration Modularity QuickStarts, the class ModuleA defined in the ModuleA project uses dependency injection to obtain a reference to the region manager service, as shown in the following code.

C# ModuleA.cs

```
public class ModuleA : IModule
{
    private readonly IRegionManager _regionManager;

    public ModuleA(IRegionManager regionManager)
    {
        _regionManager = regionManager;
    }

    ...
}
```

Because the **ModuleA** class is instantiated by a container and an instance of the region manager service is registered with the container, the **ModuleA** class receives a valid instance of the region manager service when it is constructed. Note that a mock instance of the region manager service can be supplied when testing the **ModuleA** class by passing the mock instance in the constructor's parameter.

- The following code, extracted from the NewsModule class of the Stock Trader RI (this class is located at StockTraderRI.Modules.News\NewsModule.cs), shows how an instance that implements the INewsController interface is obtained using the service locator pattern. The variable _container holds an instance to a container that has logic to locate a valid instance of the requested type.

C# NewsModule.cs

```
public void Initialize()
{
    RegisterViewsAndServices();
    INewsController controller = _container.Resolve<INewsController>();
    controller.Run();
}
```

Note that for testing purposes, you could configure the container to return a mock instance that implements the **INewsController** interface instead of the real implementation. This enables you to test the **NewsModule** class in isolation. The following code, extracted from the **NewsModuleFixture** test class (located in StockTraderRI.Modules.News.Tests\NewsModuleFixture.cs), shows how the **NewsModule** class can be tested in isolation using a mock instance for the **INewsController** interface.

C# NewsModuleFixture.cs

```
[TestMethod]
public void InitCallsRunOnNewsController()
{
    MockUnityResolver container = new MockUnityResolver();
    var controller = new MockNewsController();
    container.Bag.Add(typeof(INewsController), controller);
    var newsModule = new NewsModule(container);

    newsModule.Initialize();

    Assert.IsTrue(controller.RunCalled);
}
```

Liabilities

The Inversion of Control pattern has the following liabilities:

- You need to implement a mechanism that provides the dependencies that are required by the object that is being initialized.
- There is added complexity to the source code, which makes it harder to understand.

Related Patterns

The following patterns are related to the Inversion of Control pattern:

- Dependency Injection. The Dependency Injection pattern is a specialization of the Inversion of Control pattern. The Dependency Injection pattern uses a builder object to initialize objects and provide the required dependencies to the object.
- Service Locator. The Service Locator pattern is a specialization of the Inversion of Control pattern. The Service Locator pattern introduces a locator object that objects use to resolve dependencies.

More Information

For more information about Inversion of Control patterns, see the following:

- "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site.

Service Locator Pattern

Problem

You have classes that have dependencies on services whose concrete type is specified at design time. In this example, **ClassA** has dependencies on **ServiceA** and **ServiceB**. Figure 3.7 illustrates this.

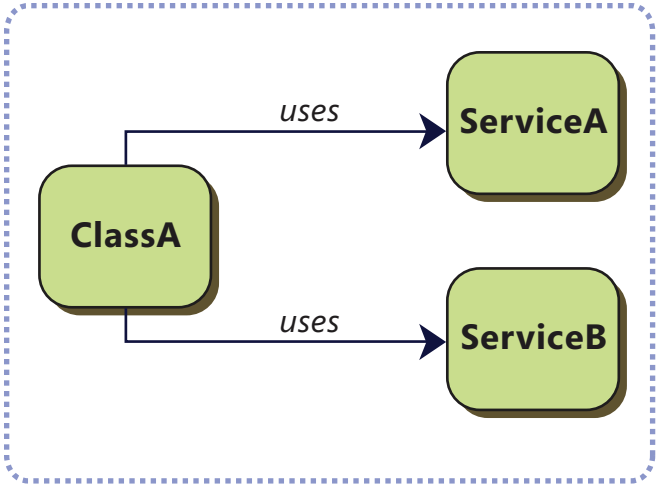


Figure 3.7 *ClassA has dependencies on ServiceA and ServiceB*

This situation has the following constraints:

- To replace or update the dependencies, you must change your classes' source code.
- The concrete implementation of the dependencies must be available at compile time.
- Your classes are difficult to test in isolation because they have a direct reference to their dependencies. This means that these dependencies cannot be replaced with stubs or mocks.
- Your classes contain repetitive code for creating, locating, and managing their dependencies.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to decouple your classes from their dependencies so that these dependencies can be replaced or updated with minimal or no changes to your classes' source code.
- You want to write classes that depend on classes whose concrete implementation is not known at compile time.

- You want to be able to test your classes in isolation, without using the dependencies.
- You want to decouple your classes from being responsible for locating and managing the lifetime of dependencies.

Solution

Create a service locator that contains references to the services and that encapsulates the logic to locate them. In your classes, use the service locator to obtain service instances. Figure 3.8 illustrates this.

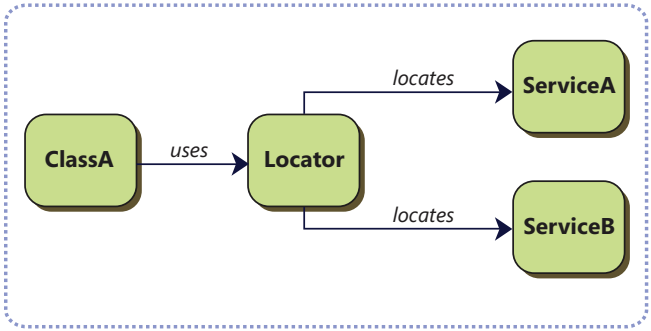


Figure 3.8 *ClassA uses the service locator to get an instance of ServiceA*

The service locator does not instantiate the services. It provides a way to register services and it holds references to the services. After the service is registered, the service locator can find the service.

Note: The service locator should provide a way to locate a service without specifying the concrete type. For example, it could use a string key or a service interface type. This enables easy replacement of the dependency concrete implementation without modifying your classes' source code.

Implementation with the Unity Application Block

The Service Locator pattern can be implemented in several ways. The Unity Application Block provides a container that can be used as a service locator. For more information about the Unity Application Block, see Unity Application Block.

Example

The following code, extracted from the **NewsModule** class of the Stock Trader RI (this class is located at StockTraderRI.Modules.News\NewsModule.cs), shows how an instance that implements the **INewsController** interface is obtained using the service locator pattern. The variable **_container** holds an instance to a Unity container that has logic to locate a valid instance of the requested type.

C# NewsModule.cs

```
public void Initialize()
{
    RegisterViewsAndServices();
    INewsController controller = _container.Resolve<INewsController>();
    controller.Run();
}
```

Note that for testing purposes, you could configure the container to return a mock instance that implements the **INewsController** interface instead of the real implementation. This enables you to test the **NewsModule** class in isolation. The following code, extracted from the **NewsModuleFixture** test class (located at StockTraderRI.Modules.News.Tests\NewsModuleFixture.cs), shows how the **NewsModule** class can be tested in isolation using a mock instance for the **INewsController** interface.

C# NewsModuleFixture.cs

```
[TestMethod]
public void InitCallsRunOnNewsController()
{
    MockUnityResolver container = new MockUnityResolver();
    var controller = new MockNewsController();
    container.Bag.Add(typeof(INewsController), controller);
    var newsModule = new NewsModule(container);

    newsModule.Initialize();

    Assert.IsTrue(controller.RunCalled);
}
```

Liabilities

The Service Locator pattern has the following liabilities:

- There are more solution elements to manage.
- You have to write additional code to add service references to the locator before your objects use it.
- Your classes have an extra dependency on the service locator.
- The source code has added complexity; this makes the source code more difficult to understand.

Related Patterns

The following patterns are related to the Service Locator pattern:

- Dependency Injection. The Dependency Injection pattern solves the same problems that the Service Locator pattern solves, but it uses a different approach.
- Inversion of Control. The Service Locator pattern is a specialized version of the Inversion of Control pattern where the concern being inverted is the process of obtaining the needed dependency.

More Information

For more information about the Service Locator pattern, see the following:

- "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site
- "Service Locator" on MSDN

Separated Presentation Pattern

Problem

A view in a composite application contains controls that display application domain data. A user can modify the data and submit the changes. The view retrieves the domain data, handles user events, alters other controls on the view in response to the events, and then it submits the changed domain data. Including the code that performs these functions in the view makes the class complex, difficult to maintain, and hard to test. In addition, it is difficult to share code between views that require the same behavior.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to maximize the code that can be tested with automation. (Views are hard to test.)
- You want to share code between views that require the same behavior.
- You want to separate business logic from user interface (UI) logic to make the code easier to understand and maintain.

Solution

Separate the presentation logic from the business logic into different artifacts. The Separated Presentation pattern can be implemented in multiple ways; the following patterns provide prescriptive approaches to implement the pattern:

- Supervising Controller. This pattern, a variant of the Model-View-Presenter pattern, separates the responsibilities for the visual display and the event handling behavior into different classes named, respectively, the view and the presenter, and permits using data binding to enable direct communication between the view and the model.
- Presentation Model. This pattern, a variant of the Model-View-Presenter pattern, supplements a façade on the model with UI-specific state and behavior that is easy to consume from the view.

Figure 3.9 shows the logical view of the Supervising Controller and the Presentation Model patterns.

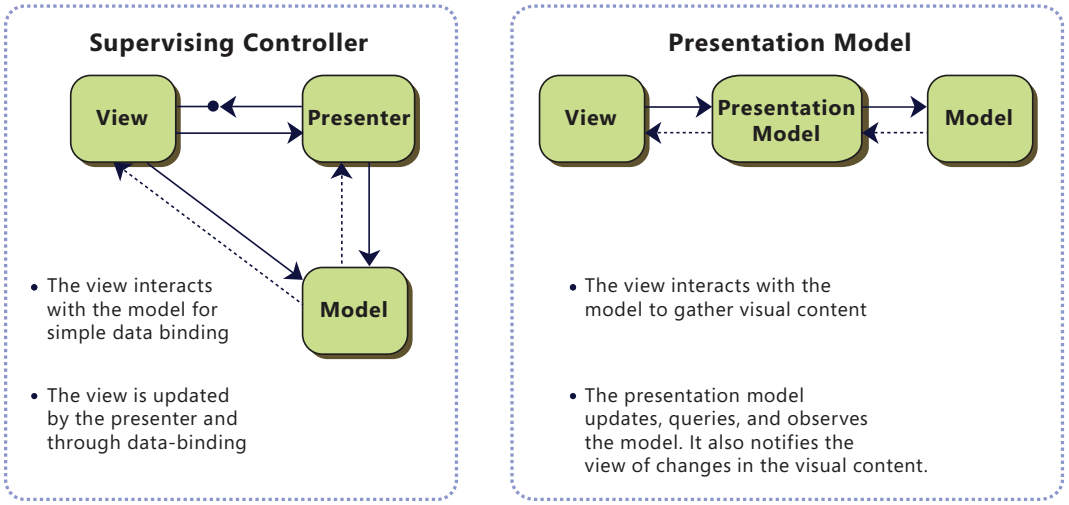


Figure 3.9 *Logical view of the Supervising Controller and Presentation Model patterns*

Liabilities

The Supervising Controller and the Presentation Model patterns have the following liabilities:

- There are more solution elements to manage.
- You need a way to create and connect the different artifacts that collaborate to provide the visual content.

Related Patterns

The following patterns are related to the Separated Presentation pattern:

- Supervising Controller. This pattern solves the same problems that the Presentation Model pattern solves. This pattern separates the responsibilities into two classes, named respectively, the view and the presenter, and the view interacts with the model for simple data binding.
- Presentation Model. This pattern is a specialization of the Separated Presentation pattern. This pattern separates the responsibilities for the visual display and the user interface state and behavior into different classes named, respectively, the view and the presentation model.

More Information

For more information about Separated Presentation patterns, see the following:

- "Separated Presentation" on Martin Fowler's Web site

Supervising Controller Pattern

Problem

A view in a composite application contains controls that display application domain data. A user can modify the data and submit the changes. The view retrieves the domain data, handles user events, alters other controls on the view in response to the events, and submits the changed domain data. Including the code that performs these functions in the view makes the class complex, difficult to maintain, and hard to test. In addition, it is difficult to share code between views that require the same behavior.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to maximize the code that can be tested with automation. (Views are hard to test.)
- You want to share code between views that require the same behavior.
- You want to separate business logic from user interface (UI) logic to make the code easier to understand and maintain.

Solution

Separate the responsibilities for the visual display and the event handling behavior into different classes named, respectively, the view and the presenter. The view class manages the controls on the user interface and forwards user events to a presenter class. The presenter contains the logic to respond to the events, update the model (business logic and data of the application) and, in turn, manipulate the state of the view.

To facilitate testing the presenter, the presenter has a reference to the view interface instead of to the concrete implementation of the view. By doing this, you can easily replace the real view with a mock implementation to run tests.

View Updates and Interaction with the Model

When the model is updated, the view also has to be updated to reflect the changes. View updates can be handled in several ways. With the Supervising Controller pattern, the view directly interacts with the model to perform simple data binding that can be declaratively defined, without presenter intervention. The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required. Examples of complex UI logic might include changing the color of a control or dynamically hiding/showing controls. Figure 3.10 illustrates the logical view of the Supervising Controller pattern.

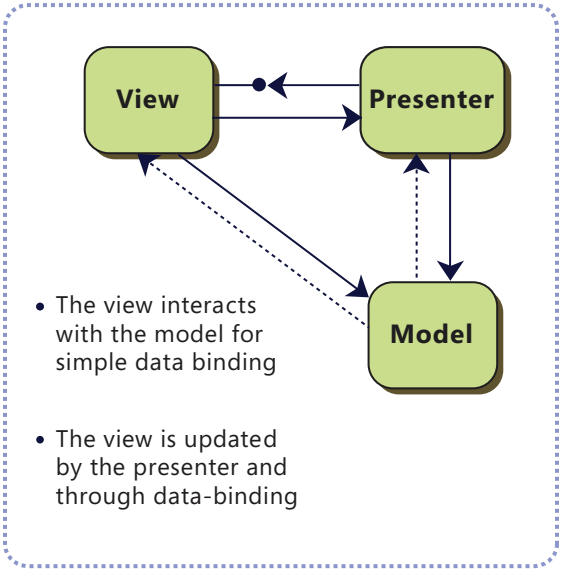


Figure 3.10 *Supervising Controller logical view*

By using the Supervising Controller pattern in a composite WPF application, developers can use the data-binding capabilities provided by WPF. With data binding, elements that are bound to application data automatically reflect changes when the data changes its value. Also, data binding can mean that if an outer representation of the data in an element changes, the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a **TextBox** element, the underlying data value is automatically updated to reflect that change.

A typical use of data binding is to place server or local configuration data into forms or other UI controls. In WPF, this concept is expanded to include the binding of a broad range of properties to a variety of data sources. In WPF, dependency properties of elements can be bound to common language runtime (CLR) objects (including ADO.NET objects or objects associated with Web services and Web properties) and XML data.

Note: For more information about data-binding in WPF, see "Data Binding" on MSDN.

Example

To see an example implementation of the Supervising Controller pattern, see the files TrendLinePresenter.cs, TrendLineView.xaml, and TrendLineView.xaml.cs in the Stock Trader RI (these files are located in the TrendLine folder of the StockTraderRI.Modules.Market project).

Liabilities

The Supervising Controller pattern has the following liabilities

- There are more solution elements to manage.
- You need a way to create and connect views and presenters.
- The model is not aware of the presenter. Therefore, if the model is changed by any component other than the presenter, the presenter must be notified. Typically, notification is implemented with the Observer pattern. For more information about the Observer pattern, see "Exploring the Observer Design Pattern" on MSDN.

Related Patterns

The following patterns are related to the Separated Presentation pattern:

- Separated Presentation. This pattern is a specialization of the Separated Presentation pattern. Separated Presentation patterns are a category of patterns that focus on keeping the logic for the presentation separate from the visual representation.
- Presentation Model. This pattern solves the same problems that the Supervising Controller pattern solves; the main difference is that the Presentation Model pattern separates responsibilities into different classes named, respectively, the view and the presentation model.

More Information

For more information about the Supervising Controller pattern, see the following:

- "Supervising Controller" on Martin Fowler's Web site
- "Model-View-Presenter Pattern" on MSDN

Presentation Model Pattern

Problem

A view in a composite application contains controls that display application domain data. A user can modify the data and submit the changes. The view retrieves the domain data, handles user events, alters other controls on the view in response to the events, and submits the changed domain data. Including the code that performs these functions in the view makes the class complex, difficult to maintain, and hard to test. In addition, it is difficult to share code between views that require the same behavior.

Forces

Any of the following conditions justifies using the solution described in this pattern:

- You want to maximize the code that can be tested with automation. (Views are hard to test.)
- You want to share code between views that require the same behavior.
- You want to separate business logic from user interface (UI) logic to make the code easier to understand and maintain.

Solution

Separate the responsibilities for the visual display and the user interface state and behavior into different classes named, respectively, the view and the presentation model. The view class manages the controls on the user interface and the presentation model class acts as a façade on the model with UI-specific state and behavior, by encapsulating the access to the model and providing a public interface that is easy to consume from the view (for example, using data binding). Figure 3.11 provides a logical view of the Presentation Model pattern.

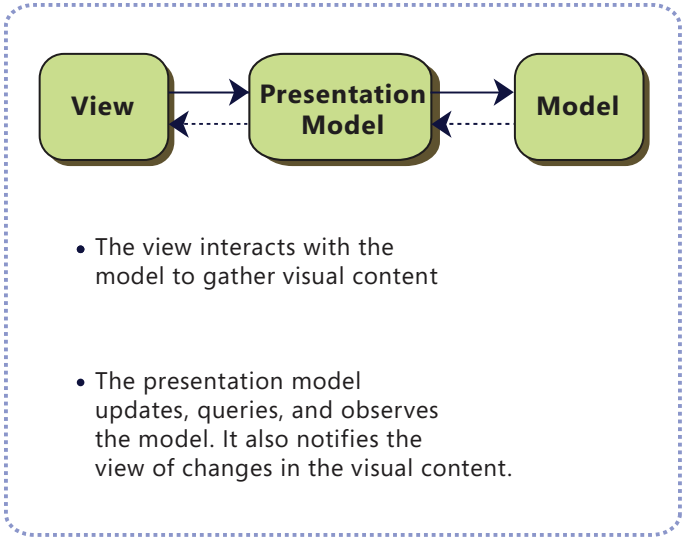


Figure 3.11 *Presentation Model pattern logical view*

By using the Presentation Model pattern in a composite WPF application, developers can use the data-binding capabilities provided by WPF. With data binding, elements that are bound to application data automatically reflect changes when the data changes its value. Also, data binding can mean that if an outer representation of the data in an element changes, the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a **TextBox** element, the underlying data value is automatically updated to reflect that change.

A typical use of data binding is to place server or local configuration data into forms or other UI controls. In WPF, this concept is expanded to include the binding of a broad range of properties to a variety of data sources. In WPF, dependency properties of elements can be bound to common language runtime (CLR) objects (including ADO.NET objects or objects associated with Web services and Web properties) and XML data.

Note: For more information about data binding in WPF, see "Data Binding" on MSDN.

When you implement the Presentation Model pattern in a WPF application, you can use data templates for your presentation model class and have WPF automatically apply the template for it to render the visual content. By doing this, developers keep the view code simple, and UI designers can focus on building the data templates. For more information about data templates, see "Data Templating Overview" on MSDN.

Note: A variant of the Presentation Model pattern named Model-View-ViewModel is commonly used in WPF applications. For more information, see John Gossman's Tales from the Smart Client blog "Introduction to Model/View/ViewModel pattern for building WPF apps" on MSDN.

Example

To see an example implementation of the Presentation Model pattern, see the files WatchListPresentationModel.cs, WatchListView.xaml, and WatchListService.cs in the Stock Trader RI (these files are located in the WatchList folder of the StockTraderRI.Modules.Watch project).

Liabilities

The Presentation Model pattern has the following liabilities:

- There are more solution elements to manage.
- You need a way to synchronize the view with the presentation model. Typically, this is done through data binding.
- The model is not aware of the presentation model. Therefore, if the model is changed by any component other than the presentation model, the presentation model must be notified. Typically, notification is implemented with the Observer pattern. For more information about the Observer pattern, see "Exploring the Observer Design Pattern" on MSDN.

Related Patterns

The following patterns are related to the Separated Presentation pattern:

- Separated Presentation. This pattern is a specialization of the Separated Presentation pattern. Separated Presentation patterns are a category of patterns that focus on keeping the logic for the presentation separate from the visual representation.
- Supervising Controller. This pattern solves the same problems that the Presentation Model pattern solves; the main difference is that the Supervising Controller pattern separates responsibilities into different classes named, respectively, the view and the presenter, and the view interacts with the model for simple data binding.

More Information

For more information about the Presentation Model pattern, see the following:

- "Presentation Model" on Martin Fowler's Web site.

4

Composite Application Library

The Composite Application Library helps architects and developers create composite Windows Presentation Foundation (WPF) applications. Composite WPF applications are composed of discrete, functionally complete, pieces that work together to create a single, integrated user interface. The Composite Application Library accelerates the development of composite applications using proven design patterns to help you build these types of applications.

The Composite Application Library is designed to address requests from architects and developers who create WPF client applications and need to accomplish the following:

- Build clients composed of independent, yet cooperating, modules.
- 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.
- Use an architectural infrastructure to produce a consistent and high quality integrated application.

Your composite WPF application will use the Composite Application Library, and it may use the Unity Extensions for Composite Application Library and the Unity Application Block. These are built on the .NET Framework 3.5, as illustrated in Figure 4.1 on the next page.

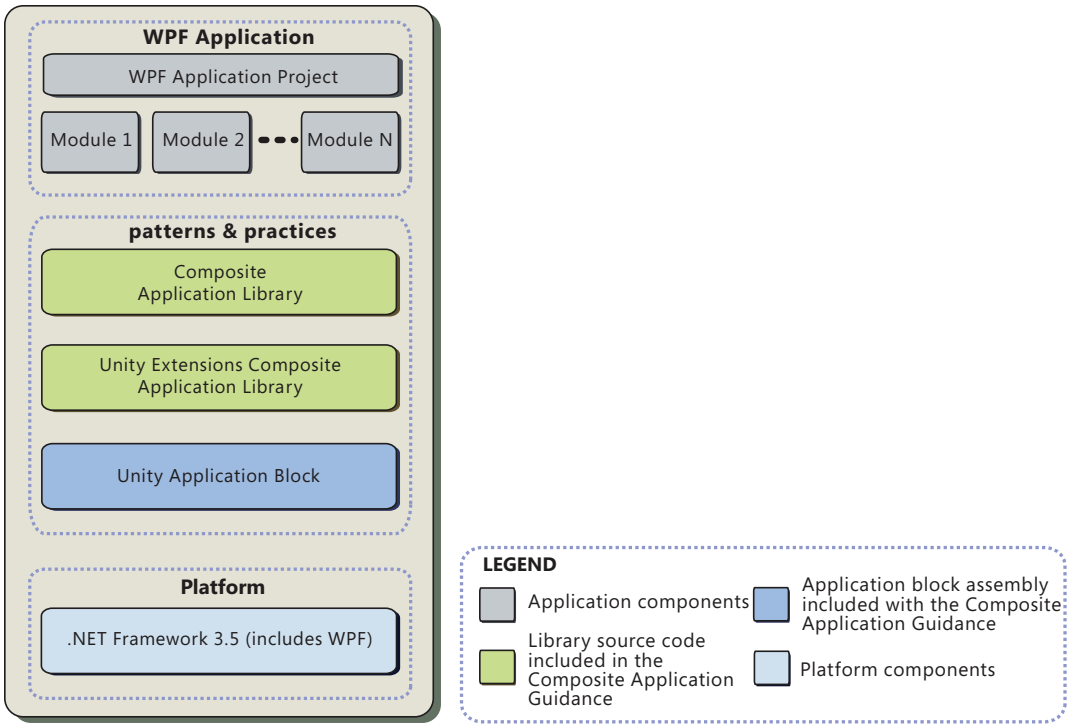


Figure 4.1 *Composite application package*

The Composite Application Library addresses a number of common requirements for building composite applications. These compositional needs are described in the "Overview of the Composite Application Guidance for WPF" in Chapter 1, "Introduction." As a whole, the Composite Application Library accelerates development by providing the services and components to address these needs.

System Requirements

This guidance was designed to run on the Microsoft Windows® Vista, Windows XP Professional, or Windows Server 2003 operating system. Applications built using this guidance will require the .NET Framework 3.5 to run.

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

- Microsoft Visual Studio 2008
- Microsoft .NET Framework 3.5 (the .NET Framework 3.5 includes WPF)

Composite Application Library Baseline Architecture

The architecture the Composite Application Library seeks to address primarily consists of a shell application that defines regions for hosting content with views and services offered by multiple modules—possibly dynamically loaded. Underlying the application, and the Composite Application Library, is a service layer to provide access to the application services based on the Composite Application Library, as illustrated in Figure 4.2.

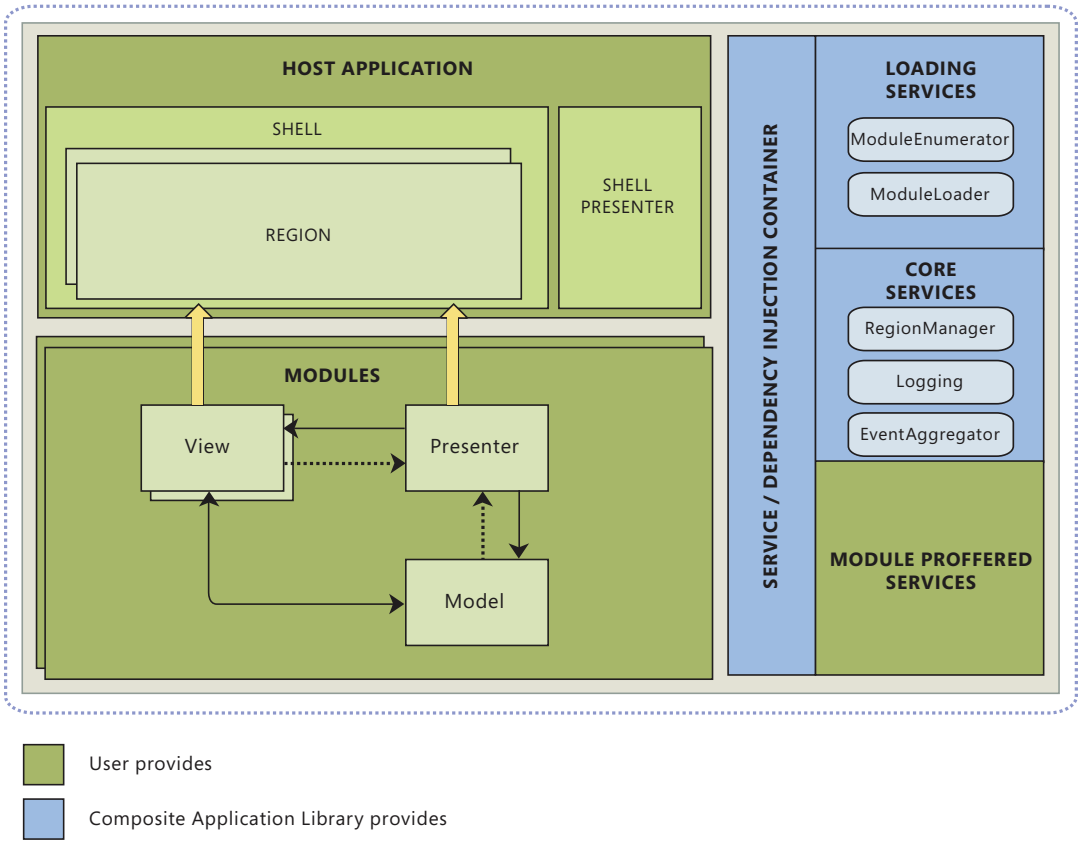


Figure 4.2 *A composite application architecture with the Composite Application Library*

The architectural pieces of a composite application are the following:

- **Shel**l. This is the top-level window to host different user interface components. The shell itself defines the layout structure, but it is typically unaware of the exact contents it will contain. It typically has minimal capability, so most of the application's functionality and content is provided by modules.
- **Shell presenter.** Any logic for the shell presentation is handled by the shell presenter. This follows the separated presentation pattern and helps separate the display of content from the user interface logic. This separation improves testability and maintainability.
- **Regions.** These are placeholders for content and host visual elements in the shell. These can be located by other components through the **RegionManager** to add content to those regions. Regions can also be used in module views to create discoverable content placeholders.
- **Modules.** These are separate sets of views and services, frequently logically related, that can be independently developed, tested, and optionally deployed. In many situations, these can be developed and maintained by separate teams. In a composite application, modules must be discovered and loaded; in the Composite Application Library this process is known as module enumeration and module loading. The following describes module enumeration and module loading:
  - **Module enumeration.** This is the process of locating individual modules for loading. This location can be done statically or dynamically through a configuration file or by examining a directory. This includes the following types of enumeration:
    - **Static enumeration.** This allows the shell application to specify the set of modules in code and is statically referenced from the shell application.
    - **Configuration-base enumeration.** This specifies the modules to load in a configuration file.
    - **Directory sweep enumeration.** This examines a folder for the modules to load.
  - **Module loading.** This feature allows a component to specify, in code, when a module should be loaded. Modules discovered during enumeration may be immediately loaded or loaded on-demand.
  - **Initialization.** The **ModuleLoader** initializes each module when it is first loaded.

- **Views.** Views are responsible for displaying content on the screen. In a composite application, views are frequently the element of composition by the shell or other views. The following are considerations for use when developing views in WPF:
  - The user interface should be testable and implement appropriate design patterns for separations of concerns. Typically, this involves some type of binding-oriented design pattern, such as Supervising Controller or Presentation Model.
  - WPF data binding should be used wherever possible.
  - WPF commands should be used for binding user interface actions.
- **Communication.** Different components in the application may need to communicate with one another whether they reside in the same module or different modules. This needs to happen without the modules requiring hard dependencies on one another. The Composite Application Library provides mechanisms to do this with the **CompositeCommand** and **EventAggregator**. Communication strategies may need to consider thread-safety issues. The following describes the **CompositeCommand** and **EventAggregator**:
  - **CompositeCommand.** Frequently, when compositing views, commands those views support must also be composited. The composite command is a strategy to combine the execution of commands. This allows the command invoker to interact with a single command that affects multiple commands.
  - **EventAggregator.** In views that need to send an event to other views or components and do not require a response, use the **EventAggregator**. Multiple components can publish an event, and multiple subscribers can receive the event.
- **Services.** The application and modules expose services for their own and shared use. These are exposed through a service container that locates and, often, constructs the services. By default, the Composite Application Library uses the Unity container for this service location.

When to Use the Composite Application Library

The Composite Application Library is most useful when building brand-new composite WPF applications. The Composite Application Library provides core services for building composite views and loading modules, and it helps solve challenges in communication across decoupled components.

Discrete services offered by the Composite Application Library also help when seeking to upgrade an existing WPF application to a composite WPF application.

A New Application Based on the Composite Application Library

Depending on application complexity, a composite WPF application can consist of a shell project with a number of module projects. The activity diagram in Figure 4.3 illustrates activities needed to develop a composite WPF application using the Composite Application Library.

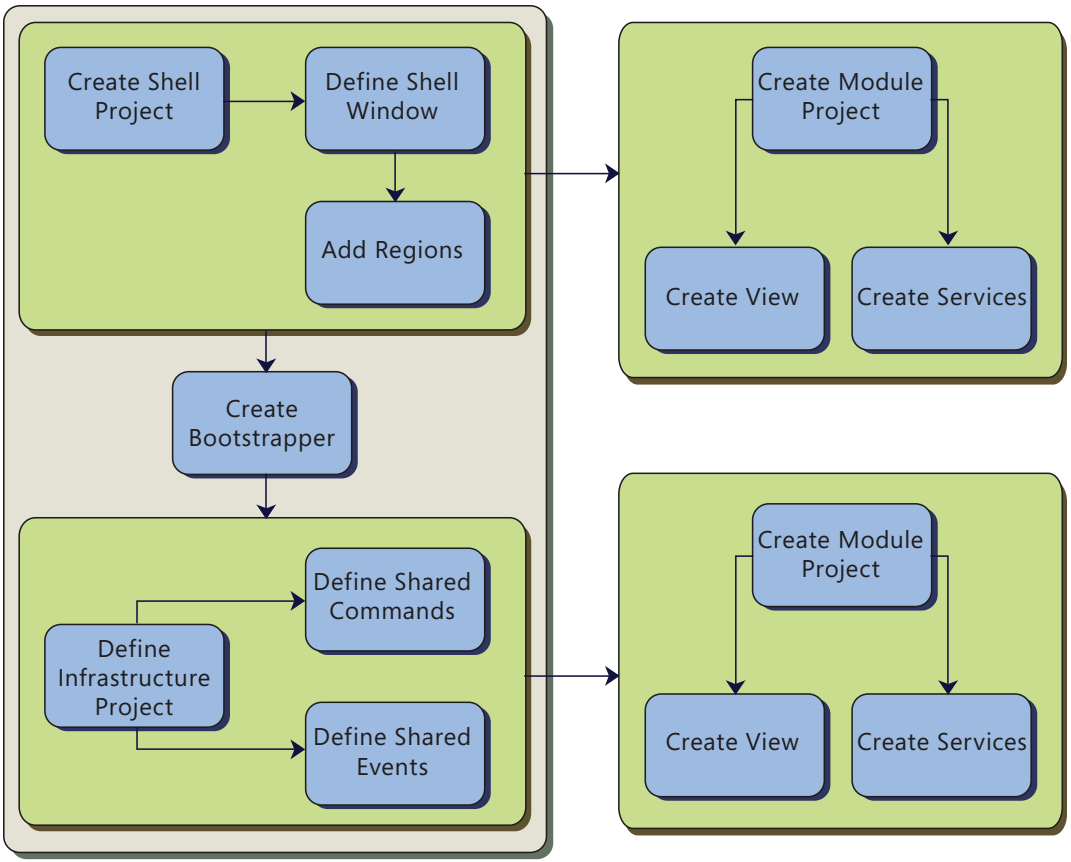


Figure 4.3 *Activities for creating a WPF composite application*

The following are the core activities needed when starting a new composite WPF application:

- Define the shell.
- Create the bootstrapper.
- Create a module.
- Add a module view to the shell.

Define the Shell

The application shell provides the basic layout for the application. This layout is defined using regions that modules can use to inject views. Views, like shells, can use regions to define discoverable areas that content can be injected into, as illustrated in Figure 4.4. Shells typically set the appearance for the entire application and contain the styles that are used throughout the application.

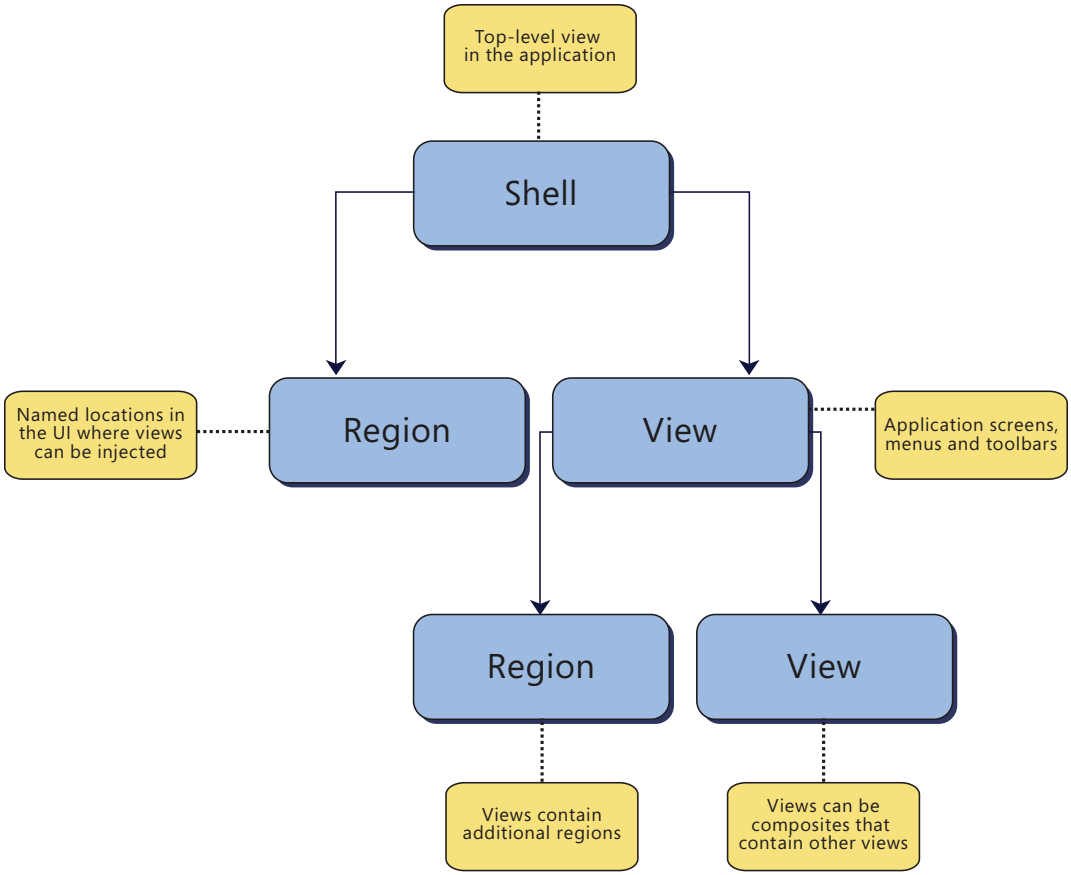


Figure 4.4 *Shells, views, and regions*

Create the Bootstrapper

The bootstrapper is the glue that connects the application with the Composite Application Library services and the default Unity container. Each application creates an application specific to the bootstrapper inheriting from **UnityBootstrapper** and defines the enumeration strategy for its modules, such as static or configuration-based, as illustrated in Figure 4.5 on the next page.

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** enables all the Composite Application Library services. These can be disabled or replaced in your application-specific bootstrapper.

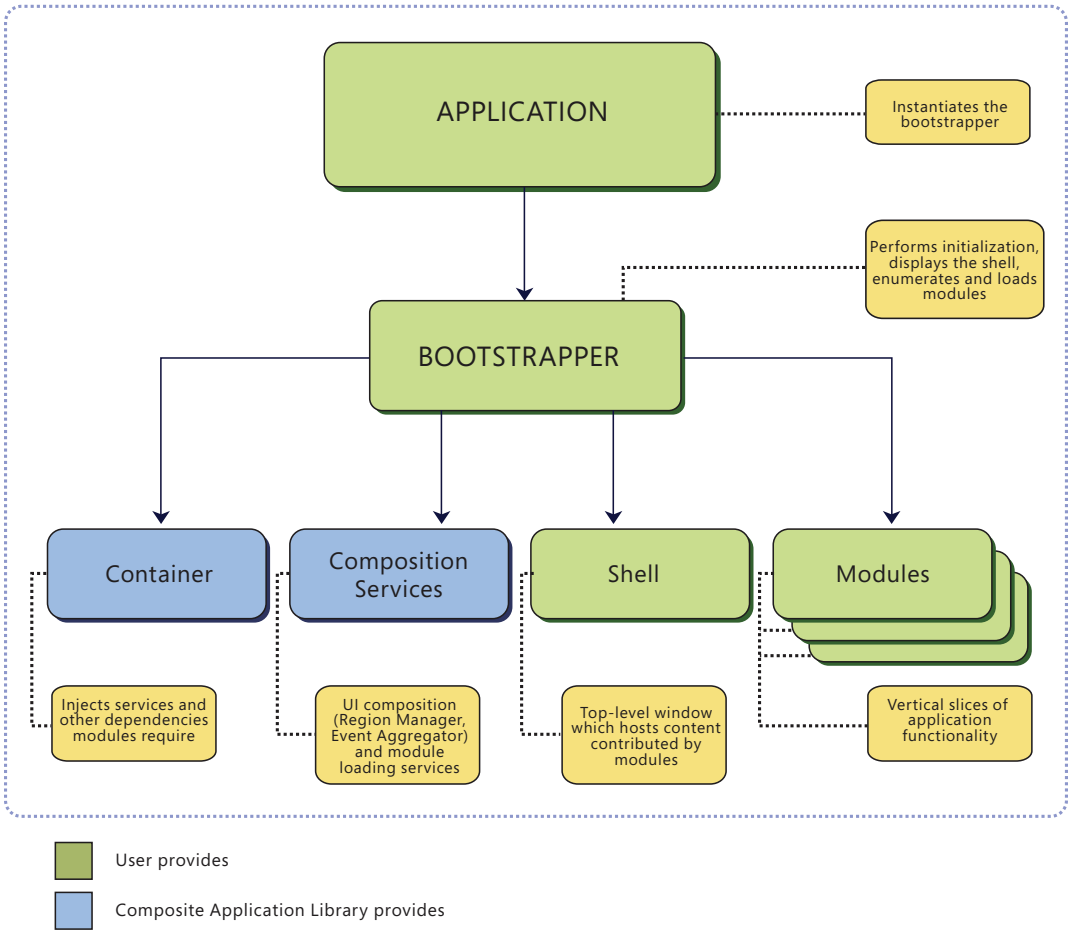


Figure 4.5 *Diagram demonstrating connecting to the Composite Application Library*

For more information about the bootstrapper, see the following sections:

- "Bootstrapper" in Chapter 6, "Technical Concepts"
- "Container and Services" in Chapter 6, "Technical Concepts"

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 loading approach, you may need to apply attributes to your module classes or define dependencies between your modules. For more information about modules, see "Module" in Chapter 6, "Technical Concepts."

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. The **RegionManager** is responsible for keeping track of regions throughout the application and is a core service initialized from the bootstrapper. For more information about shells and views, see the following sections:

- "Shell and View" in Chapter 6, "Technical Concepts"
- "UI Composition" in Chapter 2, "Design Concepts"

Goals and Benefits

This section provides an overview of the goals and benefits of using the Composite Application Library.

Architectural Goals

The Composite Application Library is designed to help architects and developers achieve the following objectives:

- Create a complex WPF application from modules that can be built, assembled, and, optionally, deployed by independent teams.
- Minimize cross-team dependencies and allow teams to specialize in different areas, such as 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.

Design Goals

The Composite Application Library is designed to support the following extensibility and developer productivity objectives:

- Different teams can create modules that can be independently deployed to a client computer.
- Developers can easily implement WPF user interfaces that use patterns to separate user interface design code from business logic, such as the Presentation Model pattern and the Model-View-Presenter with Supervising Controller pattern.
- Cooperating modules can contribute and use shared infrastructure components.
- The shell that hosts the modules can determine the appearance of shared module elements.
- Developers can use pieces of the Composite Application Library without using the entire library.
- The Composite Application Library can integrate into existing WPF applications.
- The Composite Application Library is extensible so you can customize its behavior for special or uncommon scenarios.

The Composite Application Library helps achieve these design objectives through the following strategies:

- It uses dependency injection techniques to simplify the code necessary to implement the Presentation Model and Model-View-Presenter patterns.
- It provides a module loading infrastructure and allows modules to perform startup operations. Modules can also register shared components for use by other modules or the shell application.
- It includes an extensive set of unit tests with the source code. Developers can modify the library and use the test to verify its functionality.
- It separates the interface and implementation for the library services.
- It provides ClickOnce deployment for independent modules.

Benefits

Developers can use the Composite Application Library to develop WPF applications that are composed of independent, but collaborating, modules. This allows developers or teams to concentrate on specific tasks. For example, developers of service components can focus on the business logic in the component; they do not have to be concerned with background issues, such as the appearance of the application.

Applications built with the Composite Application Library have a loosely coupled design that clearly separates user interface constructs (such as views, menu items, and toolbars), infrastructure components (such as logging, exception handling, authentication, or authorization), and business logic (such as the user interface logic, entities, and service agents of the specific application).

Modularity

The Composite Application Library promotes modularity; this means you can implement business logic, visual components, infrastructure components, presenter or controller components, and any other objects the application requires, in separate modules. Developers can easily create the UI and implement business logic independently of each other.

For more details about modularity, see "Module" in Chapter 6, "Technical Concepts."

User Interface Composition

The Composite Application Library promotes user interface composition; this means you can implement visual components from various loosely coupled visual components, known as views, which may reside in separate modules. The visual components may display content from multiple back-end systems. To the user, it appears as one seamless application.

The Composite Application Library helps you compose commands and create events that are executed across modules using a loosely coupled approach. Composing a user interface of loosely coupled views across modules usually requires commands and events to also be loosely coupled.

For more information about the user interface composition, see the following:

- "Shell and View" in Chapter 6, "Technical Concepts"
- "Region" in Chapter 6, "Technical Concepts"
- "UI Composition" in Chapter 2, "Design Concepts"

Extensibility

The Composite Application Library provides the foundation for building WPF applications; its design promotes extensibility in many different ways. With it, you can do the following:

- Replace the services and strategies provided as defaults (such as the way in which modules are loaded) with your own implementations.
- Add custom services and behaviors as required by your applications.
- Use your own container or logging service.

For more information about extensibility, see "Customizing the Composite Application Library" later in this chapter.

Adoption Experience

The Composite Application Library was designed in such a way that you can take the services you need to help solve your problem without having to use the entire library. For example, if you need to support a decoupled communication scenario, you can take the **EventAggregator** service without using any of the other services provided by the library.

Organization of the Composite Application Library

The Composite Application Library consists of three assemblies:

- **Microsoft.Practices.Composite**. This assembly contains interfaces and components to help build composite applications that are not specific to a user interface technology. These components include the **EventAggregator**, **TraceLogger**, **ModuleLoader**, **StaticModuleEnumerator**, **ConfigurationModuleEnumerator**, and **DirectoryLookupModuleEnumerator**.
- **Microsoft.Practices.Composite.Wpf.** This assembly contains interfaces and components to help build composite applications that are specific to WPF. These components include **CompositeCommand**, **DelegateCommand**, **CompositeWpfEvent**, and **RegionManager**.
- **Microsoft.Practices.Composite.UnityExtensions.** This assembly provides components to use the Unity Application Block with the Composite Application Library for WPF. These components include **UnityBootstrapper** and **UnityContainerAdapter**.

Applications based on the Composite Application Library are typically organized by shell, module, and shared projects. Figure 4.6 illustrates the solution layout for the Stock Trader RI.

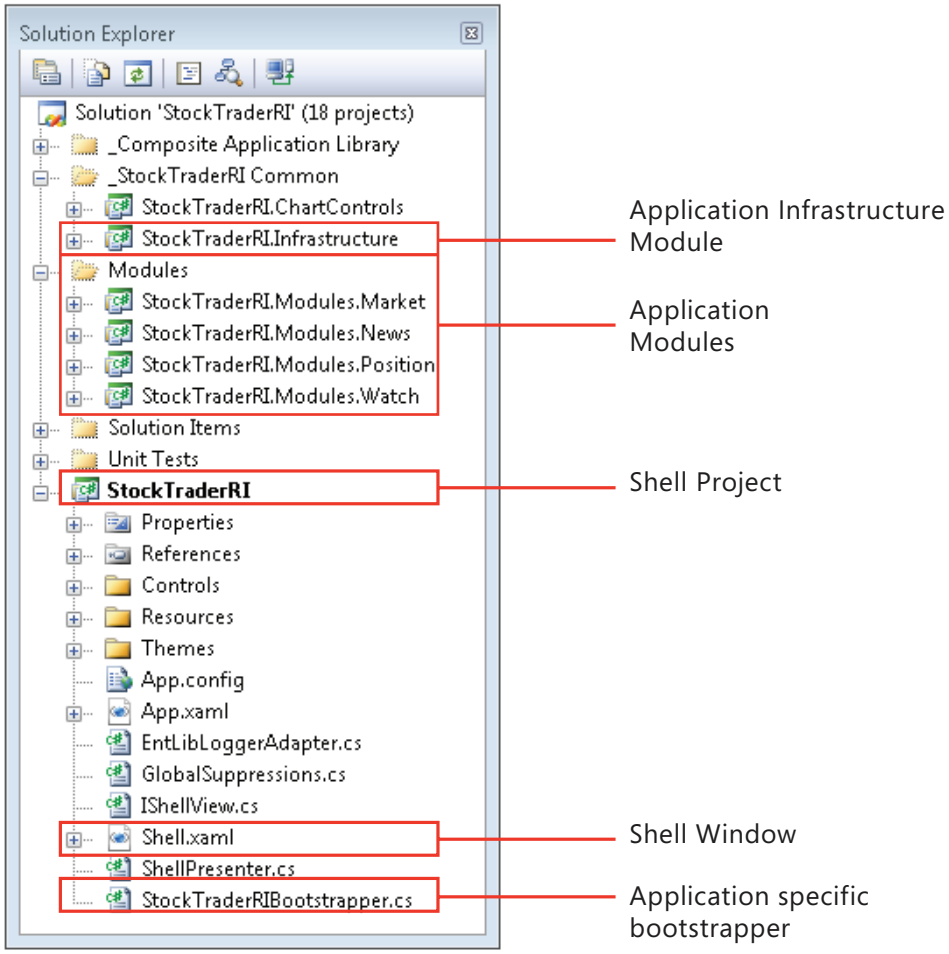


Figure 4.6 *Solution layout*

Technical Concepts

There are a number of technical concepts specific to the Composite Application Library. These technical concepts are implementations of the design patterns and concepts identified as important for building composite user interface (UI) applications. These technical concepts include "Bootstrapper," "Container and Services," "Module," "Region," "Shell and View," "Event Aggregator," "Commands," and "Communication." For details about the technical concepts, see Chapter 6, "Technical Concepts."

Development Activities

"Development Activities" in Chapter 8, "Development, Customization, and Deployment Information," contains references to the How-to topics that you should use as a point of reference when creating a composite application with the Composite Application Library. These topics describe creating your solution and working with the bootstrapper, modules, commands, events, regions, services, and views.

Deploying Your Application

You can deploy and update your application using familiar desktop-based techniques, such as by using a Windows Installer package or ClickOnce. Some of these scenarios introduce more challenges because there are often no references between assemblies, unlike in non-composite applications. Therefore, tools that support building Windows Installer packages or ClickOnce packages are unable to automatically locate the required pieces for deployment. The Composite Application Library offers some tools and techniques to help with managing ClickOnce deployments. For more information, see "Deployment Activities" in Chapter 8, "Development, Customization, and Deployment Information."

Customizing the Composite Application Library

The Composite Application Guidance for WPF contains assets that represent recommended practices for WPF client development. Developers can use an unmodified version of the guidance to create WPF applications. However, because each application is unique, you should analyze whether the Composite Application Guidance 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 Composite Application Library can serve as the foundation for your WPF client applications. The Composite Application 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 due to 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 an enumeration strategy for module loading, calling your own logger, and creating your own region adapters.

Guidelines for Extensibility

Use these guidelines when you extend the Composite Application 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 the libraries 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 Composite Application Library provides extensibility points that developers can use to tailor the library to suit their needs. For example, when using the Composite Application 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 that 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 Composite Application Library

When modifying the source code, follow these best practices:

- Make sure you understand how the library works by reading the section that describes 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.
- 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 Composite Application Library

Use the following reference list to identify the extension points, by functional area, and associated information for extending the library.

Regions

The Composite Application 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 WPF 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.

For more information about the regions, region adapters, and the region manager and customization, see the following:

- "Region" in Chapter 6, "Technical Concepts"
- "How to: Create a Custom Region Adapter" on MSDN

Logging

A number of components in the Composite Application Library 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.

For information about providing custom loggers, see "How to: Provide a Custom Logger" on MSDN.

Modules

The Composite Application Library provides a number of choices for locating and loading modules; however, your scenario may have needs that the library does not provide, such as locating or loading modules from a database.

Module loading is separated into enumeration and loading, represented by the **IModuleEnumerator** and **IModuleLoader** interfaces. The **UnityBootstrapper** uses the enumerator to retrieve the module information and supplies the results to the module loader.

For more background information about module enumeration and loading, see "Module" in Chapter 6, "Technical Concepts."

Communication

The two main forms of communication in the Composite Application Library are commands and events. Commands are handled by the **DelegateCommand** and **CompositeCommand** classes. Although these provide solutions to common command needs in composite applications, you may want to customize these classes for other scenarios. For example, you may need an alternate strategy for aggregating execution than what **CompositeCommand** offers.

The **EventAggregator** and **CompositeWpfEvent** can connect publishers and subscribers in a decoupled manner. If you need to change how events are located or created, look into extending or replacing the **EventAggregator**. If the event dispatching strategy does not suite your needs, consider extending or replacing the **CompositeWpfEvent**.

To discover more about commands and eventing in the Composite Application Library, see the following sections:

- "Commands" in Chapter 6, "Technical Concepts"
- "Communication" in Chapter 6, "Technical Concepts"
- "Event Aggregator" in Chapter 6, "Technical Concepts"

Container and Bootstrapper

The Composite Application Library comes with the Unity container; however, because the container is accessed through the **IContainerFacade** interface, the container can be replaced. To do this, your container will need to satisfy the **IContainerFacade** interface. Usually, if you are replacing the container, you will also need to provide your own container-specific bootstrapper. To provide this, examine the **UnityBootstrapper** to ensure the container is setup and registered with the services needed for the Composite Application Library.

For more information about the container and bootstrapper, see the following sections:

- "Container" in Chapter 2, "Design Concepts"
- "Container and Services" in Chapter 6, "Technical Concepts"
- "Bootstrapper" in Chapter 6, "Technical Concepts"

5

Stock Trader Reference Implementation

The Composite Application Guidance for WPF (Windows Presentation Foundation) includes a reference implementation, which is an application 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.

The reference implementation is not a real-world application; however, it is based on real-world challenges developers and architects face. When you look at this application, do not look at it as a reference point for building a stock trader application—instead, look at is 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 the Composite Application Guidance is to address challenges around building composite applications. For this reason, certain scenarios are used in the reference implementation to emphasize those challenges.

Figure 5.1 on the next page illustrates the Stock Trader Reference Implementation (Stock Trader RI).

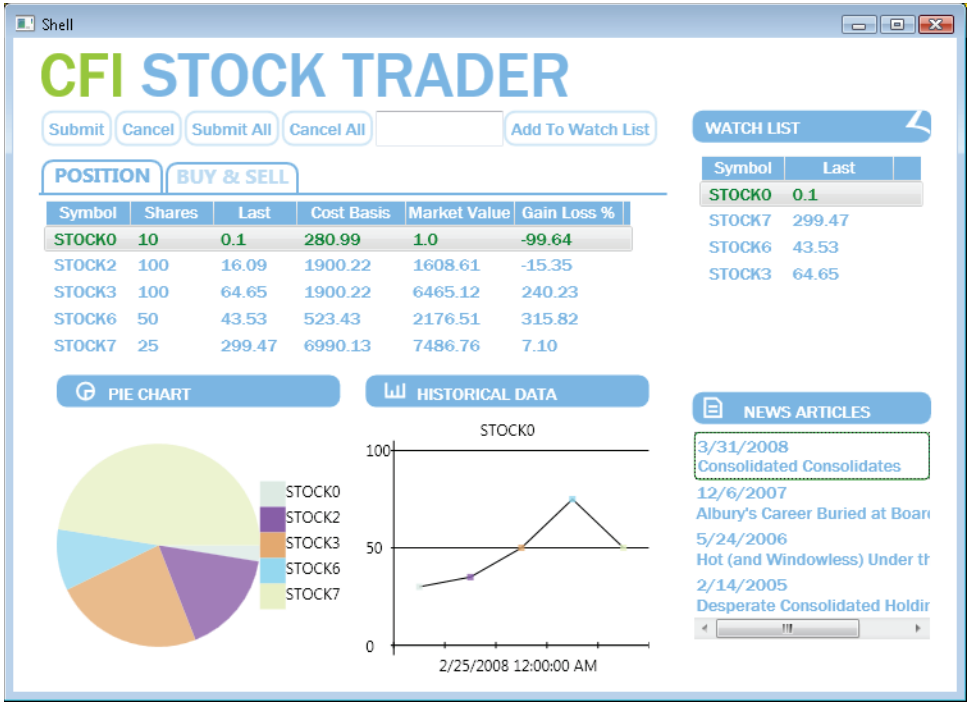


Figure 5.1 *Stock Trader RI*

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.

This chapter describes the scenario that set the context for the reference implementation and helped drive the requirements to illustrate a realistic complex, composite application. It also describes the logical architecture and includes a walkthrough of the main elements of the Stock Trader RI. To run the Stock Trader RI, see "Installing and Running" on MSDN.

Note: Before you can run the Stock Trader RI, you must download and install the Composite Application Guidance for WPF. For information, including download information, see "Composite Application Guidance for WPF" on MSDN at *http://www.microsoft.com/CompositeWpf*.

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 its stock traders. This section 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 the Composite Application Guidance.

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 its own silo meant that CFI could remove any contention that might arise, and it would pave the way for easily creating new teams. This means CFI could 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, each 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 switch 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 login 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 user interface.

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 each 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 frequently change. 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 following are some of the inefficient operating conditions:

- 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 standalone rich client experience for managing their portfolios. Clients can 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 how to take 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 user interface 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 user interface 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 corporate branding and user interface 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 parties.
- Create a new customer-facing product offering. This offering should do the following:
  - It should include a stand-alone rich client for portfolio management.
  - It should provide user interface (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: Composite Application Guidance for WPF

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, the architect decides that the best solution can be found in the Composite Application Guidance offered by the Microsoft patterns & practices team.

The Composite Application Guidance is a set of assets for building complex WPF applications. The Composite Application Guidance 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 user interface 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.

The Composite Application Guidance 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. The architect 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 utilize WPF. After the architect presents these findings to the CTO, the CTO agrees that the Composite Application Guidance will help to deliver an effective solution efficiently and cost-effectively. The CTO 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, they can see their portfolio, view trend data, buy and sell shares, manage items in their watch list, 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 corresponding 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.
- View your buy and sell orders.
- Submit or cancel your entire buy and sell orders.

Logical Architecture

Figure 5.2 on the next page illustrates a high-level logical architecture view of the Stock Trader RI.

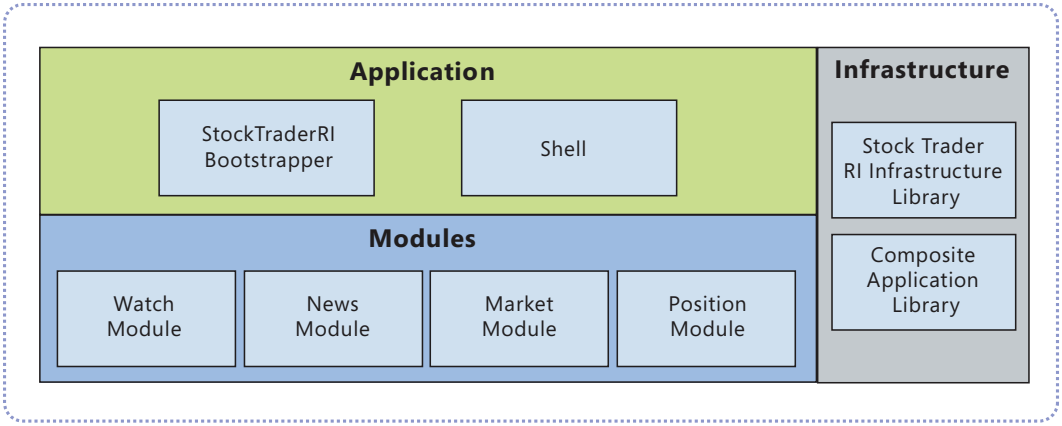


Figure 5.2 *Architectural view of the Stock Trader RI*

The following are 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 Composite Application Guidance core:
  - **Composite Application 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. The **StockTraderRIBootstrapper** inherits from the **UnityBoostrapper**.
  - **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 Composite Application Library. Figure 5.3 shows the Stock Trader RI Solution Explorer.

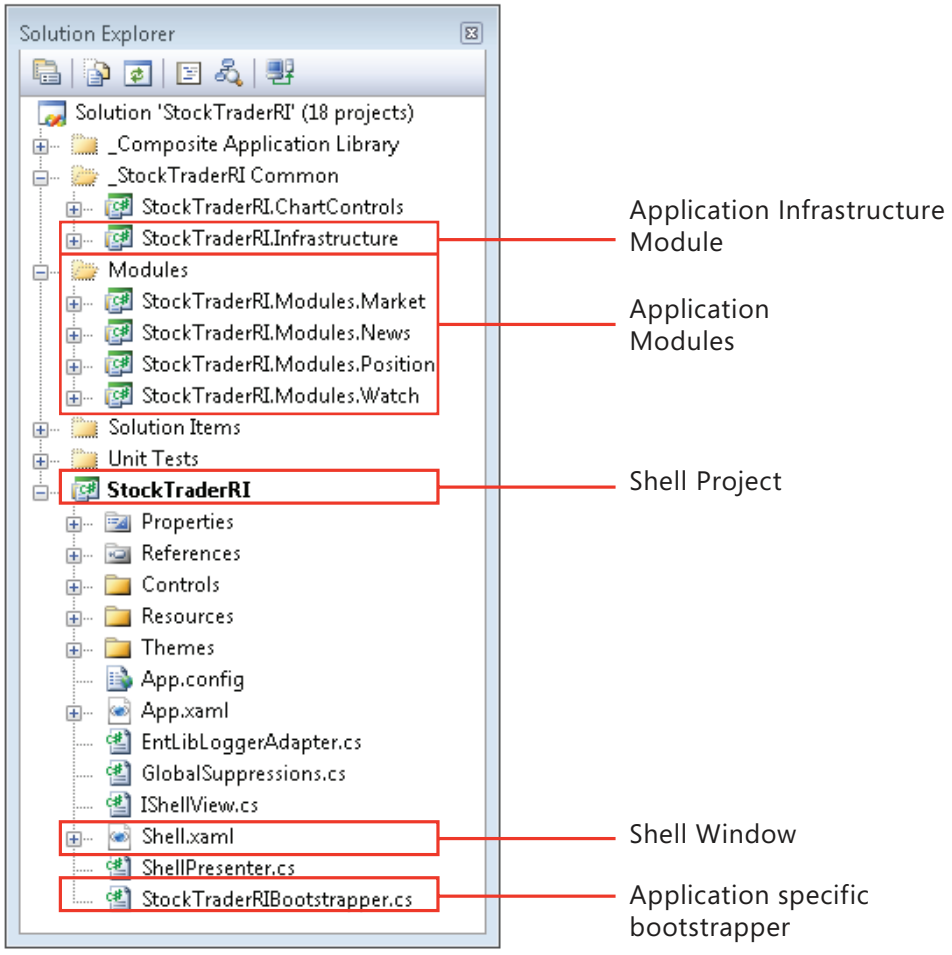


Figure 5.3 *Stock Trader RI solution view*

The next section describes the main elements.

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. Figure 5.4 illustrates the application's startup process, which includes the initialization of modules. The next sections provide details about each of these steps.

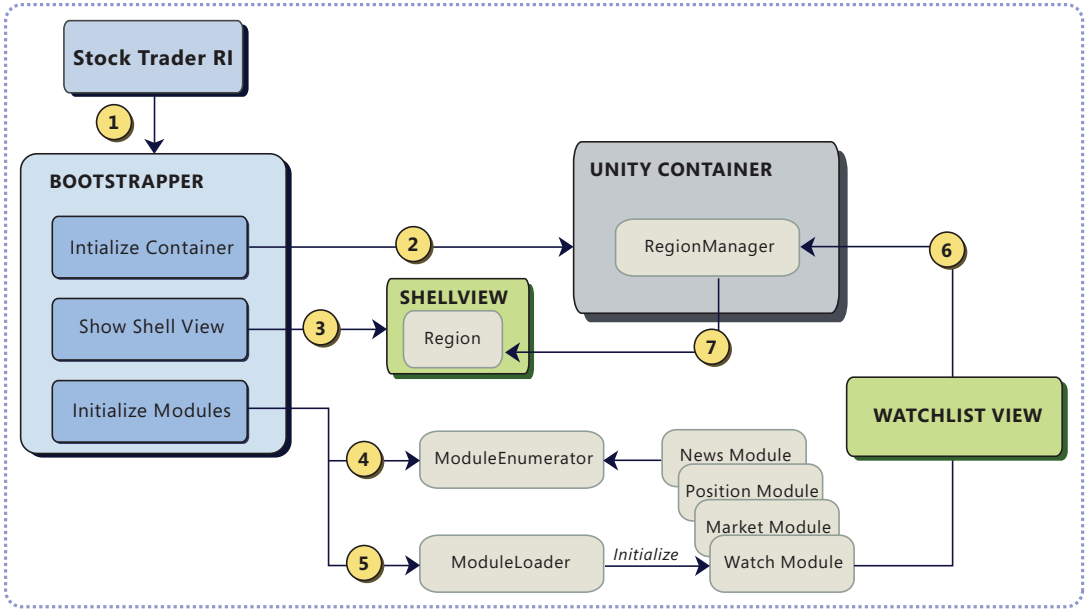


Figure 5.4 *Stock Trader RI startup process*

The Stock Trader RI startup process is the following:

1. The application uses the **StockTraderRIBootstrapper**, which inherits from the Composite Application Library's **UnityBootstrapper** for its initialization.
2. The **UnityBootstrapper** initializes the **UnityContainerAdapter** for use in the modules.
3. The **StockTraderRIBootstrapper** creates and shows the Shell view.
4. The Composite Application Library's **StaticModuleEnumerator** finds all the modules the application needs to load.
5. The Composite Application Library's **ModuleLoader** loads and initializes each of the modules.
6. Modules use the Composite Application Library's **RegionManager** service to add a view to a region.
7. The Composite Application 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.

The application does not direct each module; 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 this composability, 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 (DI) container (referred to as the container) injects these services into modules during their initialization (the Stock Trader RI, uses the Unity 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 **UnityBootstrapper**. The **UnityBootstrapper** is responsible for starting the core composition services used in an application created with the Composite Application Library. For more information, see "Bootstrapper" in Chapter 6, "Technical Concepts."

C# UnityBootstrapper.cs

```
protected virtual void InitializeModules()
        {
            IModuleEnumerator moduleEnumerator = Container.TryResolve<IModuleEnum
erator>();
            ...
            IModuleLoader moduleLoader = Container.TryResolve<IModuleLoader>();
            ...
            ModuleInfo[] moduleInfo = moduleEnumerator.GetStartupLoadedModules();
            moduleLoader.Initialize(moduleInfo);
        }
```

Module Enumeration

To enumerate a module, the bootstrapper uses an **IModuleEnumerator**, which is returned from the **GetModuleEnumerator** method. The bootstrapper in the Stock Trader RI overrides this method to return a **StaticModuleEnumerator** prepopulated with the Stock Trader RI module metadata. The **StaticModuleEnumerator** specifies a static list of modules that the Shell directly references.

C# StockTraderRIBootstrapper.cs

```
protected override IModuleEnumerator GetModuleEnumerator()
{
    return new StaticModuleEnumerator()
        .AddModule(typeof(NewsModule))
        .AddModule(typeof(MarketModule))
        .AddModule(typeof(WatchModule), "MarketModule")
        .AddModule(typeof(PositionModule), "MarketModule", "NewsModule");
}
```

Module Loading

To initialize a module, the **ModuleLoader** service first resolves the module from the container. During this resolving process, the container will inject services into the modules constructor. The following code shows that the region manager is injected.

C# WatchModule.cs

```
public WatchModule(IUnityContainer container, IRegionManager regionManager)
{
    _container = container;
    _regionManager = regionManager;
}
```

After that, the **ModuleLoader** calls the module's **Initialize** method, as shown here.

C# WatchModule.cs

```
public void Initialize()
{
    RegisterViewsAndServices();
     ...
}
```

Views

After a module is initialized, it can use these services to access the Shell (essentially, the top-level window, which contains all the content) where it can add views. A view is any content that a module contributes to the UI.

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

Modules can register views in the container, where they can be resolved. The following code shows where the **WatchListView** and **WatchListPresentationModel** are registered with the container. In the Stock Trader RI, this registration generally happens in the module's **RegisterViewsAndServices** method.

C# WatchModule.cs

```
protected void RegisterViewsAndServices()
{
   ...
  _container.RegisterType<IWatchListView, WatchListView>();
  _container.RegisterType<IWatchListPresentationModel, WatchListPresentation_
      Model>();
  ...
}
```

Note: You can also register views and services using configuration instead of code.

After they are registered, they can be retrieved from the container either by explicitly resolving them or through constructor injection.

C# WatchModule.cs

```
public void Initialize()
{
   ...
   IWatchListPresentationModel watchListPresentationModel = _container.Resolve_
       <IWatchListPresentationModel>();
}
```

Presentation Model

The Stock Trader RI uses several UI design patterns for separated presentation. One of them is the Presentation Model pattern. Using the Presentation Model, you can separate the UI rendering (the view) from the UI business logic (the presenter or, in this case, presentation model). Doing this allows the presentation model to be unit tested because the view can be mocked. It also makes the UI logic more maintainable.

In our implementation of the Presentation Model, the view is injected into the presentation model during its creation. The caller who created the presentation model (in this case, the module) can access the **View** property to get the view.

Regions and the RegionManager

After the view is created, it needs to be shown in the shell. In an application created with the Composite Application Library, you use a region, which is a named location in the UI, for this purpose. Using the **RegionManager**, a module gets a region and adds views, shows views, or removes views. The module accesses the region through an **IRegion** interface. It does not have direct knowledge of how the region will handle displaying the view.

The following code shows where the Watch module adds the Watch List view to the "Watch Region."

C# WatchModule.cs

```
public void Initialize()
{
    ...
    _regionManager.Regions["WatchRegion"].Add(watchListPresentationModel.View);
    ...
}
```

This region was defined in the shell in its XAML using the **RegionName** attached property, as shown here.

XAML Shell.xaml

```
<StackPanel Grid.Row="1" Grid.RowSpan="2" Grid.Column="3">
    <Controls:TearOffItemsControl x:Name="TearOffControl"
        cal:RegionManager.RegionName="WatchRegion"
        .../>
    <ItemsControl Margin="0,20,0,0" cal:RegionManager.RegionName="NewsRegion" />
</StackPanel>
```

Figure 5.5 shows how the watch list appears in the application.

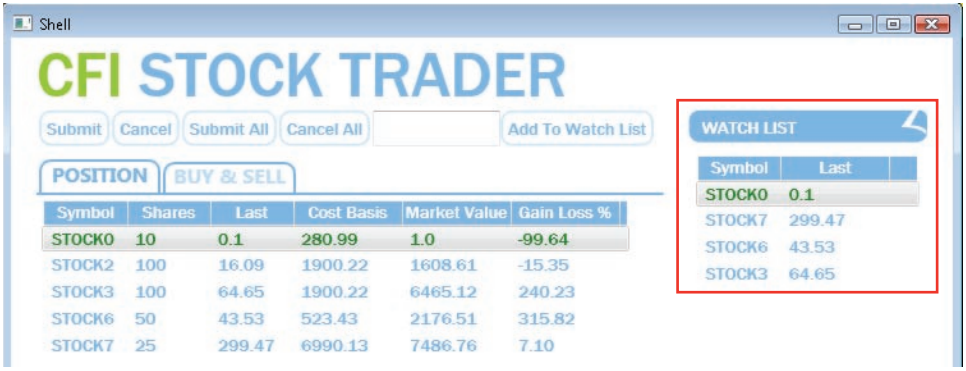


Figure 5.5 *CFI Stock Trader watch list*

Service Registration

Modules can also register services so they can be accessed by either the same module or other modules in a loosely coupled fashion. In the following code, the **WatchListService**, which manages the list of watch items, is registered by the Watch module.

C# WatchModule.cs

```
protected void RegisterViewsAndServices()
{
    _container.RegisterType<IWatchListService, WatchListService>(new Container_
        ControlledLifetimeManager());
    ...
}
```

After that, the container injects the **WatchListService** into the Watch module's **WatchListPresentationModel**, which accesses it through the **IWatchListService** interface.

C# WatchListPresenter.cs

```
public WatchListPresentationModel(IWatchListView view, IWatchListService _
    watchListService, IMarketFeedService marketFeedService, IEventAggregator
    eventAggregator)
{
   ...
   this.watchList = watchListService.RetrieveWatchList();
   ...
}
```

Commands

Views can communicate with presenters and services in a loosely coupled fashion by using commands. The **Add To Watch List** button, illustrated in Figure 5.6, 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 Composite Application Library provides. For more information about commanding in the Composite Application Guidance, see "Commands" in Chapter 6, "Technical Concepts."

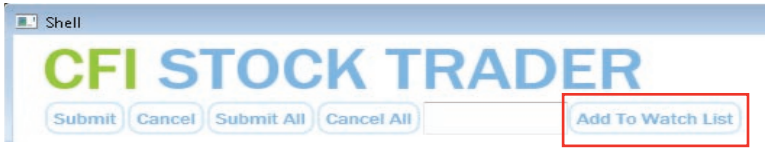


Figure 5.6 *Add To Watch List button*

By using a **DelegateCommand**, the service can delegate the command's **CanExecute** method to the service's **AddWatch** method, as shown in the following code.

C# WatchListService.cs

```
public WatchListService(IMarketFeedService marketFeedService)
{
    ...
    AddWatchCommand = new DelegateCommand<string>(AddWatch);
    ...
}

private void AddWatch(string tickerSymbol)
{
    ...
}
```

The **WatchListService** is also injected into the **AddWatchPresenter**, which calls the view's **SetAddWatchCommand** method in its constructor, as shown here.

C# AddWatchPresenter.cs

```
public class AddWatchPresenter : IAddWatchPresenter
{
    public AddWatchPresenter(IAddWatchView view, IWatchListService service)
        {
            View = view;
            View.SetAddWatchCommand(service.AddWatchCommand);
        }
    public IAddWatchView View { get; private set; }
}
```

This method sets the **AddWatchView**'s **DataContext** to the command, as shown here.

C# AddWatchView.xaml.cs

```
public void SetAddWatchCommand(ICommand addWatchCommand)
{
    this.DataContext = addWatchCommand;
}
```

The **AddWatchButton** then binds to the command (through the **DataContext**) with the command parameter binding to the **AddWatchTextBox.Text** property.

XAML AddWatchView.xaml

```
<StackPanel Orientation="Horizontal">
    <TextBox Name="AddWatchTextBox" MinWidth="100"/>
    <Button Name="AddWatchButton" DockPanel.Dock="Right" Command="{Binding}"
CommandParameter="{Binding Text, ElementName=AddWatchTextBox}">Add To Watchlist_
    </Button>
</StackPanel>
```

This means that when the **Add To Watch List** button is clicked, the **AddWatchCommand** will be invoked and the stock symbol will be passed 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 Composite Application 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 **PositionSummaryPresentationModel** in the Position module raises an event that specifies the symbol that was selected, as shown in the following code.

C# PositionSummaryPresentationModel.cs

```
EventAggregator.Get<TickerSymbolSelectedEvent>().Publish(e.Value);
```

The **NewsController** in the News module listens to the event and notifies the **ArticlePresentationModel** to display the news related to the selected symbol, as shown in the following code.

C# NewsController.cs

```
this.regionManager.Regions["NewsRegion"].Add(articlePresentationModel.View);
eventAggregator.Get<TickerSymbolSelectedEvent>().Subscribe(ShowNews, _
    ThreadOption.UIThread);
```

Note: The NewsController subscribes to the event in the UI thread to safely update the UI and avoid a WPF exception.

Technical Challenges

The 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 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 Orders view. Position, Watch, and News module initializers adding content to regions.	StockTraderRI\Shell.xaml StockTraderRI.Modules.Position\Orders\OrdersView.xaml StockTraderRI.Modules.Position\PositionModule.cs StockTraderRI.Modules.Position\Controllers\OrdersController.cs StockTraderRI.Modules.Watch\WatchModule.cs StockTraderRI.Modules.News\Controllers\NewsController.cs
Composite view: Shows how a composite view communicates with its child view.	The line chart is nested inside the composite view. See the line chart corresponding to a stock.	StockTraderRI.Modules.Position\PositionSummary\PositionSummary_PresentationModel.cs StockTraderRI.Modules.Market\TrendLine\TrendLinePresenter.cs
	Order screen	StockTraderRI.Modules.Position\Orders\OrderCompositePresentationModel.cs StockTraderRI.Modules.Position\Orders\OrderDetailsPresentationModel.cs StockTraderRI.Modules.Position\Orders\OrderCommandsView.xaml.cs StockTraderRI.Modules.Position\Controllers\OrdersController.cs
Compose UI across modules: The Watch module has a view and also is a part of the toolbar.	Add a stock to the watch list	StockTraderRI.Modules.Watch\AddWatchView.xaml StockTraderRI.Modules.Watch\WatchList\WatchListView.xaml
Decoupled communication		
Commands: Shows the Command pattern. The command to buy or sell a stock is a delegate command. This command 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 **PositionGrid** and handlers in **OrdersController**	StockTraderRI.Modules.Position\Controllers\OrdersController.cs StockTraderRI.Modules.Position\PositionGrid.xaml

Technical challenge	Feature in Stock Trader RI	Example of where feature is demonstrated
Composite commands: Use composite commands to broadcast all of the commands. The **Submit All** or **Cancel All** commands execute all of the individual instances of the **Submit** or **Cancel** commands.	**Submit All** and **Cancel All** buttons	StockTraderRI.Infrastructure\StockTraderRICommands.cs StockTraderRI.Modules.Position\Orders\OrderDetailsPresentationModel.cs StockTraderRI.Modules.Position\Controllers\OrdersController.cs
Active aware commands: Use active aware commands to determine the target model for the command depending on which view is currently active.	**Submit** and **Cancel** buttons	StockTraderRI.Infrastructure\StockTraderRICommands.cs StockTraderRI.Modules.Position\Orders\OrderDetailsPresentationModel.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\PositionSummary_PresentationModel.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 and the consumers then update the model behind the UI.	StockTraderRI.Modules.Market\Services\MarketFeedService.cs StockTraderRI.Modules.Position\PositionSummary\PositionSummary_PresentationModel.cs StockTraderRI.Modules.WatchList\WatchList\WatchListPresentationModel.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

(continued)

Technical challenge	Feature in Stock Trader RI	Example of where feature is demonstrated
Other technical challenges		
WPF: Use WPF for the user interface	Shell and Module views	The starting point for Stock Trader RI is in the StockTraderRI\App.xaml\App.xaml.cs
Bootstrapper: The use of a bootstrapper to initialize the application with global services.	Created bootstrapper with the Unity container and configuring global services, such as logging and module enumeration.	**Bootstrapper**: StockTraderRI\StockTraderRIBootstrapper.cs

6

Technical Concepts

This chapter discusses technical concepts that are specific to the Composite Application Library. These technical concepts, illustrated in Figure 6.1, are implementations of the design patterns and concepts identified as important for building composite user interface (UI) applications.

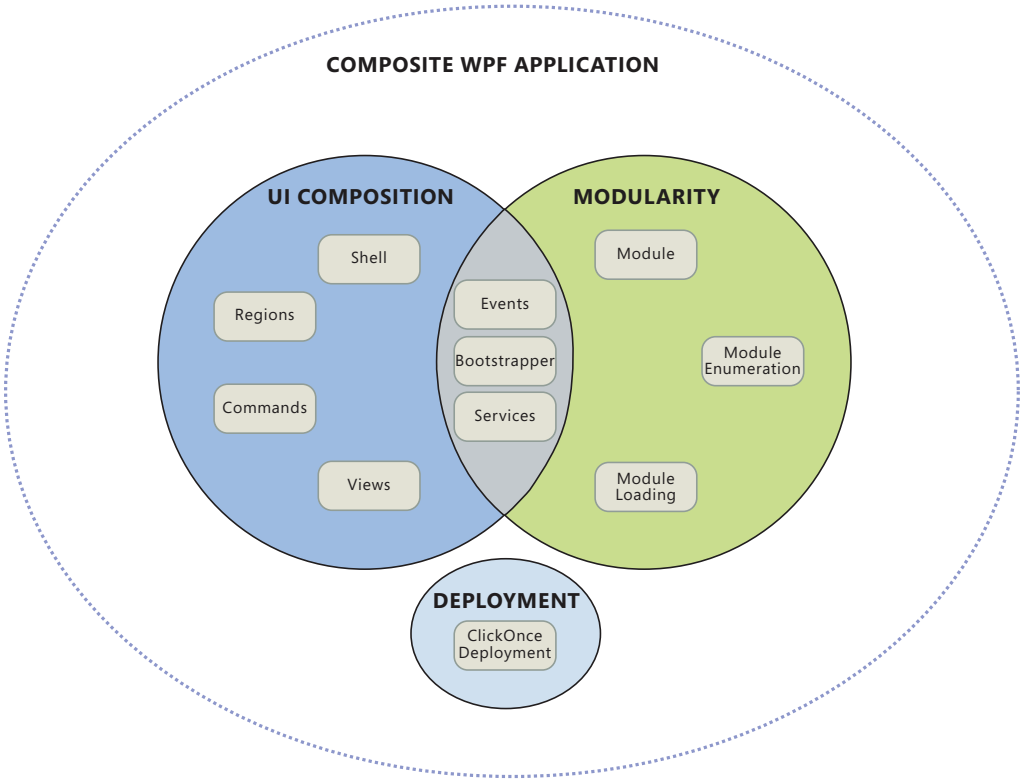


Figure 6.1 *Composite Application Library concepts*

For more details about the technical concepts, see the following sections in this chapter:

- "Bootstrapper"
- "Container and Services"
- "Module"
- "Region"
- "Shell and View"
- "Event Aggregator"
- "Commands"
- "Communication"

Bootstrapper

The bootstrapper is responsible for the initialization of an application built using the Composite Application Library. Having a bootstrapper gives you more control of how the Composite Application Library components are wired up to your application. For example, if you have an existing application that you are adding the Composite Application Library to, you can initialize the bootstrapping process after the application is already running.

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 Composite Application Library, it is the bootstrapper's responsibility to launch 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. Additionally, the shell may rely on other services that are injected into its constructor. For more information about the shell, see the "Shell and View" technical concept later in this chapter.

The Composite Application Library includes a default abstract **UnityBootstrapper** class that handles this initialization using the Unity container. Many of the methods on the **UnityBootstrapper** class are virtual methods. You should override these methods as appropriate in your own custom bootstrapper implementation. If you are using a container other than Unity, you should write your own container-specific bootstrapper.

Figure 6.2 illustrates the stages of the bootstrapping process.

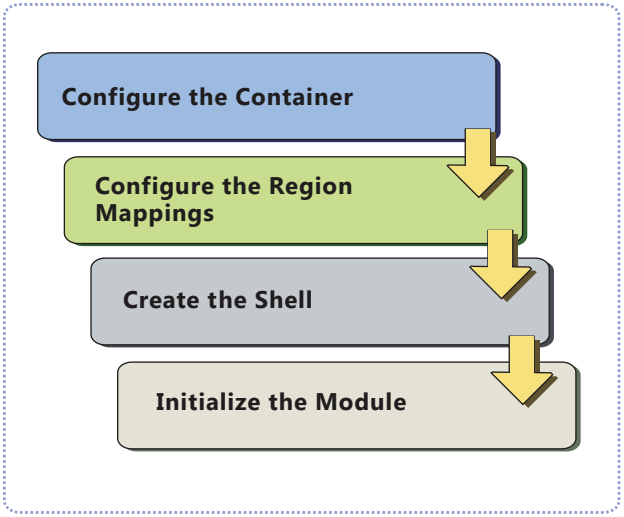


Figure 6.2 *Bootstrapping process*

Configuring the Container

Containers play a key role in an application created with the Composite Application Library. Both the Composite Application Library and the applications built on top of it depend on a container for injecting required dependencies. During the container configuration phase, the container is created and several core services are registered, as shown in the following code from the **UnityBootstrapper**. Notice that the container itself is registered with itself. This allows modules to get direct access to the container to explicitly register and resolve dependencies.

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

C# UnityBootstrapper.cs

```
protected virtual void ConfigureContainer()
{
    ...
    Container.RegisterInstance<IUnityContainer>(Container);
    ...
    if (_useDefaultConfiguration)
    {
        RegisterTypeIfMissing(typeof(IContainerFacade), typeof(UnityContainer_
            Adapter), true);
       RegisterTypeIfMissing(typeof(IEventAggregator), typeof(EventAggregator),_
           true);
       RegisterTypeIfMissing(typeof(RegionAdapterMappings), typeof(RegionAdapter_
           Mappings), true);
       RegisterTypeIfMissing(typeof(IRegionManager), typeof(RegionManager), true);
       RegisterTypeIfMissing(typeof(IModuleLoader), typeof(ModuleLoader), true);
    }
}
```

The bootstrapper will determine 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 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.

Configuring the Region Mappings

During this phase, the default region adapter mappings are registered. 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 the "Region" technical concept later in this chapter.

You can also override the **ConfigureRegionAdapterMappings** method to add your own custom region adapter mappings, as shown in the following code from the **UnityBootstrapper**.

C# UnityBootstrapper.cs

```
protected virtual RegionAdapterMappings ConfigureRegionAdapterMappings()
{
    RegionAdapterMappings regionAdapterMappings = Container.TryResolve<Region_
        AdapterMappings>();
    if (regionAdapterMappings != null)
    {
        regionAdapterMappings.RegisterMapping(typeof(Selector), new SelectorRegion_
            Adapter());
        regionAdapterMappings.RegisterMapping(typeof(ItemsControl), new Items_
            ControlRegionAdapter());
        regionAdapterMappings.RegisterMapping(typeof(ContentControl), new Content_
            ControlRegionAdapter());
     }

     return regionAdapterMappings;
}
```

Creating the Shell

During this phase, the shell will be displayed if it exists. Having the creation of the shell in the bootstrapper allows greater testability of the application because the shell can be mocked in a unit test.

If you are adding the Composite Application Library to an existing application, there may not be a shell. In these instances, you should override the **CreateShell** method to return **null**. For more information about the shell, see the "Shell and View" technical concept later in this chapter. The following code from the file StockTraderRI-Bootstrapper.cs shows the **CreateShell** method used in the Stock Trader Reference Implementation (Stock Trader RI).

C# StockTraderRIBootstrapper.cs

```
protected override DependencyObject CreateShell()
{
    ShellPresenter presenter = Container.Resolve<ShellPresenter>();
    IShellView view = presenter.View;
    view.ShowView();
    return view as DependencyObject;
}
```

Initializing the Modules

During this phase, module loading occurs. First, the module enumerator and module loader services are resolved from the container. After that, the modules that have been specified to load on startup are initialized, as shown in the following code from the **UnityBootstrapper**.

C# UnityBootstrapper.cs

```
protected virtual void InitializeModules()
{
    IModuleEnumerator moduleEnumerator = Container.TryResolve<IModule_
        Enumerator>();
    if (moduleEnumerator == null)
    {
        throw new InvalidOperationException(Resources.NullModuleEnumerator_
            Exception);
    }

    IModuleLoader moduleLoader = Container.TryResolve<IModuleLoader>();
    if (moduleLoader == null)
    {
        throw new InvalidOperationException(Resources.NullModuleLoaderException);
    }

    ModuleInfo[] moduleInfo = moduleEnumerator.GetStartupLoadedModules();
    moduleLoader.Initialize(moduleInfo);
}
```

If you are loading modules statically, you should override this method and use the **StaticModuleEnumerator**, as shown in the following code from the Stock Trader RI.

C# StockTraderRIBootstrapper.cs

```
protected override IModuleEnumerator GetModuleEnumerator()
{
    return new StaticModuleEnumerator()
        .AddModule(typeof(NewsModule))
        .AddModule(typeof(MarketModule))
        .AddModule(typeof(WatchModule), "MarketModule")
        .AddModule(typeof(PositionModule), "MarketModule", "NewsModule");
}
```

For more information about modules, see the “Module” technical concept later in this chapter.

More Information

The following topics contain procedures that customize the bootstrapper class:

- To set up the application's bootstrapper, see "How to: Create a Solution Using the Composite Application Library" on MSDN.
- To configure the bootstrapper to dynamically load modules, see "How to: Dynamically Load Modules" on MSDN.
- To configure the bootstrapper to statically load modules, see "How to: Statically Load Modules" on MSDN.
- To use a different logger in your application that uses the Composite Application Library, see "How to: Provide a Custom Logger" on MSDN.
- To configure the bootstrapper to register additional region adapter mappings, see "How to: Create a Custom Region Adapter" on MSDN.
- To configure the bootstrapper to register services in the application container, see "How to: Register and Use Services" on MSDN.

Container and Services

The Composite Application Library is designed to support other dependency injection containers. Core services, such as the **ModuleLoader** service, are container agnostic. They use the Composite Application Library container or **IContainerFacade** interface for resolving, instead of directly accessing the containers. The Composite Application Library provides the **UnityContainerAdapter**, which is a Unity-specific implementation of this interface. This container is registered by the **UnityBootstrapper**.

IContainerFacade

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

C# IContainerFacade.cs

```
public interface IContainerFacade
{
    object Resolve(Type type);
    object TryResolve(Type type);
}
```

You can see that **IContainerFacade** is used only for resolving; it is not used for registration.

The **ModuleLoader** uses **IContainerFacade** for resolving the module during module loading, as shown in the following code.

C# ModuleLoader.cs – Initalize()

```
IModule module = (IModule)containerFacade.Resolve(type);
module.Initialize();
```

UnityContainerAdapter

The following code shows the implementation of the **UnityContainer**.

C# UnityContainerAdapter.cs

```
public class UnityContainerAdapter : IContainerFacade
{
    private readonly IUnityContainer _unityContainer;

    public UnityContainerAdapter(IUnityContainer unityContainer)
    {
        _unityContainer = unityContainer;
    }

    public object Resolve(Type type)
    {
        return _unityContainer.Resolve(type);
    }

    public object TryResolve(Type type)
    {
        object resolved;

        try
        {
            resolved = Resolve(type);
        }
        catch
        {
            resolved = null;
        }

        return resolved;
    }
}
```

The **Resolve** method calls to the underlying container, except for **TryResolve**, which Unity does not support. This method returns an instance of the type to be resolved if it has been registered; otherwise, it returns **null**.

Considerations for Using IContainerFacade

IContainerFacade 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. Keeping this in mind, the Stock Trader RI uses Unity directly instead of using the **IContainerFacade**. This is the recommend approach for your application development.

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

- 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.

Composite Application Library Services

Applications based on the Composite Application Library are composed through a set of services that the application consumes. These services are injected through the container. 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 Composite Application Library.

Service interface	Description
IModuleEnumerator	Enumerates the modules in the system. The Composite Application Library provides several different enumerators. For more information, see the "Module" technical concept later in this chapter.
IModuleLoader	Loads and 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. The reference implementation uses the Enterprise Library Logging Application Block. This allows you to choose your own logging mechanism.
IContainerFacade	Allows the Composite Application 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 **PositionSummaryPresentationModel** 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	Handles persisting 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.

More Information

For more information about services and containers, see the following:

- "Container" in Chapter 2, "Design Concepts"
- "How to: Register and Use Services" on MSDN

Module

A module in the Composite Application Library is a logical unit in your application. Modules assist in implementing a modularity design. These modules are defined in such a way that they can be discovered and loaded by the application at run time. Because modules are self-contained, they promote separation of concerns in your application. Modules can communicate with other modules and access services through various means. They reduce the friction of maintaining, adding, and removing system functionality. Modules also help with testing and deployment.

A common usage of a module is to represent different portions of the system. The following are some examples of modules:

- A module that contains a specific application feature, such as news
- A module that contains a specific subsystem, such as purchasing, invoicing, and general ledger
- A module that contains infrastructure services, such as logging 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

For example, there are four modules in the Stock Trader RI:

- **NewsModule.** This module is responsible for aggregating and displaying the news related to the user's currently selected investment in his or her portfolio.
- **PositionModule.** This module is responsible for displaying the positions and for handling buy/sell orders.
- **MarketModule.** This module handles aggregating and displaying trend information for the portfolio.
- **WatchModule.** This module handles displaying the watch list, which is a list of funds the user monitors. This module also handles adding and removing items from this list.

Team Development Using Modules

Modules have explicit boundaries, typically by subsystem or feature. These boundaries make it easier for separate teams to develop modules. On large applications, teams may be organized by cross-cutting capabilities in addition to being organized by a specific subsystem or feature. For example, there may be a team assigned to shared components of the application, such as the shell or the common infrastructure module.

The following are the different teams developing the Stock Trader RI, as illustrated in Figure 6.3:

- **Infrastructure team.** This team develops the core cross-cutting services used in the application and cross-module types and interfaces.
- **Postions team.** This team maintains the positions and buy/sell functionality.
- **News team.** This team handles aggregating and displaying news.
- **UI team.** This team manages the shell and contains a set of graphic designers who set the overall appearance of the application. They work across each of the development teams.
- **Operations team.** This team is responsible for managing deploying modules to staging and production. They also manage the master module configuration files.

Figure 6.3 on the next page illustrates an example of a composite WPF application's modules and associated teams.

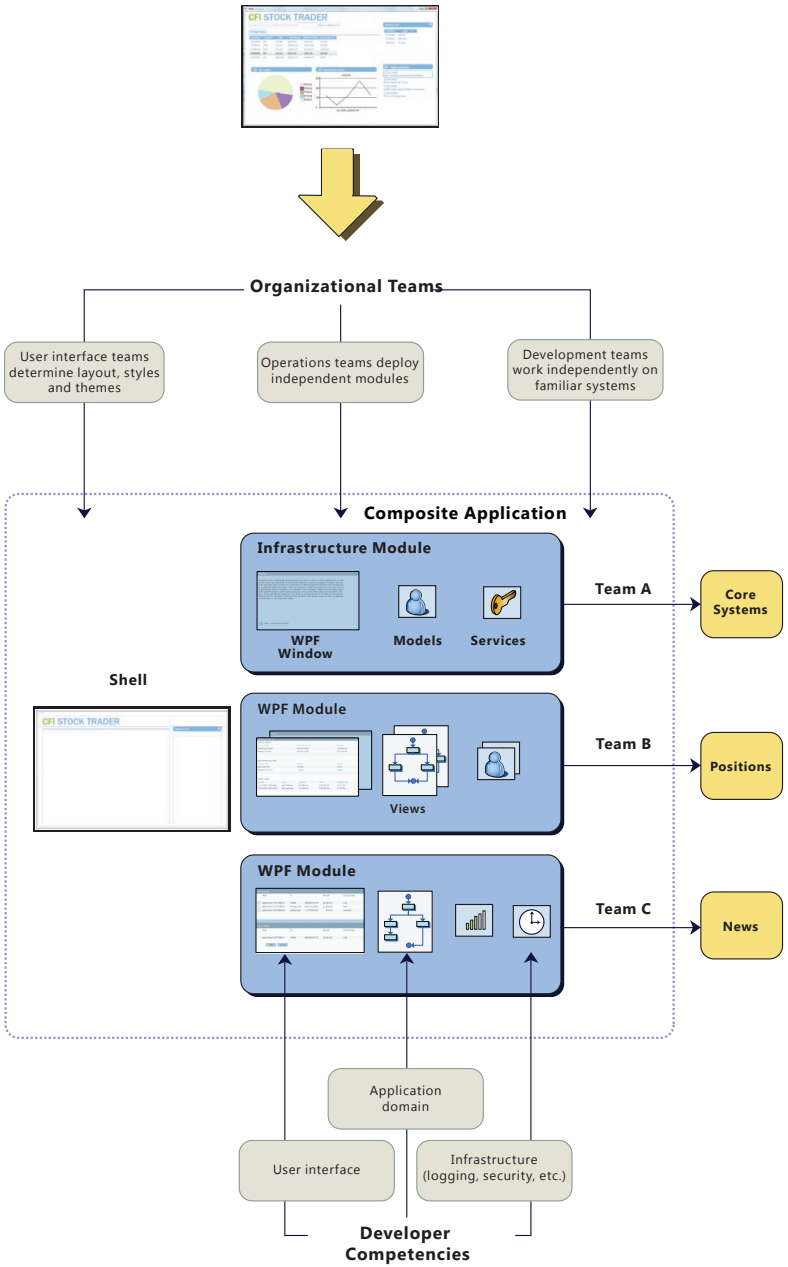


Figure 6.3 *Modules and associated teams of a composite WPF application*

Module Design

Like the application, modules can be designed following layered principals. Modules may consist of a presentation layer, a business or domain layer, and a resource access layer, as illustrated in Figure 6.4.

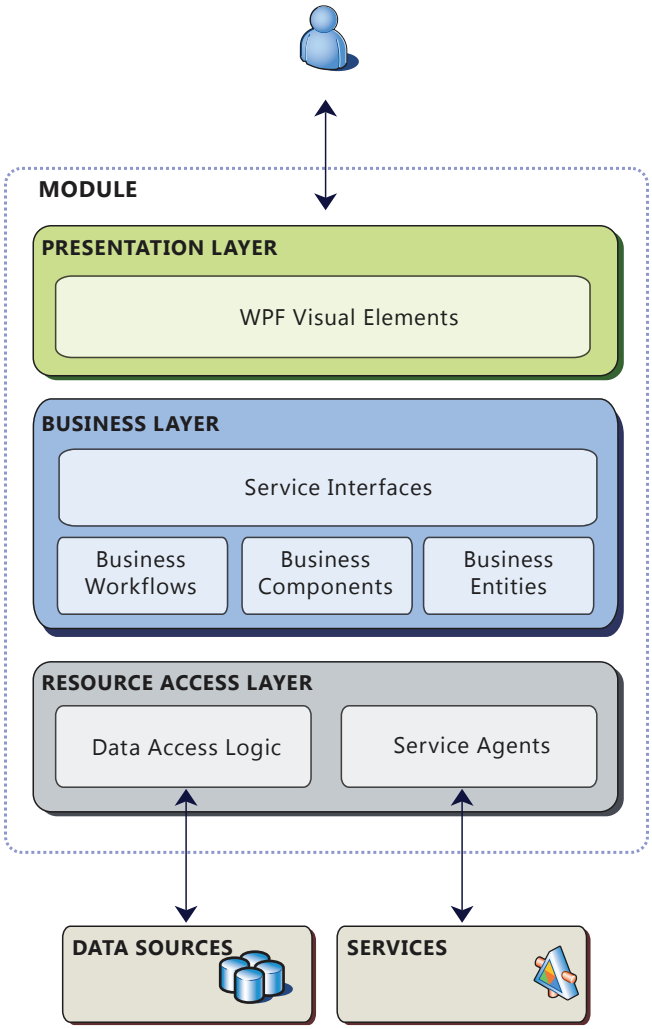


Figure 6.4 *Module design*

These responsibilities may be split across modules. One module may contain the resource access layer that other modules rely on. Or, one module may contain all the views that could then be more easily consumed by user interface designers.

IModule

A module is a class that implements the **IModule** interface. This interface contains a single **Initialize** method that is called during the module's initialization process.

C# IModule.cs

```
public interface IModule
{
    void Initialize();
}
```

Module Dependencies

Modules may need access to dependencies. Because they are constructed by the container, these dependencies are injected through the module's constructor. For example, in the Stock Trader RI, the **container** and the **regionManager** are injected into the **WatchModule**, as shown in the following code.

C# WatchModule.cs

```
public WatchModule(IUnityContainer container, IRegionManager regionManager)
{
    _container = container;
    _regionManager = regionManager;
}
```

View and Service Registration

When a module is initialized, it can register views and services. By using registration, dependencies can be provided through the container and be accessed from other modules.

To do this, the module will need to have the container injected into the module constructor (as shown in the preceding code). The registration is done in the **RegisterViewsAndServices** method, which is not required if there is no registration. The following code shows how the **MarketModule** in the Stock Trader RI registers a **MarketHistoryService**, **MarketFeedService**, and **TrendLineView**.

C# MarketModule.cs

```
protected void RegisterViewsAndServices()
{
    _container.RegisterType<IMarketHistoryService, MarketHistoryService>();
    _container.RegisterType<IMarketFeedService, MarketFeedService>(new Container_
        ControlledLifetimeManager());
    _container.RegisterType<ITrendLineView, TrendLineView>();
    _container.RegisterType<ITrendLinePresenter, TrendLinePresenter>();
}
```

Depending on which container you use, registration can also be done outside of the code through configuration.

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

Displaying Views

During a module's initialization, you may also want to inject a view into a Shell region. To do this, the region manager needs to be injected into the constructor. In the **Initialize** method, you can then locate a region through the **Regions[regionName]** method, and then add a view. In the **Initialize** method of the **Position** module, as shown in the following code, the **PositionSummary** view is added to the **MainRegion**.

C# PositionModule.cs

```
public void Initialize()
{
    RegisterViewsAndServices();

    IPositionSummaryPresentationModel presentationModel = _container.Resolve_
        <IPositionSummaryPresentationModel>();
    IRegion mainRegion = _regionManager.Regions["MainRegion"];
    mainRegion.Add(presentationModel.View);
    ...
}
```

Note: Before resolving a view, make sure that it is registered with the container.

Considerations for Modules

When creating your modules, consider the following:

- Consider putting each module in a separate namespace.
- Consider not having one module directly accessing concrete types from another module. Use interfaces instead. This reduces coupling, thereby increasing testability and prevents circular references between modules.

Module Loading

Module loading in the Composite Application Library is a two-step process:

1. The module enumerator discovers modules and creates a collection of metadata about those modules.
2. The module loader instantiates the module and calls its **Initialize** method.

Figure 6.5 on the next page illustrates module loading in the Composite Application Library.

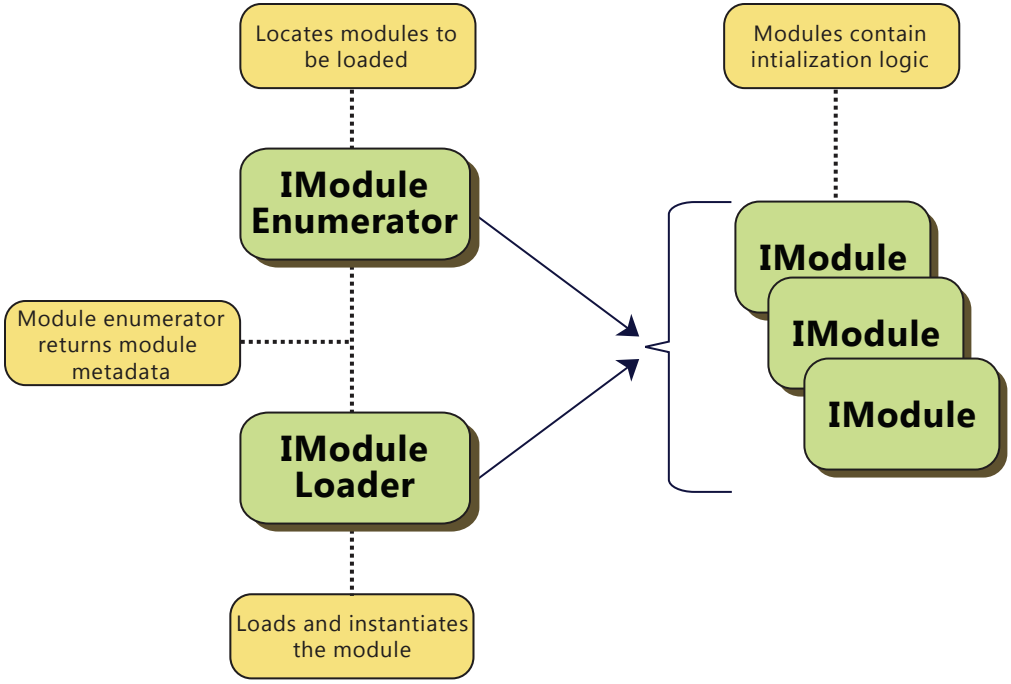


Figure 6.5 *Module loading*

Types of Module Loading

The Composite Application Library provides several ways to load modules by default. You can also provide your own custom implementation.

Static Module Loading

In static module loading, the shell has a direct reference to the modules, but the modules themselves do not directly reference the shell. They can be in the same assembly or a different assembly. The benefit of this style of loading over dynamic module loading is that modules are easier to use and debug because they are directly referenced. The disadvantage of static module loading is that you have to add new references to the shell and a line of code to the bootstrapper for each module.

The Stock Trader RI uses this style of loading. Figure 6.6 illustrates the Stock Trader RI references and highlights the shell references to the modules.

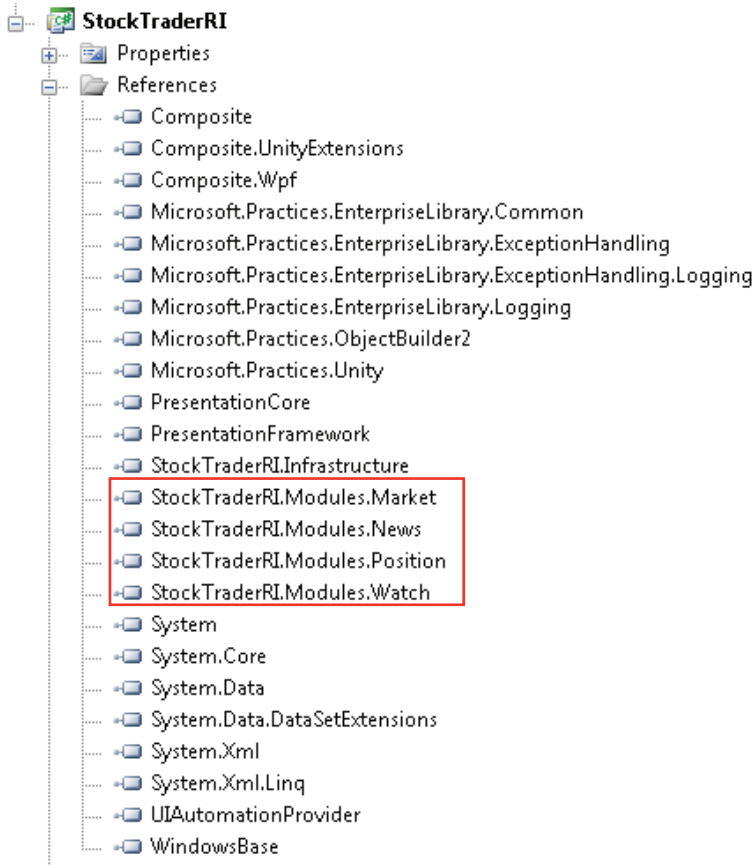


Figure 6.6 *Stock Trader RI references*

The following code shows where the modules are statically defined in the Stock Trader RI bootstrapper.

C# StockTraderRIBootstrapper.cs

```
protected override IModuleEnumerator GetModuleEnumerator()
{
    return new StaticModuleEnumerator()
        .AddModule(typeof (NewsModule))
        .AddModule(typeof (MarketModule))
        .AddModule(typeof (WatchModule), "MarketModule")
        .AddModule(typeof (PositionModule), "MarketModule", "NewsModule");
}
```

Dynamic Module Loading

In dynamic module loading, modules are discovered at run time, which the shell does not directly reference. The advantage of dynamic module loading is that modules can be added to the application without having to add any new references or modify the executable. The other advantage is that it gives you flexibility as to how modules are discovered; for example, the Composite Application Library ships with support for loading modules by scanning a directory, loading modules specified in a configuration file, or loading modules on demand.

Directory Driven Module Loading

In this style of loading, modules are located by the **DirectoryLookupModuleEnumerator**, which scans the assemblies in a directory and locates all the types that implement **IModule**. It will also look for the **ModuleDependency** attribute to determine the dependent modules that need to be loaded before loading the current module. For an example of this style of loading, see the Dynamic Modularity (DirectoryLookup) QuickStart included with the Composite Application for WPF Guidance. The following code shows where Module A and Module B are defined and that Module B depends on Module A.

C#

```
[Module(ModuleName = "ModuleA")]
public class ModuleA : IModule
{
     ...
}
```

C#

```
[Module(ModuleName = "ModuleB")]
[ModuleDependency("ModuleA")]
public class ModuleB : IModule
{
     ...
}
```

Configuration Driven Module Loading

In this style of loading, modules are located by the **ConfigurationModuleEnumerator**, which scans the App.config file for modules and module dependencies. The following example shows where Module A and Module B are defined and that Module B depends on Module A. For an example of this style of loading, see the Dynamic Modularity (Configuration) QuickStart included with the Composite Application for WPF Guidance.

XML App.config

```
<modules>
  <module assemblyFile="Modules/ModuleA.dll" moduleType="ModuleA.ModuleA"
moduleName="ModuleA">
          ...
  </module>
  <module assemblyFile="Modules/ModuleB.dll" moduleType="ModuleB.ModuleB"
moduleName=”ModuleB”>
    <dependencies>
      <dependency moduleName="ModuleA"/>
    </dependencies>
  </module>
  ...
</modules>
```

On-Demand Module Loading

In this style of loading, modules are loaded on demand in the application. The loader will use any available module enumerator to load the module (including the **DirectoryLookupModuleEnumerator** and the **ConfigurationModuleEnumerator**). For examples of this style of loading, see the Dynamic Modularity (Configuration) QuickStart and the Dynamic Modularity (DirectoryLookup) QuickStart included with the Composite Application for WPF Guidance. The following code shows an example of Module C loading on demand in the Dynamic Modularity (Configuration) QuickStart.

C# DefaultViewB.xaml.cs

```
private void OnLoadModuleCClick(object sender, RoutedEventArgs e)
{
    moduleLoader.Initialize(moduleEnumerator.GetModule("ModuleC"));
}
```

Module Enumerator

The module enumerator identifies the modules to be loaded. Each module enumerator must implement the **IModuleEnumerator** interface. The Composite Application Library includes three enumerators: **StaticModuleEnumerator**, **DirectoryLookupEnumerator**, and **ConfigurationModuleEnumerator**. You could write your own module enumerator that obtains module information over a Web service or from a database by simply implementing the **IModuleEnumerator** interface and retrieving the module information from wherever is appropriate for your situation, as shown in the following code.

C# IModuleEnumerator.cs

```
public interface IModuleEnumerator
{
    ModuleInfo[] GetModules();
    ModuleInfo[] GetStartupLoadedModules();
    ModuleInfo GetModule(string moduleName);
}
```

The following describes each method in the preceding code:

- **GetModules.** This method returns a list of all known modules, including those that load during startup and those that load on-demand.
- **GetStartupLoadedModules.** This method returns only those modules that will load at application start (determined either through the **ModuleAttribute** or in configuration).
- **GetModule.** This method returns a specific module.

ModuleInfo

The module enumerator returns a list of module metadata that the module loader will use to load and initialize the module.

Note: Modules that are statically loaded are loaded by the common language runtime (CLR) instead of by the ModuleLoader service. However, they are initialized.

The following code shows the **ModuleInfo** constructor.

C# ModuleInfo.cs

```
public ModuleInfo(string assemblyFile, string moduleType, string moduleName,
params string[] dependsOn)
    : this(assemblyFile, moduleType, moduleName, true, dependsOn)
{
}
```

In addition to information about the module, the preceding code shows that you can provide a collection of module names that the modules depend on. The module loader will build its load order based on these dependencies.

Module Loader

The module loader, which implements the **IModuleLoader** interface, loads the assemblies that contain the modules. The **IModuleLoader.Initialize** method accepts an array of **ModuleInfo** instances, which determines the list of modules to be loaded. The following code shows the **IModuleLoader** interface.

C# IModuleLoader.cs

```
public interface IModuleLoader
{
    void Initialize(ModuleInfo[] moduleInfos);
}
```

The module loader will use the **ModuleDependencySolver** to parse the list of modules and create a load order based on the module dependencies that have been defined. During loading, the module loader instantiates each module by resolving it off of the container. This is to allow the module dependencies to get injected.

Note: The **ModuleLoader** service does not depend on a specific container; instead, it uses an **IContainerFacade** instance that can wrap any container. By default, the Composite Application Library uses a **UnityContainerAdapter**, but this can be replaced.

After the module is resolved, the **IModule.Initialize** method is invoked—this allows the module's code to execute.

More Information

For more information about modules, see the following:

- "Modularity" in Chapter 2, "Design Concepts"
- "Dynamic Modularity QuickStarts" on MSDN
- Modularity How-to topics on MSDN:
  - "How to: Create a Module"
  - "How to: Dynamically Load Modules"
  - "How to: Statically Load Modules"
- Modularity patterns in Chapter 3, "Patterns in the Composite Application Library" (these patterns provide possible approaches to building composite applications):
  - "Inversion of Control Pattern"
  - "Dependency Injection Pattern"
  - "Service Locator Pattern"

Region

Conceptually, a region is a mechanism that developers can use to expose WPF controls to the application as components that encapsulate a particular visual way of displaying and laying out views. Regions can be accessed by name and support adding or removing views. This decoupling allows the appearance of the application to evolve independently of the views that appear within the region (for more information about views, see "Shell and View" in Chapter 6, "Technical Concepts").

Regions are intended to enable a compositional pattern and are commonly used in template layouts and multiple view layouts.

Template Layout

Regions are frequently used to define a layout template for a view. Components can locate and add content to regions in the template without exact knowledge of how and where the region is visually displayed. This allows the template to change without affecting the components that are adding content to the template. This is common in the shell of an application that defines a standard layout, such as a navigation region and a main content region, as illustrated in Figure 6.7.

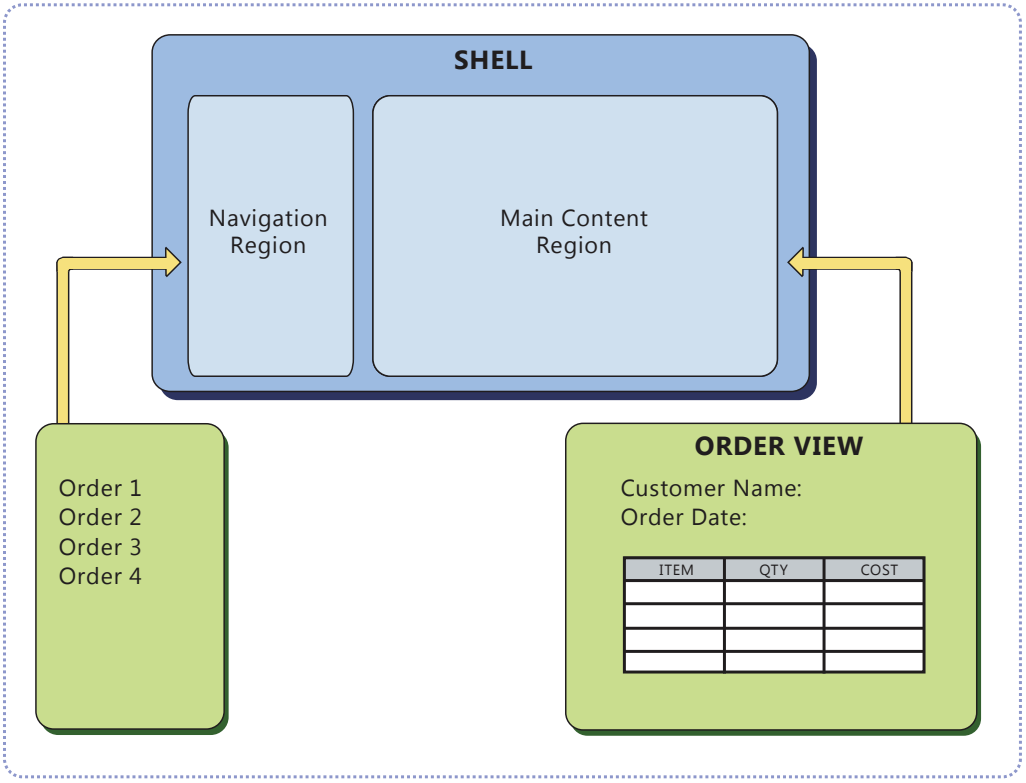


Figure 6.7 *A shell with regions*

Multiple View Layout

Regions are frequently used where multiple views need to be displayed in a list or a tab-style display. In these cases, regions serve to collect views that may be logically related and displayed in a list or a tab control.

The Stock Trader RI shows the use of both the template layout and the multiple-view layout. The template view layout can be seen in the shell for the application. Figure 6.8 illustrates the regions defined by the Stock Trader RI shell.

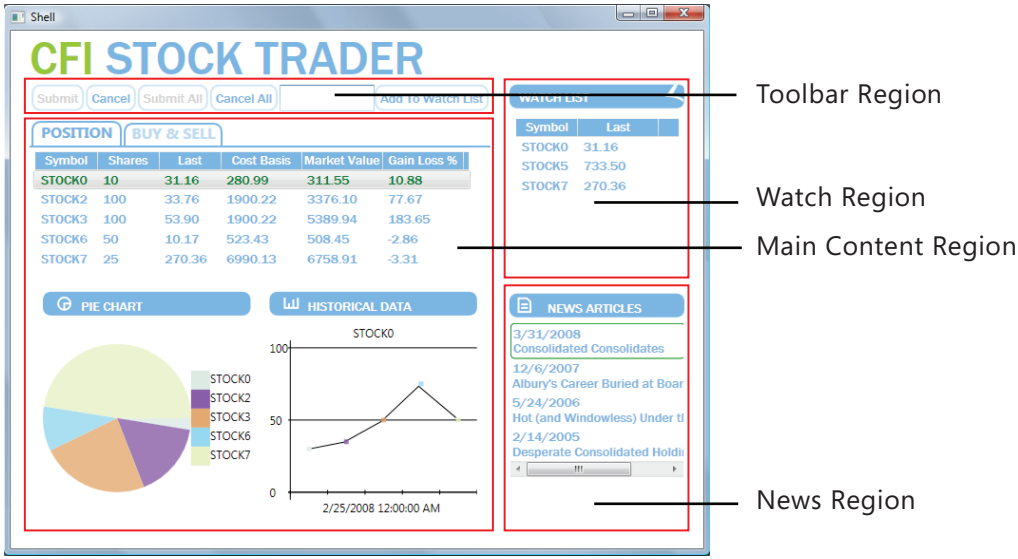


Figure 6.8 *Stock Trader RI shell regions*

The regions for the shell are defined in the Shell.xaml file and use the **RegionName** attached property from the Composite Application Library. The following code example from the Shell.xaml file shows how the **RegionManager.RegionName** attached property is used to define the regions for the Stock Trader RI.

XAML Shell.xaml

```
<ItemsControl Grid.Row="1" Grid.Column="1" x:Name="MainToolbar" cal:RegionManager_
    .RegionName="{x:Static inf:RegionNames.MainToolbarRegion}">
   <ItemsControl.ItemsPanel>
   ...
   </ItemsControl.ItemsPanel>
   </ItemsControl>
   <StackPanel Grid.Row="1" Grid.RowSpan="2" Grid.Column="3">
      <Controls:TearOffItemsControl x:Name="TearOffControl"
          cal:RegionManager.RegionName="{x:Static inf:RegionNames.WatchRegion}"
          ...
      <ItemsControl Margin="0,20,0,0" cal:RegionManager.RegionName="{x:Static _
          inf:RegionNames.NewsRegion}" />
   </StackPanel>
   <TabControl x:Name="PositionBuySellTab" Margin="0,10,0,0"
Style="{StaticResource ShellTabControlStyle}" ItemContainerStyle="{Static_
    Resource ShellTabItemStyle}" SelectedIndex="0"  Grid.Row="2" Grid.Column="1" _
    cal:RegionManager.RegionName="{x:Static inf:RegionNames.MainRegion}" />
```

A multiple layout view is demonstrated in the Stock Trader RI when buying or selling a stock. The **Buy & Sell** tab is a list-style region that shows multiple buy/sell views as part of its list, as shown in Figure 6.9.

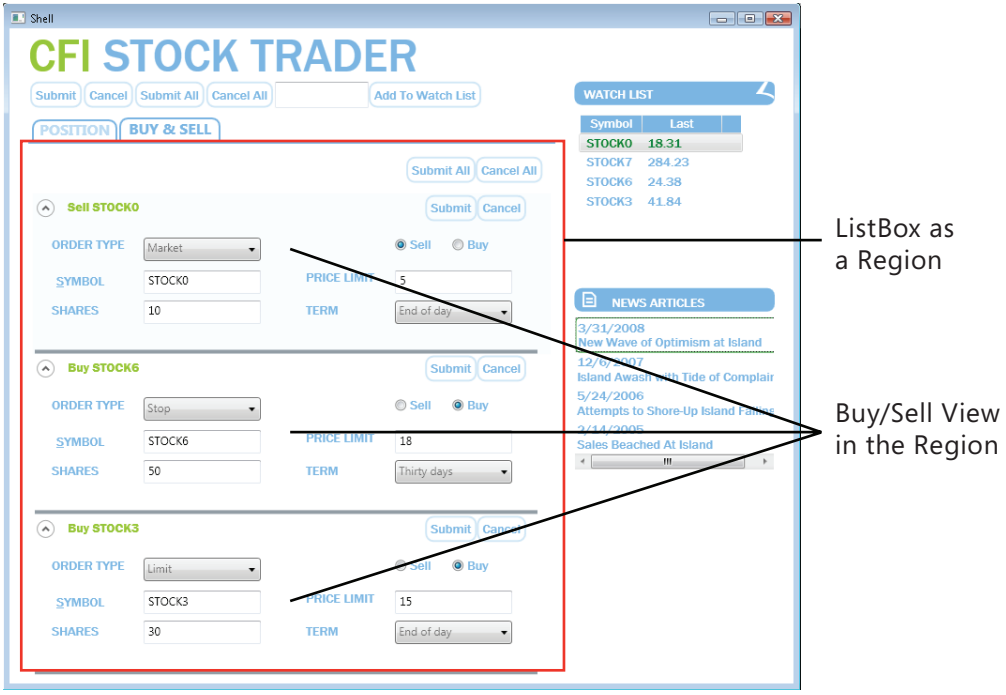


Figure 6.9 *ListBox Region*

Working with Regions

Regions are enabled in the Composite Application Library through a region manager, regions, and region adapters.

Region Manager

The **RegionManager** class is 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. Figure 6.10 illustrates the relationship between the region, control, and adapter set up by the **RegionManager**.

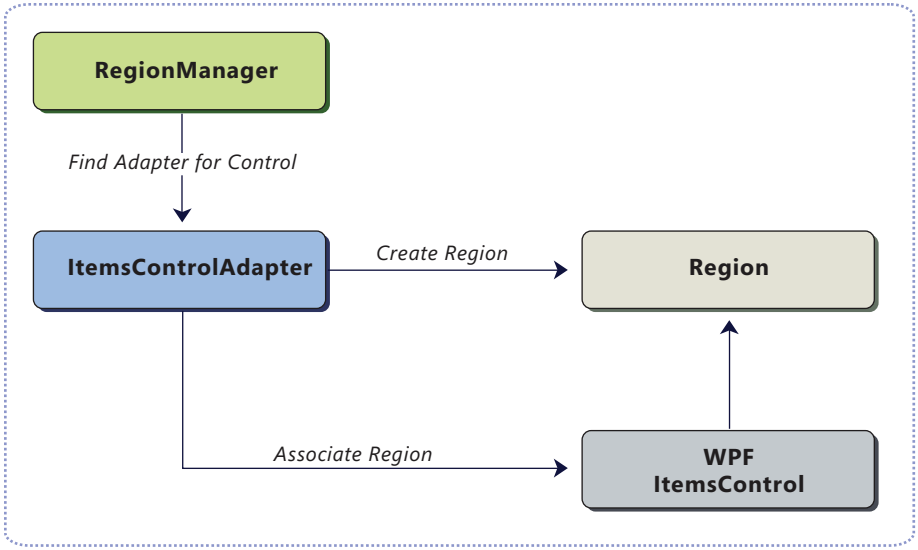


Figure 6.10 *Region, control, adapter relationship*

The **RegionManager** also supplies the attached property that can be used for simple region creation from XAML. To use the attached property, you will need to load the Composite Application Library namespace into the XAML and then use the **RegionName** attached property. The following example shows using the attached property for a window with a tab control.

XAML

```
<Window x:Class="MyApp.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:cal="http://www.codeplex.com/CompositeWPF">
...
<TabControl x:Name="TabRegion" cal:RegionManager.RegionName="MainRegion" />
...
</Window>
```

The **RegionManager** can register regions directly without using XAML. 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 following code shows registering a new region with the **RegionManager** located in a container.

C#

```
IRegionManager newRegionManager = Container.Resolve<IRegionManager>();
newRegionManager.AttachNewRegion(someControl, regionName);
```

IRegion

A region is a class that implements the **IRegion** interface. The region is the container that holds content to be displayed by the control. The following code shows the **IRegion** interface.

C# IRegion.cs

```
public interface IRegion
    {
        IViewsCollection Views { get; }
        IViewsCollection ActiveViews { get; }
        IRegionManager Add(object view);
        IRegionManager Add(object view, string viewName);
        IRegionManager Add(object view, string viewName, bool
                           createRegionManagerScope);
        void Remove(object view);
        void Activate(object view);
        void Deactivate(object view);
        object GetView(string viewName);
        IRegionManager RegionManager { get; set; }
}
```

To add a view to a region, get the region from the region manager, and call the **Add** method, as shown in the following code.

C#

```
IRegion region = _regionManager.Regions["MainRegion"];

var ordersPresentationModel = _container.Resolve<IOrdersPresentationModel>();
var _ordersView = ordersPresentationModel.View;
region.Add(_ordersView, "OrdersView");
region.Activate(_ordersView);
```

Region Adapters

Composite Application Library provides three region adapters: **ContentControl-RegionAdapter**, **SelectorRegionAdapter**, and **ItemsControlRegionAdapter**. These adapters are meant to adapt controls derived from **ContentControl**, **Selector**, and **ItemsControl**, respectively. Adapters can be replaced or new ones added for new controls, by adding to the **RegionAdapterMappings** for the **RegionManager**.

In the **UnityBootstrapper**, the **RegionAdapterMappings** is supplied to the **RegionManager** during application initialization, as shown in the following code.

C# UnityBootstrapper.cs

```
protected virtual RegionAdapterMappings ConfigureRegionAdapterMappings()
{
      RegionAdapterMappings regionAdapterMappings =
            Container.TryResolve<RegionAdapterMappings>();
       if (regionAdapterMappings != null)
       {
           regionAdapterMappings.RegisterMapping(typeof(Selector),
                                                 new SelectorRegionAdapter());
           regionAdapterMappings.RegisterMapping(typeof(ItemsControl),
                                                 new ItemsControlRegionAdapter());
           regionAdapterMappings.RegisterMapping(typeof(ContentControl),
                                                 new ContentControlRegion_
                                                     Adapter());
       }

       return regionAdapterMappings;
}
```

Scoped Regions

Views defining 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, so the second registration produces an error. Instead, use scoped regions so that each view gets its own **RegionManager** and its regions will be registered with that **RegionManager** instead of with the parent **RegionManager**, as shown in Figure 6.11 on the next page.

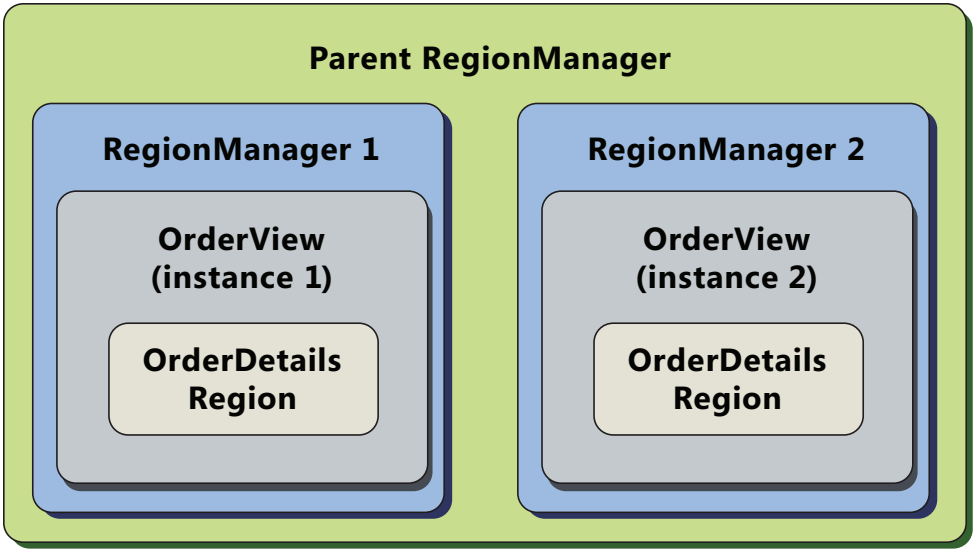


Figure 6.11 *Parent and scoped region managers*

To create a local **RegionManager** for a view, specify that a new **RegionManager** should be created when adding 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.

More Information

For more information about regions, see the following:

- "UI Composition QuickStart" on MSDN
- Region How-to topics on MSDN:
  - "How to: Add a Region"
  - "How to: Create a Custom Region Adapter"
  - "How to: Provide a Custom Logger"
  - "How to: Show a View in a Scoped Region"
  - "How to: Show a View in a Shell Region"

- Composition patterns:
  - "Composite and Composite View" in Chapter 3, "Patterns in the Composite Application Library"

Shell and View

The shell is the main window of the application where the primary user interface (UI) content is contained. The shell may be composed of multiple windows if desired, but most commonly it is just a single main window that contains multiple views. The shell may contain named regions where modules can add views. It may also define certain top-level UI elements, such as the main menu and toolbar. The shell also defines the overall appearance for the application. It may define styles and borders that are present and visible in the shell layout itself, and it may also define styles, templates, and themes that get applied to the views that are plugged into the shell.

Views are the composite portions of the user interface that are contained in the shell's window(s). It is easiest to think about a view as a user control that defines a rectangular portion of the client area in the main window. However, views in the Composite Application Library do not have to be defined with a user control. You can use a WPF data template to define a view that will be rendered based on the data in your model. A view could also share screen real estate with other views due to the rendering and compositing capabilities of WPF. In simple terms, a view is just a collection of user interface elements that define part of the rendering of the user interface. It is a unit of encapsulation for defining the separable portions of your UI.

Implementing a Shell

You do not have to have a distinct shell as part of your application architecture to use the Composite Application Library. If you are starting a new composite WPF application from the very beginning, implementing a shell provides a well-defined root and initialization pattern for setting up the main user interface of your application. However, if you are adding Composite Application 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 to add regions or 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 a shell for the content it contains.

Stock Trader RI Shell

The Stock Trader RI has a shell as its main window. In Figure 6.12, 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.

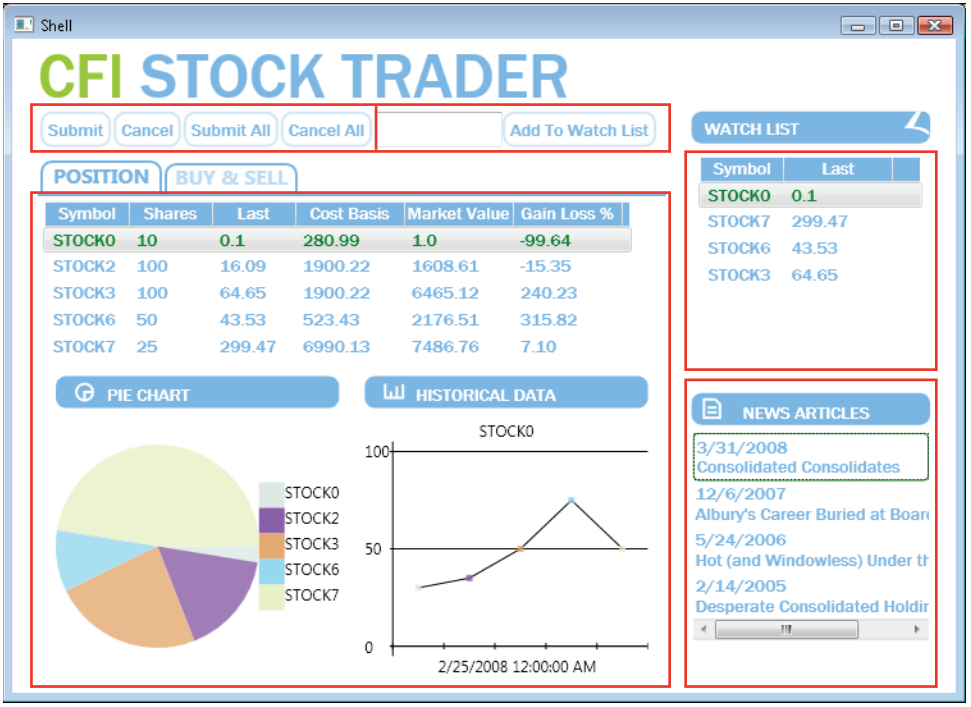


Figure 6.12 *Stock Trader RI shell window and highlighted views*

The shell implementation in the Stock Trader RI is provided by Shell.xaml and its code-behind file Shell.xaml.cs. Shell.xaml includes the layout and UI elements that are part of the shell. This includes defining several **ItemsControls** that are identified as regions for modules to add their views to, as shown here.

XAML

```
<ItemsControl Margin="0,20,0,0" cal:RegionManager.RegionName="NewsRegion" />
```

The only thing included in the code-behind file is the implementation of the **IShellView** interface defined in the Shell project, as shown here.

C# IShellView.cs

```
public interface IShellView
{
   void ShowView();
}
```

The implementation of **ShowView** simply calls **Show** on the shell class itself, which displays the main window. **ShowView** is called by the bootstrapper in the initialization of the application, as shown in the following code. For more information, see "Creating the Shell" in the Bootstrapper technical concept earlier in this chapter.

C# Shell.xaml.cs

```
public partial class Shell : Window, IShellView
{

    public Shell()
    {
        InitializeComponent();
    }

    public void ShowView()
    {
        this.Show();
    }
}
```

Implementing a View

Views are the main unit of UI construction within a composite UI application. You can define a view as a user control, data template, or even a custom control. A view encapsulates a portion of your user interface that you would like to keep as decoupled as possible from other parts of the application. You may choose what goes in a view based on only encapsulation or a piece of functionality, or you may 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 Composite Application Library required to define a view. The easiest and most common 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 that. The only thing the Composite Application Library adds is the ability to define a region into which a view can be dynamically added at run time.

The following markup shows an example user control.

XAML

```
<UserControl x:Class="WpfApplication1.ExcessivelySimpleView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="25" Width="100">
    <Label>A VERY simple view</Label>
</UserControl>
```

The following markup shows an example data template.

XAML

```
<Window x:Class="WpfApplication1.Window1" .../>
    <Window.Resources>
        <DataTemplate x:Key="ADataTemplateView">
            <Label>A really simple data template view</Label>
        </DataTemplate>
    </Window.Resources>
    <Grid .../>
</Window>
```

Composite Views

Frequently, a view that supports a specific set of functionality can get complicated. In that case, it may make sense to break up the view into several child views and have the parent view handle constructing itself using the child views as parts. It may do this statically at design time, or it may 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. Frequently, a composite view is responsible for constructing the child views and for coordinating the interactions between them. Child views can also be designed to be more loosely coupled from their sibling views and their parent composite view by using the Composite Application Library commands and the event aggregator.

Views in the Stock Trader RI

Figure 6.12 earlier in this section highlights five different views. In what the user sees as a main toolbar, there are really two views plugged in by modules that are composed of buttons and other toolbar controls. The main content area of the Stock Trader RI is a composite view with three child views, the position grid, and two chart control instances. The watch list and news articles are also individual views. The view illustrated in Figure 6.12 does not display buy/sell orders, which are defined as composite views themselves, and they are hosted in an **ItemsControl**-based region on another tab in the main content area.

Views and Design Patterns

You should consider using one of several user interface design patterns when implementing a view even though it is not required by the Composite Application Library. The Stock Trader RI and QuickStarts demonstrate both the Supervising Controller and Presentation Model patterns as a way to implement a clean separation between the view layout and the view logic.

Separating the logic from the view is important for both testability and maintainability. If you create a view with a user control or custom control and put all the logic in the code behind, it can be difficult to test because you have to create an instance of the view to unit test the logic. However, sometimes the base class of the view can interfere with testing because it expects only the class to be created as part of a user interface in a normal WPF execution context. To make sure the right things happen in the view as part of your unit tests, you will also need a way to mock out the views, which really requires a separate class for the view and the logic. If you define a view as a data template, there is no code associated with the view itself, so you have to put the associated logic somewhere else. The same clean separation of logic from layout required for testability also helps make the view easier to maintain.

More Information

For more information about patterns that separate view layout and view logic, see "Supervising Controller" and "Presentation Model" in Chapter 3, "Patterns in the Composite Application Library."

For more information about views, see the following topics on MSDN:

- "How to: Create a View with a Presenter." This topic describes how to create a view following the Model-View-Presenter pattern.
- "How to: Show a View in a Shell Region." This topic describes how to place a view in a shell-defined region.
- "How to: Show a View in a Scoped Region." This topic describes how to create scoped regions and show views in scoped regions.

Event Aggregator

The **EventAggregator** service is primarily a container for events that allow decoupling of publishers and subscribers 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, more likely, other modules.

In the Composite Application Library, **EventAggregator** allows subscribers or publishers to locate a specific **EventBase**. The event aggregator also allows for multiple publishers and multiple subscribers, as shown in Figure 6.13.

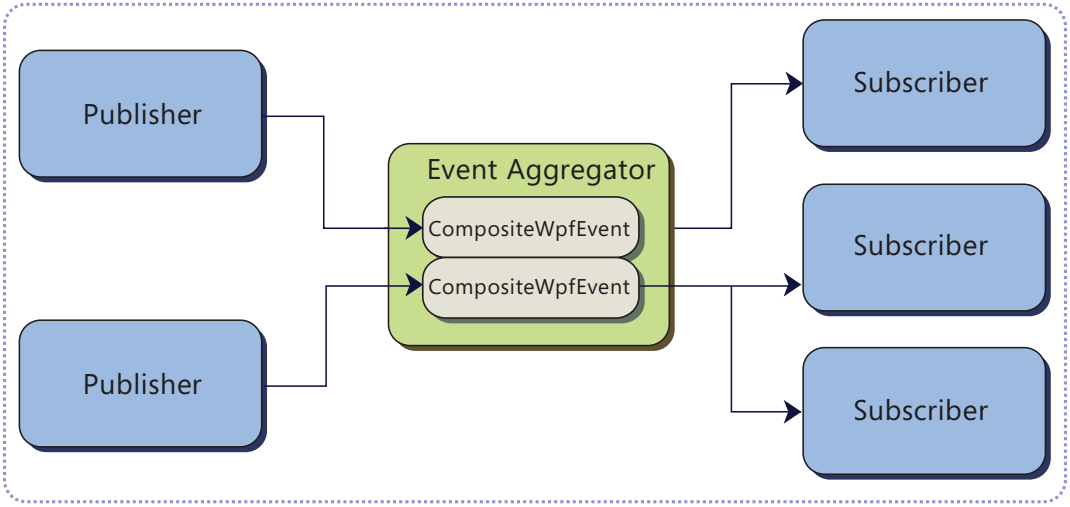


Figure 6.13 *Event aggregator*

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# IEventAggregator.cs

```
public interface IEventAggregator
{
   TEventType GetEvent<TEventType>() where TEventType : EventBase;
}
```

The **EventAggregator** will construct the event on its first access if it has not already been constructed. This relieves the publisher or subscriber from having to determine whether the event is available.

CompositeWpfEvent

The real work of connecting publishers and subscribers is done by the **CompositeWpfEvent** class. This is the only implementation of the **EventBase** class in the Composite Application Library. This class maintains the list of subscribers and handles event dispatching to the subscribers.

The **CompositeWpfEvent** 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 **CompositeWpfEvent** class.

C# CompositeWpfEvent.cs

```
public class CompositeWpfEvent<TPayload> : EventBase
{
    ...
    public SubscriptionToken Subscribe(Action<TPayload> action);
    public SubscriptionToken Subscribe(Action<TPayload> action,
           ThreadOption threadOption);
    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 void Unsubscribe(SubscriptionToken token);
    ...
}
```

The **CompositeWpfEvent** class is intended to be the base class for an application's or module's specific events. For example, the following code shows the **TickerSymbolSelectedEvent** in the Stock Trader RI.

C# TickerSymbolSelectedEvent.cs

```
public class TickerSymbolSelectedEvent : CompositeWpfEvent<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.

Subscribing to an Event

Subscribers can enlist with an event using one of the **CompositeWpfEvent** available **Subscribe** method overloads. There are a number of options for subscribing to **CompositeWpfEvents**. Use the following criteria to help determine which option best suits your needs:

- If you need to be able to update user-interface elements when an event is received, subscribe to receive the event on the user interface thread.
- If you need to filter an event, provide a filter delegate when subscribing.
- If you have noticed performance concerns with your events, consider using strongly referenced delegates when subscribing and manually unsubscribing from the **CompositeWpfEvent**.
- If none of the preceding is applicable, use a default subscription.

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 take a string parameter, as shown here.

C#

```
public void Run()
{
   eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews);
}

public void ShowNews(string companySymbol)
{
   articlePresentationModel.SetTickerSymbol(companySymbol);
}
```

Subscribing on the User Interface Thread

Frequently, subscribers will need to update user interface elements in response to events. In WPF, only a UI thread can update user interface elements. By default, the subscriber receives the event on the publisher's thread, so if the publisher sends the event from the UI thread, the subscriber will be able to update the user interface.

However, if the publisher's thread is a background thread, the subscriber may be unable to directly update user interface elements. Instead, it would need to schedule the updates on the UI thread using the WPF **Dispatcher** class. The **CompositeWpfEvent** provided with the Composite Application Library can assist by allowing the subscriber to automatically receive the event on the UI thread. The subscriber must indicate this during subscription, as shown in the following code.

C# NewsController.cs

```
public void Run()
{
   eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews,
                                                          ThreadOption.UIThread);
}

public void ShowNews(string companySymbol)
{
   articlePresentationModel.SetTickerSymbol(companySymbol);
}
```

The following options are available for **ThreadOption**:

- **Publisher.** Use this setting to receive the event on the publishers' thread. This is the default setting.
- **Background.** Use this setting to receive the event on a .NET Framework thread-pool thread.
- **UIThread.** Use this setting to receive the event on the user interface thread.

Subscription Filtering

Subscribers may not need to handle every instance of a published event. In these cases, the subscriber can subscribe and supply a delegate that filters the event before the registered handler is called. Frequently, this filter is supplied as a lambda expression, as shown in the following code.

C#

```
FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>();

fundAddedEvent.Subscribe(FundAddedEventHandler,
                         ThreadOption.UIThread, false,
                         fundOrder => fundOrder.CustomerId == _customerId);
```

Subscribing Using Strong References

If you have noticed performance concerns with your events, you may need to subscribe with strong delegate references—and therefore, manually unsubscribe from the event—instead of the default weak delegate references maintained by **CompositeWpfEvent**.

By default, **CompositeWpfEvent** maintains a weak delegate reference to the subscriber's handler and filter on subscription. This means the reference that **CompositeWpfEvent** holds to the subscriber will not prevent garbage collection of the subscriber. Using a weak delegate reference relieves the subscriber from the need to unsubscribe to enable proper garbage collection. However, maintaining this weak delegate reference is slower than a corresponding strong delegate 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 delegate references with **CompositeWpfEvent** to achieve reasonable performance. If you do use strong delegate references, your subscriber should unsubscribe to enable proper garbage collection of your subscribing object.

To subscribe with a strong reference, use the **keepSubscriberReferenceAlive** option on the **Subscribe** method, as shown in the following code.

C#

```
FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>();

bool keepSubscriberReferenceAlive = true;

fundAddedEvent.Subscribe(FundAddedEventHandler,
                         ThreadOption.UIThread, keepSubscriberReferenceAlive,
                         fundOrder => fundOrder.CustomerId == _customerId);
```

Publishing an Event

Publishers raise an event by retrieving the event from the **EventAggregator** and calling the **Publish** method. For example, the following code in the Stock Trader RI demonstrates publishing the **TickerSymbolSelectedEvent**.

C# PositionSummaryPresentationModel.cs

```
EventAggregator.GetEvent<TickerSymbolSelectedEvent>().Publish(e.Value);
```

Unsubscribing from an Event

If your subscriber no longer wants to receive events, you can directly unsubscribe using your subscriber's handler or you can unsubscribe by using a subscription token. The following example shows how to directly unsubscribe to the handler.

C#

```
compositeWpfEvent.Subscribe(
    FundAddedEventHandler,
    ThreadOption.PublisherThread);

compositeWpfEvent.Unsubscribe(FundAddedEventHandler);
```

To unsubscribe with a subscription token, the token supplied during the subscribe process can be supplied to the **Unsubscribe** method call, as shown here.

C#

```
FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>();

subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler,
                                  ThreadOption.UIThread, false,
                                  fundOrder => fundOrder.CustomerId == _
customerId);

fundAddedEvent.Unsubscribe(subscriptionToken);
```

More Information

For more information about events in the Composite Application Library, see the following resources:

- "Event Aggregation QuickStart" on MSDN
- "How to: Create and Publish Events" on MSDN
- "How to: Subscribe and Unsubscribe to Events" on MSDN
- "Event Aggregator" on Martin Fowler's Web site

Commands

Commands are a way to handle user interface (UI) actions. They are a loosely coupled way to bind the UI to the logic that performs the action.

When building composite applications, presentation design patterns such as Model-View-Presenter (MVP) and Model-View-Controller (MVC) are often used to separate the UI logic from the UI layout and presentation. When implementing these patterns with WPF, the presenter or controller handles commands but lives outside the logical

tree. WPF-routed commands deliver command messages through UI elements in the 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. Additionally, the WPF-routed commands require a command handler in the code behind.

The Composite Application Library introduces two new commands that can be routed outside the boundaries of the logical tree and that do not require handling logic in the code behind. The commands are custom implementations of the **ICommand** interface defined by WPF, and they implement their own routing mechanism to get the command messages delivered to objects outside of the logical tree. The commands in the Composite Application Library include **DelegateCommand** and **CompositeCommand**.

DelegateCommand<T>

The **DelegateCommand** allows delegating the commanding logic instead of requiring a handler in the code behind. It uses a delegate as the method of invoking a target handling method.

In the Stock Trader RI, the **Buy** and **Sell** shortcut menu items that appear when the user right-clicks a specific symbol on the positions screen is a delegate command. These commands are handled by the **OrdersController**. **DelegateCommands** can also be handled by a either a ViewModel or a presenter, depending on the scenario.

The **DelegateCommand** uses its delegate to invoke a **CanExecute** method or **Execute** method on the target object when the command is invoked. Because the class is generic, it enforces compile-time checking on the command parameters, which traditional WPF commands do not. Additionally, because it accepts delegates, it removes the need for creating a new command type for every instance where you need commanding.

DelegateCommand accepts two constructor parameters, **executeMethod** and **canExecuteMethod**. Because the parameter types are generic delegates, the handlers can be easily hooked into the underlying controllers or presenters. The following code shows how the **DelegateCommand** holds onto the provided delegates that it uses to invoke the target methods.

C# DelegateCommand.cs

```
public class DelegateCommand<T> : ICommand
{
    public DelegateCommand(Action<T> executeMethod, Func<T, bool> canExecute_
        Method)
    {
        ...
        this.executeMethod = executeMethod;
        this.canExecuteMethod = canExecuteMethod;
        ...
    }
...
}
```

IActiveAware Interface

In some instances, you may want a delegate command to execute only if it is in a view that is currently active (selected). For example, the **Submit** command (which the **Submit** button binds to) in each Buy/Sell order is active only when that order is selected.

To support this behavior, the **DelegateCommand** implements **IActiveAware**. This interface has an **IsActive** property, which can be set whenever the command becomes active. Whenever the property changes, the **DelegateCommand** raises the **IsActiveChanged** event.

The following code shows the implementation of the **IActiveAware** interface on the **DelegateCommand**.

C# DelegateCommand.cs

```
public event EventHandler IsActiveChanged;

public bool IsActive
{
    get { return _isActive; }
    set
    {
        if (_isActive != value)
        {
            _isActive = value;
            OnIsActiveChanged();
        }
    }
}

protected virtual void OnIsActiveChanged()
{
    EventHandler isActiveChangedHandler = IsActiveChanged;
    if (isActiveChangedHandler != null) isActiveChangedHandler(this, EventArgs._
        Empty);
}
```

CompositeCommand

The **CompositeCommand** is a command that has multiple child commands. The **CompositeCommand** is used in the Stock Trader RI in two cases: first, for the Buy/Sell **Submit All** button on the main toolbar and second, for the **Submit** and **Cancel** buttons on that toolbar. When you click the **Submit All** button, all the child **Submit** commands in the Buy/Sell screens are executed. When you click the **Submit** or **Cancel** button, only the active Buy/Sell order's command is executed.

When you call the **Execute** or the **CanExecute** method on the composite command, it calls the respective method on the child commands. The following code shows how the **Execute** and **CanExecute** methods are implemented in the **CompositeCommand** class. The **ShouldExecute** method is described in the next section.

C# CompositeCommand.cs

```
public virtual bool CanExecute(object parameter)
{
    bool hasEnabledCommandsThatShouldBeExecuted = false;

    foreach (ICommand command in registeredCommands)
    {
        if (ShouldExecute(command))
        {
            if (!command.CanExecute(parameter))
            {
                return false;
            }

            hasEnabledCommandsThatShouldBeExecuted = true;
        }
    }
    return hasEnabledCommandsThatShouldBeExecuted;
}

public virtual void Execute(object parameter)
{
    Queue<ICommand> commands = new Queue<ICommand>(registeredCommands);

    while (commands.Count > 0)
    {
        ICommand command = commands.Dequeue();
        if (ShouldExecute(command))
            command.Execute(parameter);
    }
}
```

Figure 6.14 on the next page illustrates a diagram of the connection between **DelegateCommands** and **CompositeCommands**.

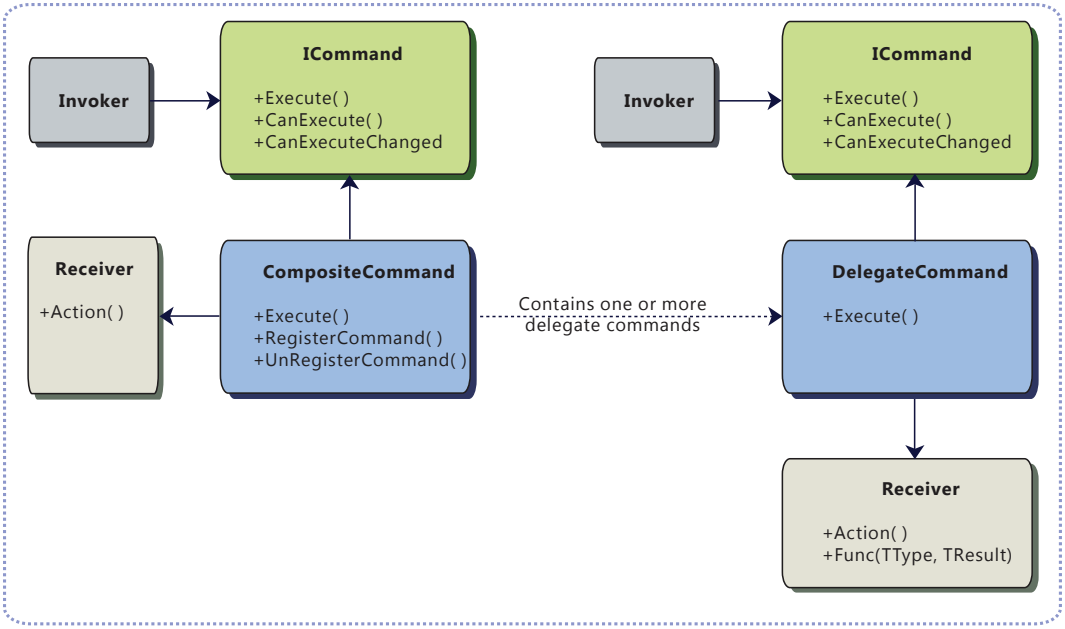


Figure 6.14 *Connection between CompositeCommand and DelegateCommand*

In some instances, you may want a composite command to execute only if it is in a view that is currently active (selected). To support this behavior, the **DelegateCommand** implements **IActiveAware**. This interface has an **IsActive** property, which can be set whenever the command becomes active. Whenever the property changes, the **DelegateCommand** throws the **IsActiveChanged** event.

In the following code, you can see the implementation of the **IActiveAware** interface on the **DelegateCommand**.

Activity Monitoring Behavior

In some cases, you may want the composite command to execute only the active command. For example, in the Stock Trader RI, the **Submit** composite command, which is invoked by the **Submit** button on the toolbar, submits only the current order. That is, it executes only the current order's **Submit** command.

Figure 6.15 illustrates the **Submit** button on the main toolbar in the Stock Trader RI.

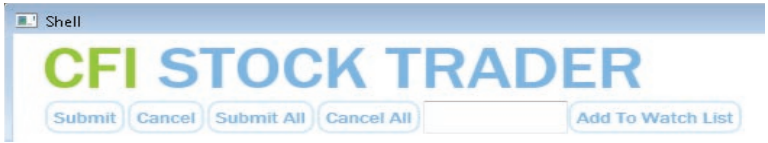


Figure 6.15 *Stock Trader RI main toolbar*

To support the submit functionality, the **CompositeCommand** has an activity monitoring behavior. This behavior is enabled in the **CompositeCommand**'s constructor by setting the **monitorCommandActivity** to **true**. When this behavior is enabled, the **CompositeCommand** performs an additional check on commands that implement **IActiveAware** before it executes them. If that command's **IsActive** property is set to **true** and the command's **CanExecute** method returns **true**, it executes. The **ShouldExecute** method handles this logic, as shown here.

C# CompositeCommand.cs

```
protected virtual bool ShouldExecute(ICommand command)
{
    var activeAwareCommand = command as IActiveAware;

    if (monitorCommandActivity && activeAwareCommand != null)
    {
        return (activeAwareCommand.IsActive);
    }

    return true;
}
```

Registering and Unregistering Composite Commands

Composite commands are either registered or unregistered through the **RegisterCommand** and **UnregisterCommand**. In the Stock Trader RI, the **OrdersController** is responsible for registering and unregistering the child order commands. The following code shows where the **Submit** and **Cancel** commands are registered in the **StartOrder** method.

C# OrdersController.cs

```
virtual protected void StartOrder(...)
{
...
     commandProxy.SubmitOrderCommand.RegisterCommand (orderCompositePresentation_
         Model.SubmitCommand);
     commandProxy.CancelOrderCommand.RegisterCommand
(orderCompositePresentationModel.CancelCommand);
...
}
```

Frequently Asked Questions About Commands

Why are WPF commands not used?

WPF commands have a number of limitations. They are coupled to elements in the logical tree because they use routed events under the covers to deliver the command messages. This means you cannot directly hook up a separate class, such as a presentation model, presenter, or controller, to be the direct command handler. The view would have to be the routed command handler, and it would have to forward the call to the presenter or controller through a method call or event. Additionally, the command handler that the routed event is delivered to is determined by the current focus in the UI. This works fine if the command handler is at the window level, because the window is always in the focus tree of the currently focused element, so it gets called for command messages. However, it does not work for child views who have their own command handlers unless they have the focus at the time. Finally, only one command handler is ever consulted with routed commands. After one command handler is invoked (for **CanExecute** or **Execute**), no other handlers are consulted, even if they are in the focus tree of the focused element. For scenarios where multiple views (or their presenters) need to handle the command, there is no decoupled way to address it with routed commands.

Can delegate commands be replaced with routed commands?

No, because both are meant for two different purposes. Routed commands, such as **Cut**, **Copy**, and **Paste**, are meant for controls with command binding that live within the logical tree and that will have the focus for the intent of the command. They can also be used for general purposes if it is acceptable to put centralized command handling at the root window or page element and have it as part of the view. However, that approach does not scale for composite applications, so the **DelegateCommand** approach allows you to have the flexibility of multiple command handlers that live outside the logical tree.

Can the order of execution of commands be set up inside the Composite commands?

Currently, you cannot specify the order that commands are executed within **Composite** commands. Moreover, this is not required from a high level, because command handlers should be decoupled from one another and not rely on a specific invocation order. The workaround for this is the judicious usage of **DelegateCommands** and the implementation logic. You just need to know how the composite command works. Because the composite command orders the command list internally, execution mode will be First In, First Out (FIFO).

More Information

For more information about commands, see the following topics on MSDN:

- "Commanding QuickStart"
- "How to: Create Globally Available Commands"
- "How to: Create Locally Available Commands"

Communication

When building large complex 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. This allows the modules to be independently developed, tested, deployed, and updated, and it forces loosely coupled communication.

When communicating between modules, you can use commanding, event aggregation, or shared services. Use the following to help decide which approach to use:

- **Commanding.** Use this in response to user gestures and custom enablement.
- **Event aggregator.** Use this to publish an event across modules.
- **Shared services.** Use this if neither of the preceding is applicable.

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 you want the invoker to be enabled based on business logic, use commanding**.**

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 controller that does not have any associated elements in the visual tree or is not the focused element. To support this scenario, the Composite Application Library provides **CompositeCommand** and **DelegateCommand**, which has a direct routing mechanism, compared to **RoutedCommand**, which uses tunneling and bubbling.

The **CompositeCommand** is an implementation of **ICommand** so that it can be bound to invokers. **CompositeCommands** can be connected to several child commands and when the **CompositeCommand** is invoked, the child commands are also invoked.

CompositeCommands support the notion of 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 with cross module communication? Applications based on the Composite Application Library may have global **CompositeCommands** that are defined in the shell that have meaning across modules, such as **Save**, **Save All**, **Cancel**. Modules can then register their local commands with these global commands and participate in their execution.

For more information about using composite commands, see "Commands" earlier in this chapter.

Event Aggregator

If you need to publish an event across modules and do not need a response, use **EventAggregator**.

Consider using **EventAggregator** when sending a message between business logic code, such as controllers and presenters. An example is when the **Process Order** button has been clicked and the order successfully processed; in this case, other modules need to know the order is successfully processed so they can update their views.

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.

This helps with cross module communication because events can be defined in a shared assembly in a way that publishers and subscribes can reside in entirely separate modules.

For more information about using **EventAggregator**, see "Event Aggregator" earlier in this chapter.

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 RI, the News module provides an implementation of **INewsFeedService**. The Position module consumes these services by using the shell application's dependency injection container, which provides service location and resolution. The **INewsFeedService** is meant to be consumed by other modules, so it can be found in the **StockTraderRI.Infrastructure** common assembly.

To see how these modules register their services into the Unity dependency injection container, see the files NewsModule.cs, as shown in the following code, and MarketModule.cs. The Position module's **PositionSummaryPresentationModel** receives these services through constructor dependency injection. For more information about the Dependency Injection pattern, see "Dependency Injection Pattern" in Chapter 3, "Patterns in the Composite Application Library."

C# NewsModule.cs

```
protected void RegisterViewsAndServices()
{
   _container.RegisterType<INewsFeedService, NewsFeedService>(new Container_
       ControlledLifetimeManager());
}
```

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.

For more information about services and containers, see "Container and Services" earlier in this chapter.

7

Designer Guidance

Overview

The approach to building applications with the Composite Application Library can vary widely from implementation to implementation. Designing a user interface (UI) can become confusing without clear guidance. The majority of the confusion stems from the difference between the design-time experience of monolithic applications and the design-time experience of composite applications.

A monolithic application is an application where all its pieces are combined into a single application. It is designed without modularity—this means the parts are very tightly coupled and cannot be separated or replaced. From a user interface (UI) perspective, the individual screens in these applications are often laid out at design time and statically composed at run time. This type of application is simpler to design because the majority of the UI artifacts are placed on screens during design time and the overall appearance of the application is easy to see.

A composite application is an application where the UI is dynamically composed at run time. This allows business logic to determine what is shown to the user. It also allows you to add new functionality to the UI with less friction. In these types of applications, only portions of screens can be seen at design time, and the design experience is more focused on creating islands of visual representation that are combined to represent an overall screen. It is the dynamic nature of these types of applications that presents the most confusion to designers because, in many cases, the complete view of the application can be seen only by actually running the application.

The goal of this section is to provide some high-level guidance to designers when working on a team that is building an application with the Composite Application Library. This section describes UI layout, visual representation techniques, binding, and resources distribution. After reading this section, you should have a high-level understanding of how to approach designing the UI of an application based on the Composite Application Library and some of the techniques that can help you create a maintainable UI in composite applications.

Layout

The layout of applications created with the Composite Application Library builds on the standard principals of Windows Presentation Foundation (WPF)—the layout uses the concepts of panels that contain related items. One of the differences in the Composite Application Library is that the content inside the various panels is dynamic and is not known during design time. This turns the design experience upside-down by forcing designers to create page structures that can contain various content and then work on the actual content in another context. As a designer, this forces you to think about two main layout concepts in the Composite Application Library: container composition and regions.

Container Composition

The term "container composition" is really just an extension to the containment model that WPF inherently provides. This term is used here because of the extremely flexible containment model that WPF has. The term "container" can mean any element, including a window, page, user 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 reoccurring themes that stand out. You will be creating either a window, page, or user controls that will contain 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 more information about regions, see the next section). For example, the Stock Trader Reference Implementation (Stock Trader RI) application has a startup window (named Shell.xaml) that contains the overall structure for the application. The window in Microsoft Expression Blend, as shown in Figure 7.1, shows a fixed portion of the window. The remaining sections of the window 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 during run time is something that you need to design around. To see an example of this, compare the designer view of the main page in Figure 7.1 to the run-time view in Figure 7.2. In the designer view, the page is mostly empty; it has a title and a section header titled Watch List. Contrast that with the run-time view, where there is a toolbar area with a series of buttons, a

position area that contains a tab control with position data, and a News and Watch List area that contain stocks to watch and news pertaining to the selected stocks. The differences between the designer view and run-time view demonstrate the challenges designers face when they create applications built with the Composite Application Library. The items cannot be seen during design time, so 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 in situations where a lot of dynamic content needs to fit into a confined area.
- Pay close attention to how content enlarges as the screen content grows to ensure the appearance of your application is appealing in any resolution.

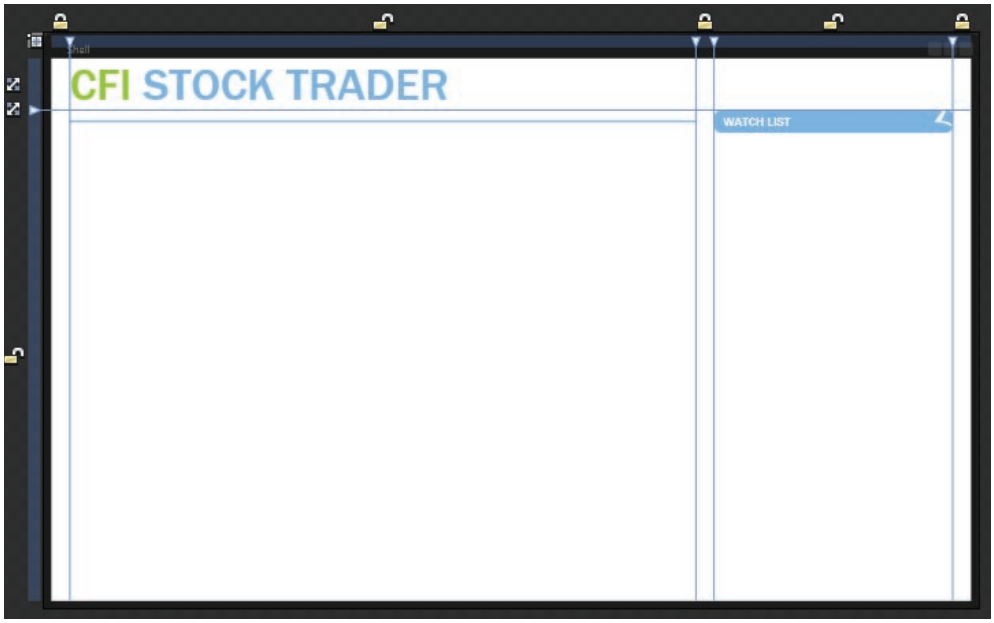


Figure 7.1 *Stock Trader RI main window in Microsoft Expression Blend*

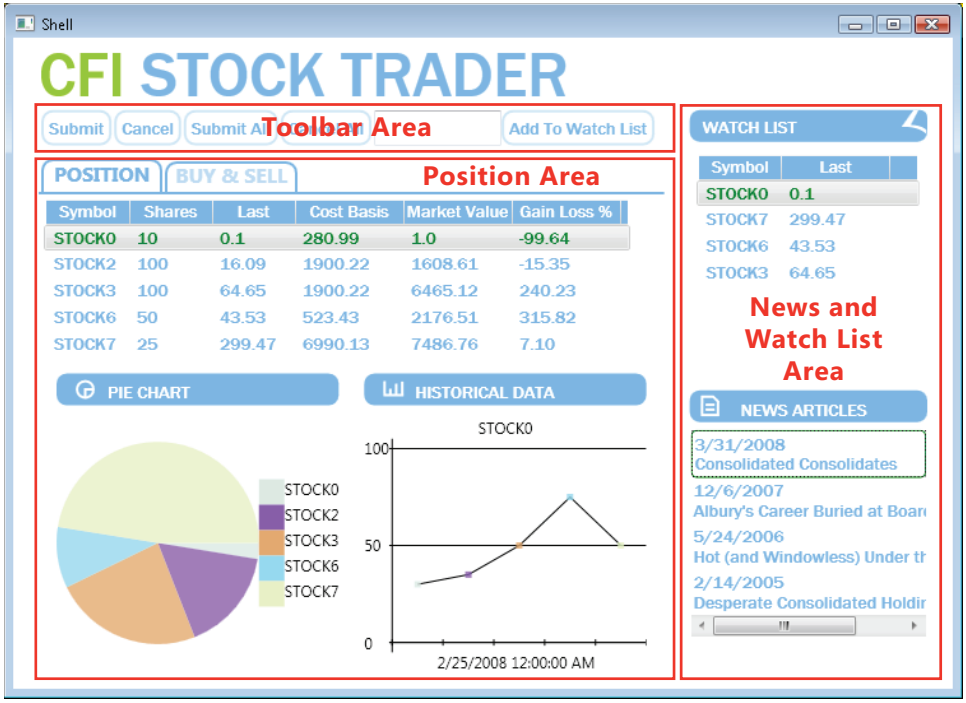


Figure 7.2 *Stock Trader RI main window during run time*

Region

The term "region" represents a container that can hold dynamic data that is manifested into a UI representation. A region allows the Composite Application Library to place dynamic content contained in modules in predefined placeholders in a UI container. The power of regions comes in their ability to hold any type of UI content. A module can contain UI content manifested as a user control, a data type that is associated with a data template, a custom control, or any combination of these. Essentially, this gives a designer the ability to create an appearance and behavior for a UI area and then to have modules place content in these predetermined areas. Figure 7.3 illustrates that the Stock Trader RI main window contains four regions: MainToolbarRegion, MainRegion, WatchListRegion, and NewsRegion. These regions are populated by the various modules in the application—the content can be changed at any time.

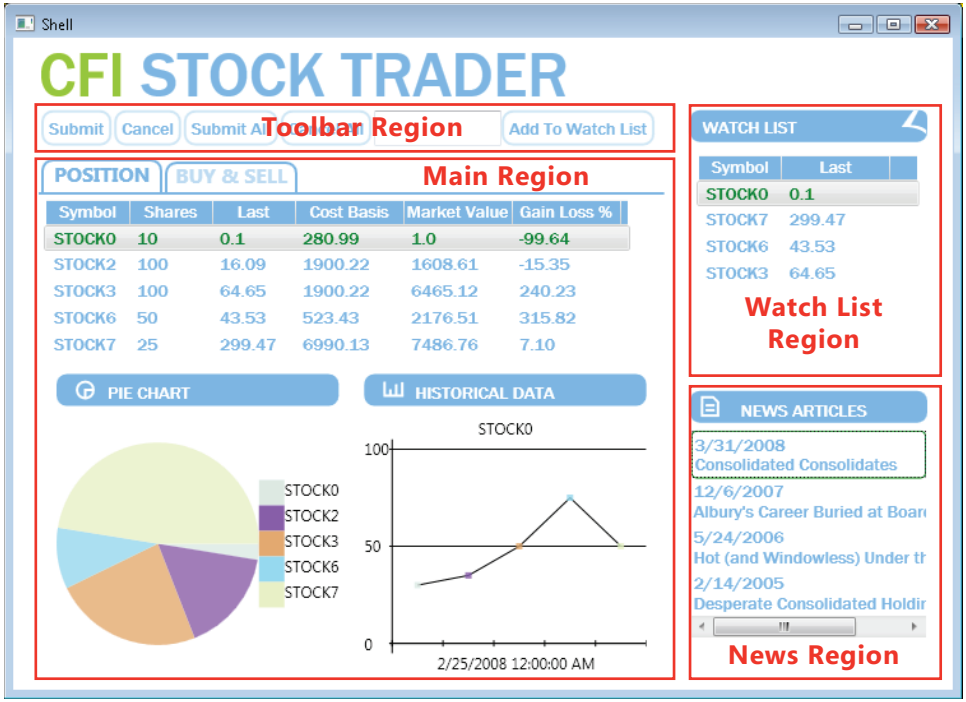


Figure 7.3 *Stock Trader RI regions*

To create a region, you need to add the **RegionManager.RegionName** property in XAML to a **ContentControl**-based UI element or an **ItemsControl**-based UI element that has built-in support in the Composite Application Library. The value of the name can be anything and will be used by the module developer to determine where content will be placed. The XAML in the following code shows how the four regions are assigned in the Stock Trader RI. If these two types of containers do not provide what you need, the Composite Application Library provides the ability to create region adaptors that can be created to work with any type of UI element. For more information, see "How to: Create a Custom Region Adapter" on MSDN.

XAML Shell.xaml

```
<Window x:Class="StockTraderRI.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:cal="http://www.codeplex.com/CompositeWPF"
    xmlns:inf="clr-namespace:StockTraderRI.Infrastructure;assembly=StockTraderRI._
        Infrastructure"
    xmlns:Controls="clr-namespace:StockTraderRI.Controls"
    Title="Shell" Height="630" Width="1024" WindowStartupLocation="CenterScreen">
    <Window.Resources>
    ...
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
        ...
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
        ...
        </Grid.RowDefinitions>

        <TextBlock .../></TextBlock>
        <ItemsControl Grid.Row="1" Grid.Column="1" x:Name="MainToolbar" _
            cal:RegionManager.RegionName="{x:Static inf:RegionNames.MainToolbar_
            Region}">
            <ItemsControl.ItemsPanel>
            ...
            </ItemsControl.ItemsPanel>
        </ItemsControl>
        <StackPanel Grid.Row="1" Grid.RowSpan="2" Grid.Column="3">
            <Controls:TearOffItemsControl x:Name="TearOffControl"
                cal:RegionManager.RegionName="{x:Static inf:RegionNames._
                    WatchRegion}"
                                          HeaderBackground="#FF77B6EB"
                                          HeaderButtonBackground="#FF77B6EB"
                                          HeaderButtonRollOverBackground=_
                                              "#7F77B6EB"
                                          HeaderText="WATCH LIST"
                                          HeaderTextStyle="{StaticResource
                                              TearOffControlHeaderTextStyle}"
                                          WindowHeight="400" WindowWidth="300" />
          <ItemsControl Margin="0,20,0,0" cal:RegionManager.RegionName="{x:Static
              inf:RegionNames.NewsRegion}" />
        </StackPanel>
        <TabControl x:Name="PositionBuySellTab" Margin="0,10,0,0"
            Style="{StaticResource ShellTabControlStyle}" ItemContainerStyle=_
            "{StaticResource ShellTabItemStyle}" SelectedIndex="0"  Grid.Row="2"
            Grid.Column="1" cal:RegionManager.RegionName="{x:Static inf:Region_
            Names.MainRegion}" />
    </Grid>
</Window>
```

To demonstrate how modules and their content are associated with regions, see Figure 7.4, which shows the association of **WatchListModule** and the **NewsModule** with their corresponding regions on the main window. The **WatchRegion** contains the **WatchListView.xaml** user control, which is contained in the **WatchListModule**, and the **NewsRegion** contains the **ArticleView.xaml** user control, which is contained in the **NewsModule**. In applications created with the Composite Application Library, mappings like this will be a part of the design process as designers and developers use to determine what content is proposed to be in a particular region. This allows designers to determine the overall space needed and any additional items that need to be added to ensure the content will be viewable in the allowable space.

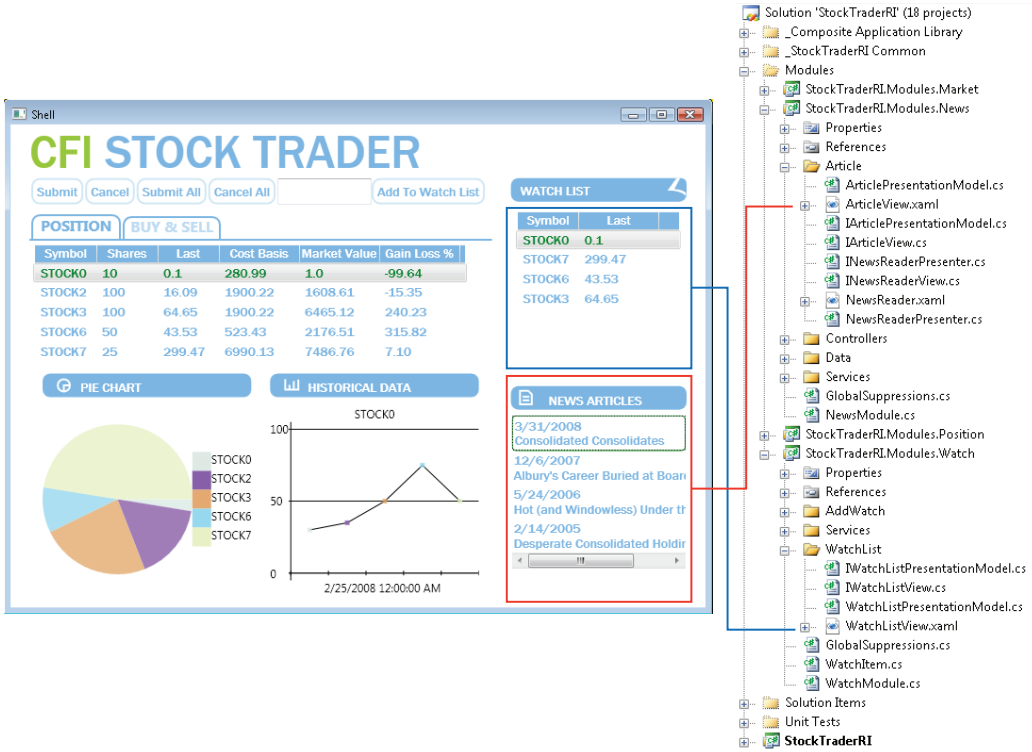


Figure 7.4 *User control to region mapping*

Visual Representation

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 predominately used to represent distinct sections on the main window, but this is not a standard. In your application, you should use an approach that is most familiar to you and fits into how you work as a designer. Regardless of the predominate visual representation in your application, you will inevitably use a combination of user controls, custom controls, and data templates in your overall design. Figure 7.5 shows where the Stock Trader RI used these various items. This illustration also serves as a reference for the following sections, which describe each of the items.

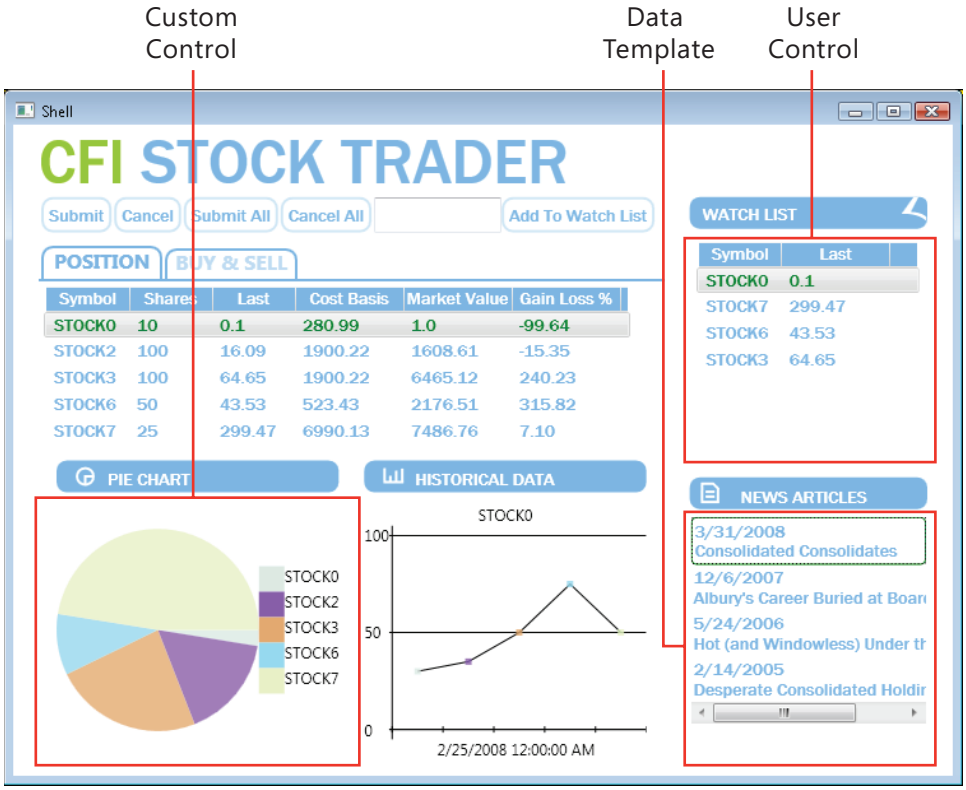


Figure 7.5 *Stock Trader RI usage of user controls, custom controls, and data templates*

User Controls

The rich support in Blend for creating user controls makes them an easy choice for creating UI content in the Composite Application Library. As mentioned earlier in this section, the Stock Trader RI uses them extensively to create content that will be inserted into regions. The **WatchList.xaml** user control is a good example of a simple UI representation that is contained inside the **WatchListModule**. 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 your overall portfolio. This type of control is a little more challenging than a user control to create, and it has limited support in Blend.

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, complete visual representations can be created using a data template alone, without the need to create any type of control. The News Article region uses a data template to show articles and, in conjunction with an **Items** style, provides an indication of which item was selected.

Data Binding

The design experience for data binding will mirror that of most dynamic data-driven applications where the data context is not assigned until run time. Of course, this limits some of the designer-specific data binding features of Blend, but it is a reality in most applications of this type.

In most cases, the view will have its data context assigned by the presenter or presentation model, which will expose data and delegate commands that are available for binding during design time. Whether binding to data or commands, a custom path expression must be used in the Create Data Binding wizard in Blend. In the case of binding to data, Figure 7.6 shows an example of binding to the **Articles** property, which is a collection, to the **ItemsSource** property of the **ListBox** contained in the ArticleView.xaml user control.

Note: The following defines some terms as they are used here:

View. A view can consist of a window, page, or user control that represents your visual representation. This could also refer to a data template in some cases.

Presenter. A presenter is the class that works with a View in a Model-View-Presenter (MVP) pattern.

Presentation model. A presentation model is a class that works with a view and serves as a combined representation of the presenter and model in a Model-View-Presenter (MVP) pattern.

Delegate command. A delegate command is a specific type of command in the Composite Application Library. For more information, see "Commands" in Chapter 6, "Technical Concepts."

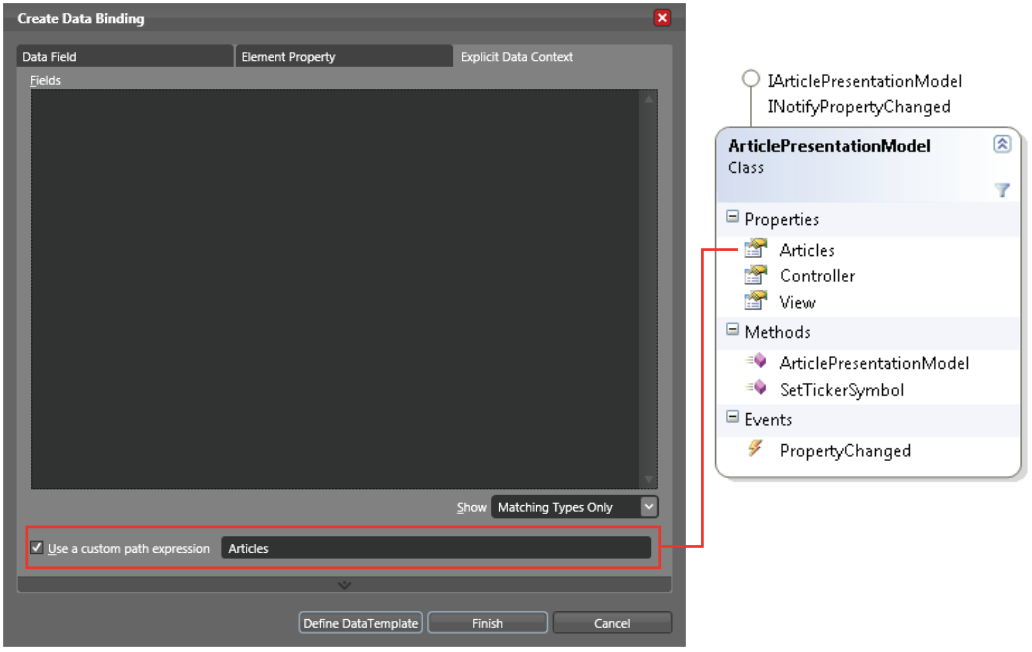


Figure 7.6 *Binding to data*

Figure 7.7 shows the technique for binding to a command where the **SubmitCommand** property, which is a **Delegate** command, is bound to the **SubmitCommand** property of the **Submit** button contained in the **OrdersCommandView.xaml** user control.

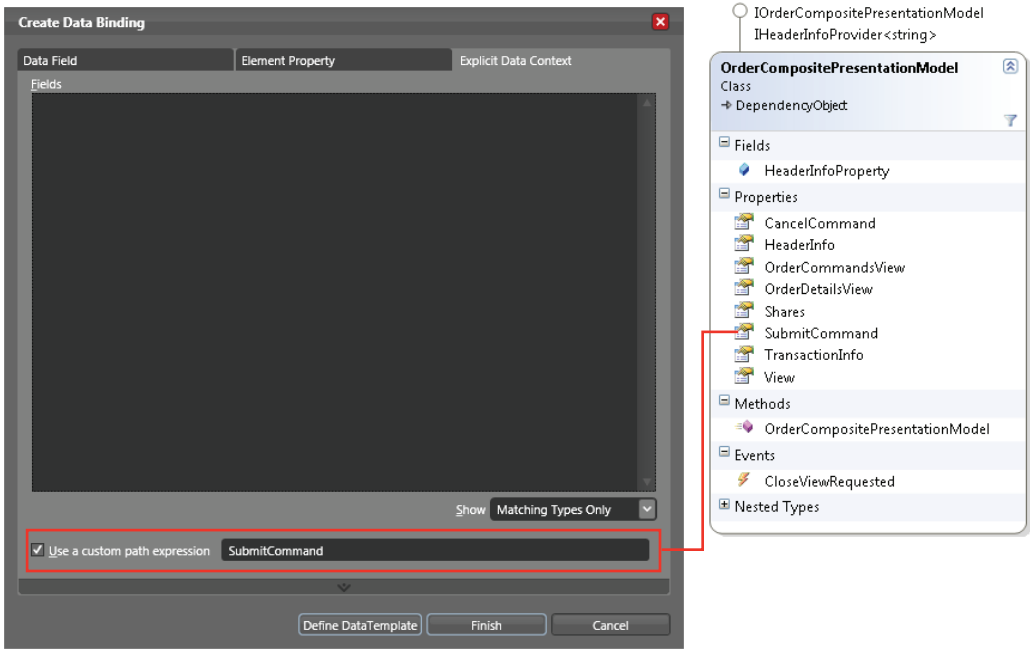


Figure 7.7 Binding to commands

Resources

Resources such as styles, resource dictionaries, and control templates can be scattered throughout an application—even more so with a composite application. When considering where to place resources, special attention needs to be paid to dependencies between UI elements and the resources they need. The Stock Trader RI solution is shown in Figure 7.8, with labels indicating the various areas where resources can live.

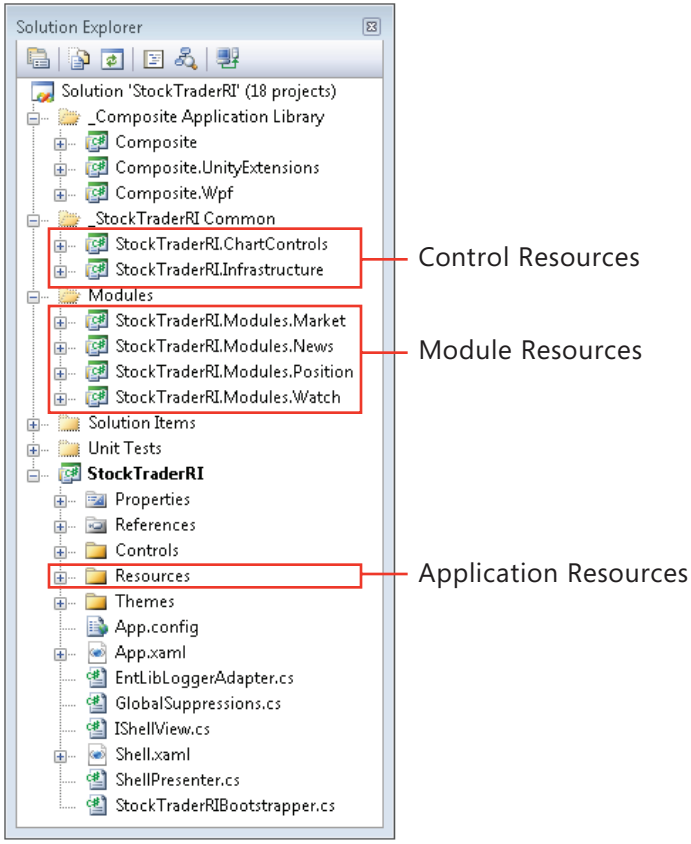


Figure 7.8 *Resource distribution across a solution*

Application Resources

Typically, application resources are resources that are available to an application as a whole. These types of 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 and behavior across the entire module and 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 to each individual module. Creating dependencies between modules can potentially 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. In terms of scope, these resources tend to have the finest scope because control libraries typically contain very specific controls and do not contain user controls, which in an application created with the Composite Application Library, usually go in the modules where they are used.

8

Development, Customization, and Deployment Information

This chapter summarizes the additional information included with the Composite Application Guidance and provides links to the getting started, development, customization, and deployment information that is available on MSDN.

Composite Application Guidance for WPF Hands-On Lab

The "Composite Application Guidance for WPF Hands-On Lab" on MSDN helps you learn the basic concepts of the Composite Application Guidance for WPF and apply them to create a Composite Application Library 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 understand the basic concepts of the Composite Application Guidance for WPF.
- You will create a new solution based on the Composite Application Library.
- You will create a module and load it statically.
- You will create a view and show it in the Shell window.

QuickStarts

The QuickStarts included with the Composite Application Guidance for WPF are brief, easy-to-understand illustrations of key software factory activities. QuickStarts 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. The Composite Application Guidance includes the following QuickStarts on MSDN:

- "Dynamic Modularity." This includes two QuickStarts that demonstrate how to build WPF applications composed of modules that are dynamically discovered and loaded at run time.

- "UI Composition." This demonstrates how to build a WPF user interface composed of different views that are dynamically loaded into regions and that interact with each other in a decoupled way.
- "Commanding." This demonstrates how to build a WPF user interface (UI) that uses commands provided by the Composite Application Library to handle UI actions in a decoupled way.
- "Event Aggregation." This 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.

Development Activities

Figure 8.1 illustrates the mapping of composite application concepts to How-to topics included with the Composite Application Guidance.

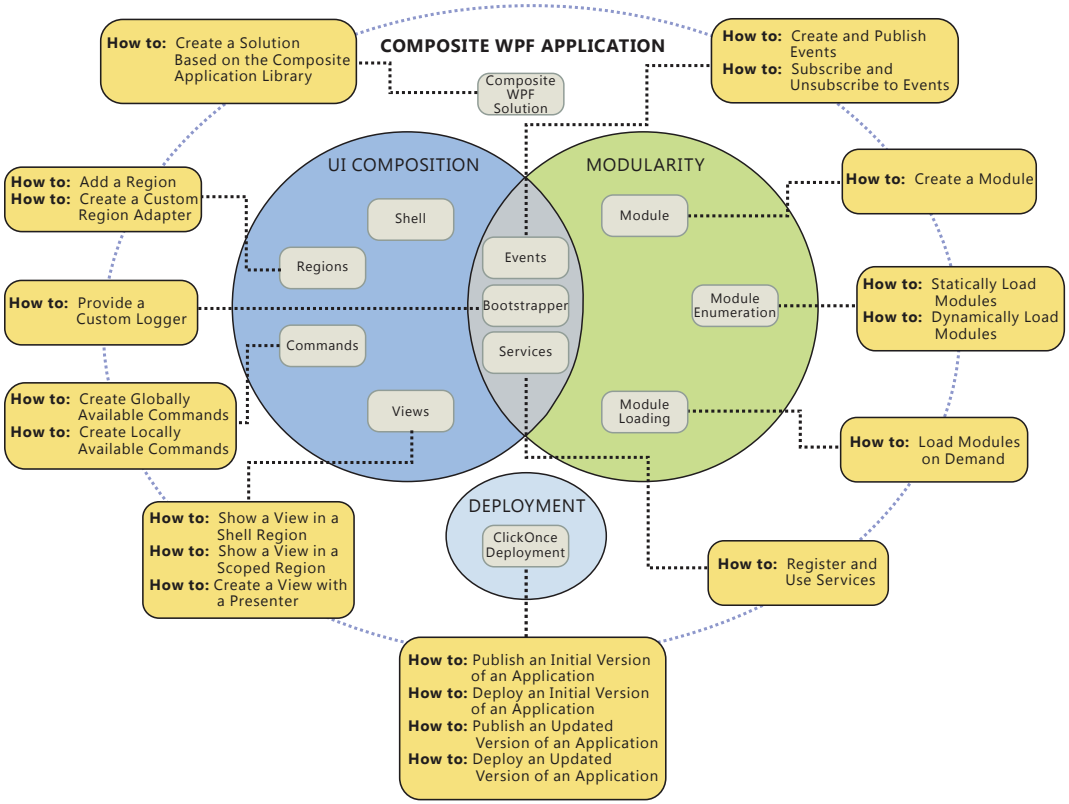


Figure 8.1 *Mapping of composite application concepts to How-to topics*

The next sections describe activities that developers usually perform when creating applications based on the Composite Application Library. Each How-to topic listed in the following sections provides the main steps for performing a particular task. As you review the How-to topics, consider how you can apply them to your application.

Creating Your Solution

A solution based on the Composite Application Library is a solution that you can use as a starting point for your composite WPF application. The solution includes recommended practices and techniques, and it is the basis for the procedures in the Composite Application Guidance for WPF. For procedures related to creating a solution that uses the Composite Application Library, see "How to: Create a Solution Using the Composite Application Library" on MSDN.

Bootstrapper

The bootstrapper is responsible for the initialization of an application built using the Composite Application Library. Having a bootstrapper gives you more control of how the Composite Application Library components are wired up to your application. The following topics on MSDN contain procedures that help you customize the bootstrapper class:

- "How to: Dynamically Load Modules." This topic describes how to configure the bootstrapper to dynamically load modules.
- "How to: Statically Load Modules." This topic describes how to configure the bootstrapper to statically load modules.
- "How to: Provide a Custom Logger." This topic describes how to use a different logger in your application that uses the Composite Application Library.
- "How to: Create a Custom Region Adapter." This topic describes how to configure the bootstrapper to register additional region adapter mappings.
- "How to: Register and Use Services." This topic describes how to configure the bootstrapper to register services in the application container.

For more information about the bootstrapper, see "Bootstrapper" in Chapter 6, "Technical Concepts."

Modules

A module encapsulates a set of related concerns. Modules are independently developed and deployed, and they interact with each other to create an application. For procedures related to creating modules, see the following topics on MSDN:

- "How to: Create a Module." This topic describes how to create a module.
- "How to: Dynamically Load Modules." This topic describes how to dynamically load modules.
- "How to: Statically Load Modules." This topic describes how to statically load modules.
- "How to: Load Modules On Demand." This topic describes how to load modules on demand.

For more information about modules, see "Module" in Chapter 6, "Technical Concepts."

Regions

Conceptually, a region is a mechanism that developers can use to expose to the application's WPF container controls—those that permit child elements—as components that encapsulate a particular visual way of displaying views (typically, views are user controls). Regions can be accessed in a decoupled way by their name and support adding or removing views dynamically at run time. For procedures related to regions, see the following topics on MSDN:

- "How to: Add a Region." This topic describes how to add a region to a view or the Shell window through XAML.
- "How to: Create a Custom Region Adapter." This topic describes how to create custom region adapters.
- "How to: Show a View in a Shell Region." This topic describes how to place a view in a Shell-defined region.
- "How to: Show a View in a Scoped Region." This topic describes how to create scoped regions and show views in scoped regions.

For more information about the regions, see "Region" in Chapter 6, "Technical Concepts."

Views

Views are objects that contain visual content. For more information about views, see the following topics on MSDN:

- "How to: Create a View with a Presenter." This topic describes how to create a view following the Model-View-Presenter pattern.
- "How to: Show a View in a Shell Region." This topic describes how to place a view in a Shell-defined region.
- "How to: Show a View in a Scoped Region." This topic describes how to create scoped regions and show views in scoped regions.

For more information about the shell and views, see "Shell and View" in Chapter 6, "Technical Concepts."

Services

A service is an object that provides functionality in a loosely coupled way to other components. These components can be in the same module or in other modules. The Composite Application Library includes a set of basic services that you can use in your applications. You can also develop your own services to provide infrastructure capabilities that are specific to your applications. For procedures related to services, see the following topic on MSDN:

- "How to: Register and Use Services." This topic describes how to register and obtain references to services in an application that uses the Composite Application Library and the Unity container.

For more information about the services, see "Container and Services" in Chapter 6, "Technical Concepts." For more information about when to use shared services, see "Communication" in Chapter 6, "Technical Concepts."

Commands

Commands are a way to handle user interface (UI) actions. They are a loosely coupled way to bind the UI to the logic that performs the action. For procedures related to commands, see the following topics on MSDN:

- "How to: Create Locally Available Commands." This topic describes how to create locally available commands.
- "How to: Create Globally Available Commands." This topic describes how to create globally available commands.

For more information about the commands, see "Commands" in Chapter 6, "Technical Concepts." For more information about when to use commands, see "Communication" in Chapter 6, "Technical Concepts."

Events

The Composite Application Library provides an event mechanism that enables communications between loosely coupled components in the application. By using this mechanism, based on the event aggregator service, publishers and subscribers can communicate through events that do not have a direct reference to each other. For more information about events, see the following topics on MSDN:

- "How to: Create and Publish Events." This topic describes how to create and publish an event that can be consumed in a loosely coupled way.
- "How to: Subscribe and Unsubscribe to Events." This topic describes how to subscribe and unsubscribe to an event that can be consumed in a loosely coupled way.

For more information about events, see "Event Aggregator" in Chapter 6, "Technical Concepts." For more information about when to use events, see "Communication" in Chapter 6, "Technical Concepts."

Customization Activities

The Composite Application Guidance for WPF contains assets that represent recommended practices for composite application development in WPF. Developers can use an unmodified version of the Composite Application Library. However, because each application is unique, you may want to modify it to suit your particular needs.

The following topics on MSDN describe how to customize the Composite Application Library to meet the needs of your development team:

- "How to: Create a Custom Region Adapter." This topic describes how to create custom region adapters.
- "How to: Provide a Custom Logger." This topic describes how to use a different logger in your application that uses the Composite Application Library.

For more information about extensibility points in the Composite Application Library, see "Customizing the Composite Application Library" in Chapter 4, "Composite Application Library."

Deployment Activities

ClickOnce is a 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. Composite WPF applications can use ClickOnce to get the shell, modules, and any other dependencies deployed to the client computer. The main challenge with composite applications is that ClickOnce does not easily address deploying dynamically loaded modules.

By using ClickOnce deployment, you can publish Windows-based applications to a Web server or network file share for simplified installation. The Composite Application Guidance for WPF includes guidance that helps you use ClickOnce to deploy an application built using the Composite Application Library. For more information, see the following topics on MSDN:

- "Deploying WPF Applications with ClickOnce." This topic provides general information about deploying WPF applications with ClickOnce.
- "How to: Publish an Initial Version of an Application." This topic describes how to publish an initial version of a composite WPF application.
- "How to: Deploy an Initial Version of an Application." This topic describes how to deploy an application to a client computer.
- "How to: Publish an Updated Version." This topic describes how to publish an updated version of your application.
- "How to: Deploy an Updated Version." This topic describes how to deploy an updated version of an application to a client computer.

The Manifest Manager Utility sample utility available on the Composite Application Guidance for WPF Community site 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 automated procedures for the How-to topics. Where appropriate, the activities are described using both the Manifest Manager Utility (automated) and Mage (manual) so that you can see the steps the utility automates.

Bibliography

This section consolidates the references provided in each chapter.

Chapter 1: Introduction

For more information and to download the source code, see "Composite Application Guidance for WPF" on MSDN:
http://www.microsoft.com/CompositeWpf

For more information about the Unity Application Block, see "Unity Application Block" on MSDN:
http://www.msdn.com/Unity

To ask questions and engage with the community, see "Composite Application Guidance for WPF" community site on CodePlex:
http://www.codeplex.com/CompositeWPF

For information about aspects of composite application development that are not addressed by this guidance, see the following sources:

- "Smart Client Architecture and Design Guide" on MSDN:
  http://msdn.microsoft.com/en-us/library/ms998506.aspx
- "Occasionally Connected Systems Architecture: The Client":
  http://www.developer.com/design/article.php/3708006
- "Occasionally Connected Systems Architecture: Concurrency":
  http://www.developer.com/design/article.php/3705396

For general information about WPF, see the following sources:

- "Windows Presentation Foundation" on MSDN:
  http://msdn.microsoft.com/en-us/library/ms754130.aspx
- Sells, Chris and Ian Griffiths. *Programming WPF: Building Windows UI with Windows Presentation Foundation*. Second Edition. O'Reilly Media, Inc., 2007.
- Nathan, Adam. *Windows Presentation Foundation Unleashed*. Indianapolis, IN: Sams Publishing, 2006.

For more information about other deliverables for building composite applications, see the following sources:

- "Smart Client Software Factory" on MSDN:
  http://msdn.microsoft.com/en-us/library/aa480482.aspx
- "Smart Client Contrib Project" on CodePlex:
  http://www.codeplex.com/scsfcontrib

For more information about the Composite UI Application Block, see "Composite UI Application Block" on MSDN:
http://msdn.microsoft.com/en-us/library/cc540684.aspx

Chapter 2: Design Concepts

For more information about regions, see "Region" in Chapter 6, "Technical Concepts" and "UI Composition QuickStart" on MSDN:
http://msdn.microsoft.com/en-us/library/cc707868.aspx

For more information about commands, see "Commands" in Chapter 6, "Technical Concepts" and "Commanding QuickStart" on MSDN:
http://msdn.microsoft.com/en-us/library/cc707837.aspx

For more information about the Publish/Subscribe pattern, see "Publish/Subscribe" on MSDN:
http://msdn.microsoft.com/en-us/library/ms978603.aspx

For more information about event aggregation, see "Event Aggregator" in Chapter 6, "Technical Concepts" and Event Aggregation QuickStart on MSDN:
http://msdn.microsoft.com/en-us/library/cc707857.aspx

For more information about modules, see "Module" in Chapter 6, "Technical Concepts" and Dynamic Modularity QuickStarts on MSDN:
http://msdn.microsoft.com/en-us/library/cc707860.aspx

For an introduction to dependency injection and inversion of control, see "Loosen Up - Tame Your Software Dependencies for More Flexible Apps" on MSDN:
http://msdn.microsoft.com/en-us/magazine/cc337885.aspx

For more information about dependency injection containers, see the following sources:

- "Unity Application Block" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc468366.aspx
- Castle Windsor:
  http://www.castleproject.org/container/index.html
- StructureMap:
  http://structuremap.sourceforge.net/Default.htm
- Spring.NET:
  http://www.springframework.net/

For an example of entering configuration information through the Unity container, see "Entering Configuration Information" on MSDN:
http://msdn.microsoft.com/en-us/library/cc440941.aspx

Chapter 3: Patterns in the Composite Application Library

The following are references and links to the patterns found in the Stock Trader RI and in the Composite Application Library:

- "Composite" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Adapter" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Façade" in Chapter 4, "Structural Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Template Method" in Chapter 5, "Behavioral Patterns," in *Design Patterns: Elements of Reusable Object-Oriented Software*[1]
- "Plugin" on Martin Fowler's Web site:
  http://www.martinfowler.com/eaaCatalog/plugin.html
- "Event Aggregator" on Martin Fowler's Web site:
  http://www.martinfowler.com/eaaDev/EventAggregator.html
- "Separated Interface" on Martin Fowler's Web site:
  http://www.martinfowler.com/eaaCatalog/separatedInterface.html
- "Model View Controller" and "Model-View-Presenter (MVP)" sections in "GUI Architectures" on Martin Fowler's Web site:
  http://martinfowler.com/eaaDev/uiArchs.html
- "Introduction to Model/View/ViewModel pattern for building WPF apps" on John Gossman's blog:
  http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx

For more information about the Dependency Injection pattern, see the following sources:

- "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site at:
  http://www.martinfowler.com/articles/injection.html
- "Design Patterns: Dependency Injection" by Griffin Caprio on MSDN:
  http://msdn.microsoft.com/en-us/magazine/cc163739.aspx

For more information about the Unity Application Block, see "Unity Application Block" on MSDN:
http://www.msdn.com/Unity

For more information about Inversion of Control patterns, see "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site:
http://www.martinfowler.com/articles/injection.html

1 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

For more information about the Service Locator pattern, see the following sources:

- "Inversion of Control Containers and the Dependency Injection pattern" on Martin Fowler's Web site:
  http://www.martinfowler.com/articles/injection.html
- "Service Locator" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc304894.aspx

For more information about Separated Presentation patterns, see "Separated Presentation" on Martin Fowler's Web site:
http://www.martinfowler.com/eaaDev/SeparatedPresentation.html

For more information about the Supervising Controller pattern, see the following sources:

- "Supervising Controller" on Martin Fowler's Web site:
  http://www.martinfowler.com/eaaDev/SupervisingPresenter.html
- "Model-View-Presenter Pattern" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc304760.aspx

For more information about data-binding in WPF, see "Data Binding" on MSDN:
http://msdn.microsoft.com/en-us/library/ms750612.aspx

- For more information about the Observer pattern, see "Exploring the Observer Design Pattern" on MSDN:
  http://msdn.microsoft.com/en-us/library/ms954621.aspx

For more information about data templates, see "Data Templating Overview" on MSDN:
http://msdn.microsoft.com/en-us/library/ms742521.aspx

For more information about the Presentation Model pattern, see "Presentation Model" on Martin Fowler's Web site:
http://www.martinfowler.com/eaaDev/PresentationModel.html

Chapter 4: Composite Application Library

Solution

For more information about building a solution, see the following source:

- "How to: Create a Solution Based on the Composite Application Library" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707864.aspx

Bootstrapper

For more information about the bootstrapper, see the following sources:

- "Bootstrapper" in Chapter 6, "Technical Concepts"
- "Container and Services" in Chapter 6, "Technical Concepts"

- "Composite Application Guidance for WPF Hands-On Lab" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707878.aspx
- "Dynamic Modularity QuickStarts" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707860.aspx
- "How to: Provide a Custom Logger" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707911.aspx
- "How to: Dynamically Load Modules" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707834.aspx
- "How to: Statically Load Modules" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707839.aspx
- "How to: Load Modules on Demand" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707852.aspx

Container and Services

For more information about the container and services, see the following sources:

- "Container and Services" in Chapter 6, "Technical Concepts"
- "Container" in Chapter 2, "Design Concepts"
- "How to: Register and Use Services" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707881.aspx

Modules

For more information about modules, see the following sources:

- "Module" in Chapter 6, "Technical Concepts"
- "How to: Create a Module" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707899.aspx
- For module loading How-to topics, see the "Bootstrapper" section listed earlier.

Regions

For more information about regions, see the following sources:

- "Region" in Chapter 6, "Technical Concepts"
- "UI Composition QuickStart" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707868.aspx
- "How to: Add a Region" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707908.aspx

Shell and Views

For more information about shells and views, see the following sources:

- "Shell and View" in Chapter 6, "Technical Concepts"
- "UI Composition" in Chapter 2, "Design Concepts"
- "How to: Show a View in a Shell Region" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707854.aspx
- "How to: Show a View in a Scoped Region" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707903.aspx
- "How to: Create a View with a Presenter" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707895.aspx

Events

For more information about events, see the following sources:

- "Communication" in Chapter 6, "Technical Concepts"
- "Event Aggregator" in Chapter 6, "Technical Concepts"
- "Event Aggregation QuickStart" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707857.aspx
- "How to: Create and Publish Events" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707855.aspx
- "How to: Subscribe and Unsubscribe Events" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707892.aspx

Commands

For more information about commands, see the following sources:

- "Communication" in Chapter 6, "Technical Concepts"
- "Commands" in Chapter 6, "Technical Concepts"
- "Commanding QuickStart" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707837.aspx
- "How to: Create Locally Available Commands" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707896.aspx
- "How to: Create Globally Available Commands" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707861.aspx

Deployment

For more information about deployment, see the following sources:

- "Deploying WPF Applications with ClickOnce" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707835.aspx
- "How to: Publish an Initial Version of an Application" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707846.aspx
- "How to: Deploy an Initial Version of an Application" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707893.aspx
- "How to: Publish an Updated Version" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707849.aspx
- "How to: Deploy an Updated Version" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc671928.aspx

Chapter 5: Stock Trader Reference Implementation

To download and install the Stock Trader RI, see "Composite Application for WPF" on MSDN:
http://www.microsoft.com/CompositeWpf

To run the Stock Trader RI, see "Installing and Running" on MSDN:
http://msdn.microsoft.com/en-us/library/cc707870.aspx

For more information about the concepts described in this chapter, see the following technical concepts in Chapter 6, "Technical Concepts":

- "Module"
- "Commands"
- "Region"
- "Shell and View"
- "Container and Services"
- "Bootstrapper"
- "Event Aggregator"

For more information about the Presentation Model pattern, see "Presentation Model Pattern" in Chapter 3, "Patterns in the Composite Application Library."

For more information about the Unity Application Block, see "Unity Application Block" on MSDN:
http://www.msdn.com/Unity

For an introduction to dependency injection and inversion of control, see "Loosen Up - Tame Your Software Dependencies for More Flexible Apps" on MSDN:
http://msdn.microsoft.com/en-us/magazine/cc337885.aspx

Chapter 6: Technical Concepts

Bootstrapper

For more information on the bootstrapper, see the following sources:

- To set up the application's bootstrapper, see "How to: Create a Solution Using the Composite Application Library" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707864.aspx
- To configure the bootstrapper to dynamically load modules, see "How to: Dynamically Load Modules" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707834.aspx
- To configure the bootstrapper to statically load modules, see "How to: Statically Load Modules" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707839.aspx
- To use a different logger in your application that uses the Composite Application Library, see "How to: Provide a Custom Logger" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707911.aspx
- To configure the bootstrapper to register additional region adapter mappings, see "How to: Create a Custom Region Adapter" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707884.aspx
- To configure the bootstrapper to register services in the application container, see "How to: Register and Use Services" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707881.aspx

Container and Services

For more information about services and containers, see the following sources:

- "Container" in Chapter 2, "Design Concepts"
- "How to: Register and Use Services" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707881.aspx

Modules

For more information about modules, see the following soruces:

- "Modularity" in Chapter 2, "Design Concepts"
- "Dynamic Modularity QuickStarts" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707860.aspx

- Modularity How-to topics on MSDN:
  - "How to: Create a Module":
    http://msdn.microsoft.com/en-us/library/cc707899.aspx
  - "How to: Dynamically Load Modules":
    http://msdn.microsoft.com/en-us/library/cc707834.aspx
  - "How to: Statically Load Modules":
    http://msdn.microsoft.com/en-us/library/cc707839.aspx
- Modularity patterns in Chapter 3, "Patterns in the Composite Application Library" (these patterns provide possible approaches to building composite applications):
  - "Inversion of Control Pattern"
  - "Dependency Injection Pattern"
  - "Service Locator Pattern"

Regions

For more information about regions, see the following sources:

- "UI Composition QuickStart" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707868.aspx
- Region How-to topics:
  - "How to: Add a Region":
    http://msdn.microsoft.com/en-us/library/cc707908.aspx
  - "How to: Create a Custom Region Adapter":
    http://msdn.microsoft.com/en-us/library/cc707884.aspx
  - "How to: Provide a Custom Logger":
    http://msdn.microsoft.com/en-us/library/cc707911.aspx
  - "How to: Show a View in a Scoped Region":
    http://msdn.microsoft.com/en-us/library/cc707903.aspx
  - "How to: Show a View in a Shell Region":
    http://msdn.microsoft.com/en-us/library/cc707854.aspx
- Composition patterns:
  - "Composite and Composite View" in Chapter 3, "Patterns in the Composite Application Library"

Shell and Views

For more information about shells and views, see the following sources:

- "UI Composition" in Chapter 2, "Design Concepts"
- "How to: Show a View in a Shell Region" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707854.aspx
- "How to: Show a View in a Scoped Region" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707903.aspx
- "How to: Create a View with a Presenter" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707895.aspx

Events

For more information about events, see the following sources:

- "Event Aggregation QuickStart" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707857.aspx
- "How to: Create and Publish Events" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707855.aspx
- "How to: Subscribe and Unsubscribe to Events" on MSDN:
  http://msdn.microsoft.com/en-us/library/cc707892.aspx
- "Event Aggregator" on Martin Fowler's Web site:
  http://www.martinfowler.com/eaaDev/EventAggregator.html

Commands

For more information about the logical tree in WPF, see "Trees in WPF" on MSDN:
http://msdn.microsoft.com/en-us/library/ms753391.aspx

For more information about commands, see the following topics on MSDN:

- "Commanding QuickStart":
  http://msdn.microsoft.com/en-us/library/cc707837.aspx
- "How to: Create Globally Available Commands":
  http://msdn.microsoft.com/en-us/library/cc707861.aspx
- "How to: Create Locally Available Commands":
  http://msdn.microsoft.com/en-us/library/cc707896.aspx

Chapter 7: Designer Guidance

For information about creating region adaptors that can work with any type of UI element, see "How to: Create a Custom Region Adapter" on MSDN:
http://msdn.microsoft.com/en-us/library/cc707884.aspx

Chapter 8: Development, Customization, and Deployment Information

Getting Started

For more information and to download the source code, see "Composite Application Guidance for WPF" on MSDN:
http://www.microsoft.com/CompositeWpf

To learn about and get started with the Composite Application Library, see the following topics on MSDN:

- "Composite Application Guidance for WPF Hands-On Lab":
  http://msdn.microsoft.com/en-us/library/cc707878.aspx
- QuickStarts:
  - "Dynamic Modularity QuickStarts":
    http://msdn.microsoft.com/en-us/library/cc707860.aspx
  - "UI Composition":
    http://msdn.microsoft.com/en-us/library/cc707868.aspx
  - "Commanding":
    http://msdn.microsoft.com/en-us/library/cc707837.aspx
  - "Event Aggregation":
    http://msdn.microsoft.com/en-us/library/cc707857.aspx
- "Stock Trader Reference Implementation":
  http://msdn.microsoft.com/en-us/library/cc707869.aspx

Development Topics

For more information about activities that developers usually perform when creating applications based on the Composite Application Library, see the following topics on MSDN:

- "How to: Create a Solution Using the Composite Application Library":
  http://msdn.microsoft.com/en-us/library/cc707864.aspx
- "How to: Create a Module":
  http://msdn.microsoft.com/en-us/library/cc707899.aspx
- "How to: Dynamically Load Modules":
  http://msdn.microsoft.com/en-us/library/cc707834.aspx
- "How to: Statically Load Modules":
  http://msdn.microsoft.com/en-us/library/cc707839.aspx
- "How to: Load Modules On Demand":
  http://msdn.microsoft.com/en-us/library/cc707852.aspx

- "How to: Add a Region":
  http://msdn.microsoft.com/en-us/library/cc707908.aspx
- "How to: Show a View in a Shell Region":
  http://msdn.microsoft.com/en-us/library/cc707854.aspx
- "How to: Show a View in a Scoped Region":
  http://msdn.microsoft.com/en-us/library/cc707903.aspx
- "How to: Create a View with a Presenter":
  http://msdn.microsoft.com/en-us/library/cc707895.aspx
- "How to: Register and Use Services":
  http://msdn.microsoft.com/en-us/library/cc707881.aspx
- "How to: Create Locally Available Commands":
  http://msdn.microsoft.com/en-us/library/cc707896.aspx
- "How to: Create Globally Available Commands":
  http://msdn.microsoft.com/en-us/library/cc707861.aspx
- "How to: Create and Publish Events":
  http://msdn.microsoft.com/en-us/library/cc707855.aspx
- "How to: Subscribe and Unsubscribe to Events":
  http://msdn.microsoft.com/en-us/library/cc707892.aspx

Customization Topics

For more information about activities that developers can perform to customize the Composite Application Library, see the following topics on MSDN:

- "How to: Create a Custom Region Adapter":
  http://msdn.microsoft.com/en-us/library/cc707884.aspx
- "How to: Provide a Custom Logger":
  http://msdn.microsoft.com/en-us/library/cc707911.aspx

Deployment Topics

For more information about deploying WPF applications with ClickOnce, see the following topics on MSDN:

- "Deploying WPF Applications with ClickOnce":
  http://msdn.microsoft.com/en-us/library/cc707835.aspx
- "How to: Publish an Initial Version of an Application":
  http://msdn.microsoft.com/en-us/library/cc707846.aspx
- "How to: Deploy an Initial Version of an Application":
  http://msdn.microsoft.com/en-us/library/cc707893.aspx
- "How to: Publish an Updated Version":
  http://msdn.microsoft.com/en-us/library/cc707849.aspx
- "How to: Deploy an Updated Version":
  http://msdn.microsoft.com/en-us/library/cc671928.aspx

For a demonstration of how to use the ClickOnce publishing API to manage deployment and application manifests in a simpler way, see the Manifest Manager Utility sample utility available on the Composite Application Guidance for WPF Community site:
http://www.codeplex.com/CompositeWPF/Release/ProjectReleases.aspx?ReleaseId=14771

Index

A

active aware commands, 89
Adapter pattern, 31
AddWatchPresenter, 86
AddWatchView, 86
adoption experience, 8
  Composite Application Library, 64
applications
  deployment, 66
  development challenges, 4-5
  resources, 150
architecture
  Composite Application Library baseline architecture, 55-57
  goals, 61
  goals and principles, 7-8
  scenario, 76-77
assemblies
  Microsoft.Practices.Composite, 64
  Microsoft.Practices.Composite.UnityExtensions, 64
  Microsoft.Practices.Composite.Wpf, 64
assets, 2
audience, 1
authors and contributors, xi

B

bibliography, 161-173
  Composite Application Library, 164-167
  bootstrapper, 164-165
  commands, 166
  container and services, 165
  deployment, 167
  events, 166
  modules, 165
  regions, 165
  shells and views, 166
  solution, 164
  customization topics, 172
  deployment topics, 173
  design concepts, 162
  designer guidance, 170
  development, customization, and deployment, 171-173
  development topics, 171-172
  patterns in the Composite Application Library, 163-164
  Stock Trader RI, 167
  technical concepts, 168-170
  bootstrapper, 168
  commands, 170
  container and services, 168
  events, 170
  modules, 168-169
  regions, 169
  shells and views, 170
bootstrapper, 92-97
  bibliography, 164-165, 168
  Composite Application Library, 59-60, 69
  configuring region adapter mappings, 94-95
  container configuration, 93-94
  development activities, 155
  module loading, 96
  more information, 97, 155
  shell creation, 95
  Stock Trader RI, 81
  technical challenges, 90

C

challenges, 4-5, 87-90
commanding, 17-18
Commanding QuickStart described, 154
Command pattern, 31
commands
  activity monitoring behavior, 132-133
  bibliography, 166, 170
  CompositeCommand, 130-132
  Composite commands, 134
  delegate commands, 134
  DelegateCommand<T>, 18, 129
  development activities, 157
  FAQ, 134
  IActiveAware interface, 129
  more information, 157
  order of execution, 134
  RegisterCommand, 133
  routed commands, 134
  Stock Trader RI, 85-86
  Submit button, 132
  technical challenges, 88
  technical concepts, 128-135
  UnregisterCommand, 133
  WPF commands, 134
common technical challenges, 87-90
communication
  commanding, 135-136
  Composite Application Library, 69
  EventAggregator, 136
  shared services, 136-137
  technical concepts, 135-137
community, 4
compatibility defined, 8
Composite Application Guidance, 9-10
Composite Application Guidance for WPF
  documentation, 3
  Hands-On Lab described, 2, 153
Composite Application Library, 53-69
  application deployment, 66
  baseline architecture, 55-57

bibliography, 164-167
bootstrapper, 59-60, 69
communication, 69
concepts diagram, 91
container, 69
core services, 99
customizing, 66-69
development activities, 66
exposing functionality, 67
extensibility guidelines, 67
extensibility points, 68
goals and benefits, 61-64
adoption experience, 64
architectural goals, 61
design goals, 62
extensibility, 64
modularity, 63
user interface composition, 63
IContainerFacade interface, 69
library extension, 67
logging, 68
modification recommendations, 68
modularity, 63
modules, 61, 69
new applications, 58
organization, 64-65
regions, 68
services, 99
shell defining, 59
solution layout diagram, 65
system requirements, 54
technical concepts, 65
UnityBootstrapper, 69
UnityContainerAdapter class, 30
when to use, 57
composite applications
benefits, 5-7
described, 1, 139
how-to topics, 154-159
with multiple back-end systems diagram, 6
package diagram, 54
patterns diagram, 27
CompositeCommand class, 18, 31, 69, 130-136
composite commands, 89
Composite pattern, 28-29
more information, 29
Template pattern, 30
Composite UI Application Block
compared to Composite Application Guidance, 11-12
comparing, 11-12
customer feedback, 11
Windows Forms development experience, 12
WPF, 12
Composite UI deliverables, 10-11
composite views, 88
Composite WPF applications described, 53
CompositeWpfEvent class, 19, 124-125
composition, 18
example, 29
concerns not addressed, 9
configuration driven module loading, 108-109
Configuration Modularity QuickStart, 40
container and services
Composite Application Library services, 99-100
more information, 100
containers, 20-25
advantages, 22-23
bibliography, 165
Composite Application Library, 69
composition described, 140-142
considerations, 25
IContainerFacade considerations, 98
IContainerFacade interface, 97
registering, 24
resolving, 24-25
technical concepts, 97-100
UnityContainer, 98
ContentControl-RegionAdapter region adapter, 116
Contoso Financial Investments (CFI) See Stock Trader RI
control resources, 151
customization
activities, 158
Composite Application Library, 66-69

D

data binding, 147-149
Presentation Model pattern, 51
Supervising Controller pattern, 48
DelegateCommand class, 31, 69
delegate command defined, 148
DelegateCommands, 131-132
DelegateCommand<T> class, 18, 129
delegation, 17
Dependency Injection (DI) pattern, 34-37
constructor injection, 35-36
forces, 35
Inversion of Control (IoC) pattern, 30
liabilities, 37
more information, 37
problem, 34
related patterns, 37
setter injection, 35-36
solution, 35-37
Unity Application Block (Unity), 36
deployment, 167
activities, 159
design
concepts, 13-25
goals, 62
designer guidance, 139-151
data binding, 147-149
layout, 140-145
container composition, 140-142
region, 142-145
resources, 149-151
application resources, 150
control resources, 151
module resources, 151
visual representation, 146-147
custom controls, 147
data templates, 147
user controls, 147

development activities, 154-158
bootstrapper, 155
challenges, 4-5
commands, 157
Composite Application Library, 66
diagram, 154
information, 153-158
modules, 156
regions, 156
services, 157
solution creation, 155
views, 157
directory driven module loading, 108
documentation, 2-4
Dynamic Modularity QuickStart described, 153
dynamic module loading, 108

E

event aggregation, 18-20, 87
Event Aggregation QuickStart described, 154
Event Aggregator pattern, 32
Stock Trader RI, 87
technical challenges, 87-90
EventAggregator service, 19, 123-128, 136
CompositeWpfEvent class, 124-125
default subscriptions, 126
IEventAggregator interface, 124
more information, 128
publishing an event, 127-128
subscribing on UI thread, 126
subscribing to an event, 125
subscribing with strong references, 127
subscription filtering, 127
unsubscribing from an event, 128
eventing, 18-20
bibliography, 166
Event Services, 18-19
extensibility
Composite Application Library, 64
defined, 8
guidelines, 67
points, 68

F

Façade pattern, 32
figures
development activities, 154
documentation, 3
logical architecture, 78
modularity, 21
Stock Trader RI, 72
Stock Trader RI composite application, 7
Stock Trader RI shell regions, 113
functionality, 67

G

GetModule method, 110
GetModules method, 110
GetStartupLoadedModules method, 110
goals and principles, 7-8
guidelines for choosing Composite UI deliverables, 10-11

H

Hands-On Lab described, 2, 153
how-to topics, 154-159

I

IAccountPositionService, 99
IActiveAware interface, 130
IContainerFacade interface, 69, 97-99
IEventAggregator interface, 99, 124
ILoggerFacade, 99
IMarketFeedService, 99
IMarketHistoryService, 99
IModuleEnumerator, 99
IModule interface, 104
IModuleLoader interface, 99, 110-111
implementation view, 79
INewsFeedService, 99
intended audience, 1
Inversion of Control (IoC) pattern, 30-31, 38-41
Dependency Injection (DI) pattern, 30
forces, 39
implementation details, 39-41
liabilities, 41
more information, 41
problem, 38
related patterns, 41
and Service Locator pattern, 31
solution, 39
Template Method pattern, 30
IOrdersService, 99
IRegion interface, 116
IRegionManager, 99
ItemsControlRegionAdapter region adapter, 116
IWatchListService, 99

L

labs, 153
layout, 14-16
designer guidance, 140-145
learnability defined, 8
library See Composite Application Library
ListBox Region pictured, 114
loading
configuration driven module loading, 108-109
directory driven module loading, 108
dynamic module loading, 108
on-demand module loading, 109
static module loading, 106-107
logical architecture, 77-78

M

Manifest Manager Utility sample utility, 159
Model-View-ViewModel pattern, 52

modularity, 20-22
  Composite Application Library, 63
  designing, 21-22
  diagram, 21
ModuleInfo constructor described, 110
ModuleLoader service, 82
modules, 100-111
  bibliography, 165
  Composite Application Library, 61
  considerations, 105
  described, 100-101
  design, 103
  development activities, 156
  enumeration, 82
  IModule interface, 104-105
  loading, 82, 105-111
  configuration driven module loading, 108-109
  directory driven module loading, 108
  dynamic module loading, 108
  on-demand module loading, 109
  static module loading, 106-107
  module enumerator, 109-110
  module loader, 110-111
  resources, 151
  Stock Trader RI, 81
  team development with, 101-102
  technical concepts, 111
monolithic application described, 139
monolithic style, 5
more information
  bootstrapper, 97, 155
  commands, 157
  Composite pattern, 29
  container and services, 100
  customization activities, 158
  Dependency Injection (DI) pattern, 37
  deployment activities, 159
  EventAggregator service, 128
  Inversion of Control (IoC) pattern, 41
  modules, 156
  patterns, 33
  Presentation Model pattern, 52
  regions, 118-119, 156
  Separated Presentation pattern, 47
  Service Locator pattern, 45
  services, 100, 157
  shell and view, 123
  Supervising Controller pattern, 49
  technical challenges, 87-90
  views, 157
  WPF, 10
more information See also bibliography; technical concepts

N

NewsModule class, 43
NewsModuleFixture test class, 44
NewsReaderPresenter class, 36

O

on-demand module loading, 109
order of execution, 134
overview, 4-10
  concerns not addressed, 9
  considerations for choosing, 9-10

P

patterns, 27-52
performance defined, 8
Plug-In pattern, 30
PositionSummaryPresentation-Model, 87
presentation model
  defined, 148
  Stock Trader RI, 83
Presentation Model pattern, 50-52
  data binding, 51
  forces, 50
  more information, 52
  Presentation Model pattern logical view, 51
  problem, 50
  related patterns, 52
  Separated Presentation pattern, 45
  solution, 50-52
Presentation Model pattern logical view, 51
presenter defined, 148
Publish/Subscribe pattern, 18-20

Q

QuickStarts, 153-154

R

RegionManager, 84
regions, 111-119
  bibliography, 165
  designer guidance, 142-145
  development activities, 156
  IRegion interface, 116
  more information, 118-119, 156
  multiple view layout, 112-114
  region adapters, 116-117
  RegionManager class, 114-115
  scoped regions, 117
  Stock Trader RI, 84
  technical challenges, 88
  template layout, 112
  working with, 114-117
resources
  application resources, 150
  control resources, 151
  module resources, 151
  root application resources, 151
root application resources, 151
RoutedUICommand mechanism, 17

S

scalability defined, 8
scenarios, 9-11
  architecture, 76-77
SelectorRegionAdapter region adapter, 116
Separated Interface pattern, 30

Separated Presentation pattern, 45-47
forces, 45
liabilities, 46
more information, 47
overview, 32
Presentation Model pattern, 45
problem, 45
related patterns, 46
solution, 45-46
Supervising Controller pattern, 45
Service Locator pattern, 42-45
forces, 42-43
and Inversion of Control (IoC) pattern, 31
liabilities, 44
more information, 45
overview, 31
problem, 42
related patterns, 44-45
solution, 43-44
Unity Application Block, 43-44
service-oriented architecture (SOA), 74
services
bibliography, 165
Composite Application Library services, 99-100
and containers, 81
Event Services, 18-19
IContainerFacade considerations, 98
IContainerFacade interface, 97
more information, 100, 157
registration, 85
shared services, 136-137
technical challenges, 89
technical concepts, 97-100
UnityContainer, 98
setter injection, 35-36
shared services, 136-137
shell
Composite Application Library, 59
described, 119
shell and view, 119-123, 166
composite views, 122
more information, 123
shell implementing, 119
Stock Trader RI shell, 120-121
Stock Trader RI views, 122-123
view implementing, 121-122
views and design patterns, 123
simplicity defined, 8
singletons, 24
Smart Client Software Factory, 10
solutions
bibliography, 164
creation development activities, 155
startup process, 80
static module loading, 106-107
Stock Trader Reference Implementation See Stock Trader RI
Stock Trader RI, 71-90
bootstrapping, 81
CFI
development challenges, 76
emerging requirements, 74
meeting objectives, 75
operating environment, 73
operational challenges, 74
solution, 76-77
commands, 85-86
common technical challenges, 87-90
composite application diagram, 7
diagram, 72
event aggregator, 87
features, 77
how it works, 80-87
implementation view, 79
logical architecture, 77-78
main window during run time, 141
main window in Microsoft Expression Blend, 141
module enumeration, 82
module loading, 82
modules, 81
presentation model, 83
RegionManager, 84
regions, 84
service registration, 85
services and containers, 81
shell regions pictured, 113
startup process, 80
teams, 101
view registration, 83
views, 82-83
views (illustration), 14
Stock Trader RI See also CFI
strategy for CFI, 75
strong references, 127
subsetability defined, 8
Supervising Controller logical view, 48
Supervising Controller pattern, 48
Supervising Controller pattern, 47-49
data binding, 48
forces, 47
liabilities, 49
more information, 49
problem, 47
related patterns, 49
Separated Presentation pattern, 45
solution, 47-48
Supervising Controller logical view, 48
View updates, 48

T

technical challenges, 87-90
technical concepts, 91-137
bootstrapper, 92-97
bibliography, 168-170
code creating the shell, 95
configuring region adapter mappings, 94-95

container configuration, 93-94
module loading, 96
more information, 97
commands, 128-135
activitiy monitoring behaviour, 132-133
CompositeCommand, 130-132
Composite commands, 134
delegate commands, 134
DelegateCommand<T>, 129
FAQ, 134
IActiveAware interface, 129
order of execution, 134
RegisterCommand, 133
routed commands, 134
Submit button, 132
UnregisterCommand, 133
WPF commands, 134
communication, 135-137
commanding, 135-136
EventAggregator, 136
shared services, 136-137
container and services, 97-100
Composite Application Library services, 99-100
IContainerFacade considerations, 98
IContainerFacade interface, 97
more information, 100
UnityContainer, 98
EventAggregator service, 123-128
CompositeWpfEvent class, 124-125
default subscriptions, 126
IEventAggregator interface, 124
more information, 128
publishing an event, 127-128
subscribing on UI thread, 126
subscribing to an event, 125
subscribing with strong references, 127
subscription filtering, 127
unsubscribing from an event, 128
module, 100-111
considerations, 105
described, 100-101
design, 103
IModule interface, 104-105
loading, 105-111
configuration driven module loading, 108-109
directory driven module loading, 108
dynamic module loading, 108
on-demand module loading, 109
static module loading, 106-107
module enumerator, 109-110
module loader, 110-111
team development with, 101-102
technical concepts, 111
region, 111-119
IRegion interface, 116
more information, 118-119
multiple view layout, 112-114
region adapters, 116-117
RegionManager class, 114-115
scoped regions, 117
template layout, 112
working with, 114-117
shell and view, 119-123
composite views, 122
more information, 123
shell implementing, 119
Stock Trader RI shell, 120-121
Stock Trader RI views, 122-123
view implementing, 121-122
views and design patterns, 123
Template Method pattern, 30
testability defined, 8
TickerSymbolSelectedEvent, 125-126

U

UI composition, 13-14
Composite Application Library, 63
technical challenges, 88
UI Composition QuickStart, 154
Unity Application Block (Unity)
Dependency Injection (DI) pattern, 36
Service Locator pattern, 43-44
UnityBootstrapper, 59-60, 69, 81, 96
UnityContainer, 31
UnityContainerAdapter class, 30, 98
upgradeability defined, 8

V

View Discovery, 16
views
defined, 148
described, 119
development activities, 157
injection, 15
more information, 157
registration in Stock Trader RI, 83
Stock Trader RI, 82-83
views See also shell and view

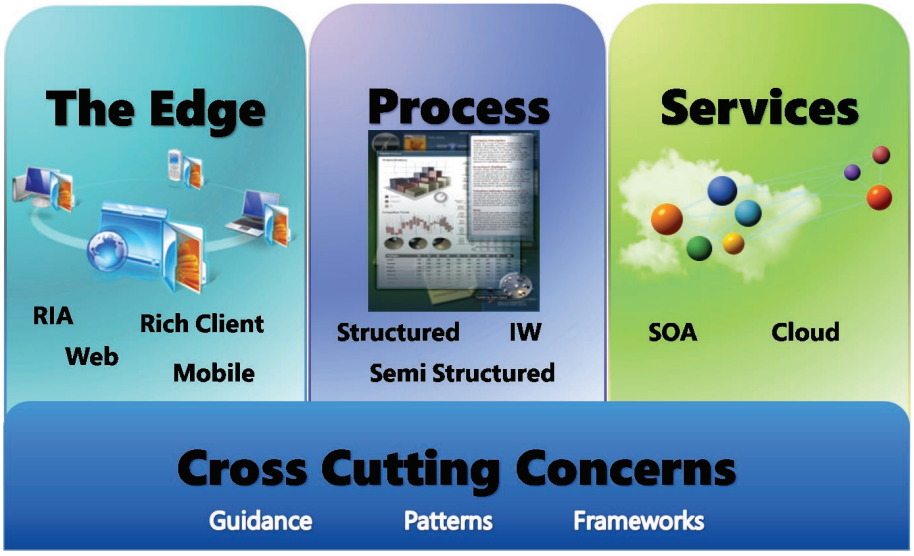
W

Web sites See more information
Windows Forms development experience, 12
WPF
Composite Application Guidance for Hands-On Lab, 2, 153
creating composite application, 58
technical challenges, 90
WPF/CAB layer, 10-11

About Microsoft *patterns & practices*

The Microsoft *patterns & practices* team is responsible for delivering applied engineering guidance that helps software architects, developers, and their teams take full advantage of Microsoft's platform technologies. Organizations around the world use Microsoft's proven software engineering practices to reduce project cost, reduce risk, save time, and prepare for future Microsoft technologies.

patterns & practices areas of guidance include those shown in the following image:



patterns & practices guides are reviewed by Microsoft engineering teams, consultants, and by partners and customers.

To learn more about *patterns & practices*, visit *http://www.microsoft.com/practices*.