**目录**

该教程包括以下主题:

* [自述](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40))
* [Prism 4.0更新说明](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40))
* Chapter 1: [介绍](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40))
* Chapter 2: [初始化Prism](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40))
* Chapter 3: [组件间的依赖管理](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40))
* Chapter 4: [模块化应用开发](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40))
* Chapter 5: [实现MVVM模式](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))
* Chapter 6: [MVVM模式进阶](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))
* Chapter 7: [UI](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))组合
* Chapter 8: [导航](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))
* Chapter 9: [松耦合组件之间的通讯](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40))
* Chapter 10: [Silverlight 和 WPF之间共享代码](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40))
* Chapter 11: [部署Prism应用](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40))
* Appendix A: [词汇表](http://msdn.microsoft.com/en-us/library/ff921135(v=pandp.40))
* Appendix B: [Prism中的设计模式](http://msdn.microsoft.com/en-us/library/ff921146(v=pandp.40))
* Appendix C: [Prism库](http://msdn.microsoft.com/en-us/library/gg405476(v=pandp.40))

当你从MSDN下载Prism时，文档中还包括了以下主题:

* Appendix D: 从早期版本升级
  + [从Prism 2.x](http://msdn.microsoft.com/en-us/library/gg430859(v=pandp.40))升级
  + [从WPF 1.0](http://msdn.microsoft.com/en-us/library/ff921144(v=pandp.40))的复合应用升级
  + [从复合UI应用块升级](http://msdn.microsoft.com/en-us/library/ff921081(v=pandp.40))
* Appendix E: [扩展Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))
* Appendix F: 实现参考
  + [Stock Trader 实现](http://msdn.microsoft.com/en-us/library/ff921074(v=pandp.40))
  + [MVVM 实现](http://msdn.microsoft.com/en-us/library/gg405492(v=pandp.40))
* Appendix G: 快速入门
  + [WPF](http://msdn.microsoft.com/en-us/library/ff921068(v=pandp.40))模块化快速入门
  + [Silverlight模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921163(v=pandp.40))
  + [MVVM基础快速入门](http://msdn.microsoft.com/en-us/library/gg430857(v=pandp.40))
  + [MVVM快速入门](http://msdn.microsoft.com/en-us/library/gg430869(v=pandp.40))
  + [命令快速入门](http://msdn.microsoft.com/en-us/library/ff921082(v=pandp.40))
  + [UI组合快速入门](http://msdn.microsoft.com/en-us/library/ff921174(v=pandp.40))
  + [基于状态的导航快速入门](http://msdn.microsoft.com/en-us/library/gg405495(v=pandp.40))
  + [基于视图切换的导航快速入门](http://msdn.microsoft.com/en-us/library/gg430881(v=pandp.40))
  + [事件聚合快速入门](http://msdn.microsoft.com/en-us/library/ff921173(v=pandp.40))
  + [多目标工程快速入门](http://msdn.microsoft.com/en-us/library/ff921176(v=pandp.40))
* Appendix H: 动手做Prism
  + [WPF:开始Prism](http://msdn.microsoft.com/en-us/library/ff921141(v=pandp.40))
  + [Silverlight:开始Prism](http://msdn.microsoft.com/en-us/library/ff921096(v=pandp.40))
  + [部署WPF版Prism:通过ClickOnce发布和更新](http://msdn.microsoft.com/en-us/library/gg405497(v=pandp.40))
* [参考书目](http://msdn.microsoft.com/en-us/library/gg405487(v=pandp.40))
* [版权](http://msdn.microsoft.com/en-us/library/ff921171(v=pandp.40))

# Prism 4.1 自述

|  |
| --- |
| **索引:** |
| [**该版本更新内容**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec1) | [**该版本包含内容**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec2) | [**安装Prism**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec3) | [**Step 1: 安装系统环境**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec4) | [**Step 2: 解压Prism源码，库，和文档**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec5) | [**Step 3: 将Prism注册到Visual Studio**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec6) | [**文档**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec7) | [**Prism 4.1新增文档**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec8) | [**使用 InvokeCommandAction**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec9) | [**应用 ClearChildViewsRegionBehavior**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec10) | [**同步指定Regions**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec11) | [**快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec12) | [**基础MVVM快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec13) | [**MVVM快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec14) | [**模块化快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec15) | [**基于状态的导航 快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec16) |[**视图切换导航 快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec17) | [**UI组合快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec18) | [**命令快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec19) | [**事件聚合**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec20)**快速入门** | [**多目标快速入门**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec21) | [**HelloWorld**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec22)**快速入门** | [**参考实现**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec23) | [**MVVM RI**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec24) | [**Stock Trader RI**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec25) | [**更多**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec26) | [**UpdatePrismBinaries.bat**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec27)| [**已知问题**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec28) | [**将Prism**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec29)**源码添加到工程** | [**Prism 库和代码访问安全**](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#sec30) |

# 该版本更新内容

Prism 4.1的更新主要是升级到了Silverlight 5以及对CodePlex上确定的问题的修正。

基于Silverlight 5特性的更新有：

* **UpdateTextBindingOnPropertyChanged** Expression Blend行为(仅适用Silverlight)被标记为已过时，Silverlight 5现在提供了**UpdateSourceTrigger=PropertyChanged**。
* **DataTemplateSelector** (仅适用Silverlight) 被标记为已过时，Silverlight 5现在提供了隐式数据模版(implicit data templates)。
* **ButtonBaseClickCommand** 被标记为已过时。

CodePlex 问题：

* [CodePlex Issue 5495](http://compositewpf.codeplex.com/workitem/5495) – 追加 **InvokeCommandAction.**
* [CodePlex Issue 4052](http://compositewpf.codeplex.com/workitem/4052) – 更改类型约束**Control**为**UIElement**，在WPF的 **CommandBehaviorBase**和**InvokeCommandAction**类中。
* [CodePlex Issue 8061](http://compositewpf.codeplex.com/workitem/8061): 追加**SyncActiveStateAttribute** ，用于指定Region追加同步激活状态。
* [CodePlex Issue 7234](http://compositewpf.codeplex.com/workitem/7234): 取消**SubscriptionToken**
* [CodePlex Issue 3896](http://compositewpf.codeplex.com/workitem/3896): 追加**ClearChildViewsRegionBehavior**和**ClearChildViews**附加属性。用于清理View中的**RegionManager**。

其他更新：

* 库和工程的引用已经更新到最新通用版本。
* 新的NuGet包(4.1.0.0)。包括了面向.NET Framework 4.0, Silverlight 4 和5，以及 Windows Phone 7.1的包。注意面向Silverlight 4的版本是Prism 4.0.0.0。

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| 文档并不包括在这次更新的范围内，所以它仍指向4.0版本，除以上列出的项目，文档仍然是准确的。以上新项目的文档可以在该指南后面发现。 |

# 该版本包含内容

随Prism 4.1发布的有以下内容：

* 为Windows Presentation Foundation (WPF) 和 Silverlight署名
* 为Windows Phone 7署名
* 批处理文件：创建自定义Prism库
* 批处理文件：注册Prism库到Visual Studio
* Model-View-ViewModel 实现(MVVM RI)
* Stock Trader 实现(Stock Trader RI)
* Prism 4.0文档
* 快速入门:
  + Prism 4.0的快速入门:
    - 基础 MVVM快速入门
    - MVVM快速入门
    - 模块化快速入门 (MEF和Unity)
    - 基于状态的导航快速入门
    - 基于切换视图的导航快速入门
    - UI组合快速入门
  + 从Prism 2.x移植的快速入门:
    - 命令快速入门
    - 事件聚合快速入门
    - 多目标快速入门
    - Hello World 快速入门

除MVVM, 导航, 和UI组合的快速入门,每个快速入门都包括两套解决方案,一套面向WPF，另一套面向Silverlight。

本次更新不包含任何文档。"[Prism 4.0更新内容](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40))"仅包括Prism 2.1到4.0的变更。

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| 有关在Windows Phone 7上使用Prism的更多信息，请参阅 [Windows Phone 7开发人员指南](file:///C:\Users\615331\Dropbox\eBook\Patterns%20&%20Practices\Prism\Windows%20Phone%207开发人员指南)，位于CodePlex (<http://wp7guide.codeplex.com/>)。 |

# 安装Prism

本部分介绍如何安装Prism。它涉及以下三个步骤:

1. 安装系统环境
2. 解压Prism源码，库，和文档。
3. 注册Prism。

## Step 1: 安装系统环境

Prism目标平台是Microsoft Windows 7, Windows Vista,或Windows Server 2008。该版本已经在Windows XP Professional和Windows Server 2003上做过可用性测试,但并没有完成全面测试。生成本指南WPF应用需要.NET Framework 4.0, Silverlight应用需要Silverlight 5。

在你能使用Prism之前,必须安装以下:

* Microsoft .NET Framework 4.0 (随Visual Studio 2010安装)
* Microsoft Visual Studio 2010 Professional, Premium, 或Ultimate版

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| Visual Studio 2010 Express版同样能使用Prism库开发Prism应用。 |

如果你开发的是Silverlight应用,必须安装以下:

* [Visual Studio 2010的Silverlight 4工具](http://go.microsoft.com/fwlink/?LinkID=177428) (开发Silverlight必须;包括Silverlight开发运行时)

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| 虽然Visual Studio 2010的Silverlight工具不是必须的,但建议所有的WPF和Silverlight开发人员下载和使用最新版Visual Studio 2010的Silverlight工具Visual Studio的WPF和Silverlight设计器也随Silverlight开发运行时及SDK一起更新, 这也包括在了下载中。并且更新也追加了一些新特性以及修复了Bug。 |

可选，你可以考虑安装以下:

* [Microsoft Expression Blend 4](http://www.microsoft.com/expression/products/Blend_Overview.aspx).为WPF和Silverlight创建引人注目的用户体验的专业设计工具。
* [Windows Phone Developer Tools SDK](http://go.microsoft.com/fwlink/?LinkId=185968).用于Windows Phone 7.1开发。

## Step 2: 解压Prism源码, 库, 和文档

要安装Prism,对Prismv41.exe点击鼠标右键,然后选择以管理员身份运行。它将解压源码，库，和文档到你选择的文件夹中。

## Step 3: 将Prism注册到Visual Studio

注册Prism到Visual Studio并不是必须的,但是对于大多数开发者，这样做简化了在他们工程中引用Prism库的操作。如果你注册了Prism，它将在追加引用时，显示在Visual Studio追加引用对话框中**。**如果你不注册Prism，你需要手动将它追加引用到工程中。Prism签名程序集可以在下列文件夹中找到:

* {prism}\Bin\Desktop
* {prism}\Bin\Silverlight
* {prism}\Bin\Phone

要注册Prism,启动位于解压文件夹下的RegisterPrismBinaries.bat批处理文件。此批处理文件创建一个临时的.reg文件。此文件包括了注册到Desktop，Silverlight和Phone所需要的所有信息，以及使用它更新注册表。由于更新注册表需要权限，如果没有权限，在更新之前会出现一个UAC警告。想要获得更多关于UAC的信息,参阅"什么是用户账户控制(UAC)"在<http://windows.microsoft.com/en-US/windows7/What-is-User-Account-Control>.

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| 通常，用此脚本只会注册一个副本；如果注册了多份Prism副本，只有最后一个注册副本会在Visual Studio中可用。 |

# 文档

Prism包括以下文档:

* Prism4.chm是指南文档。
* Prism4APIReference-Desktop.chm是使用Prism创建WPF应用的参考文档。
* Prism4APIReference-Silverlight.chm是使用Prism创建Silverlight应用的参考文档。
* Prism4APIReference-Phone.chm是使用Prism创建Windows Phone 7应用的参考文档。

# Prism 4.1新增文档

## 使用InvokeCommandAction

虽然Expression Blend提供了**EventTrigger**和**InvokeCommandAction**，但是该操作不会更新控件的可用状态，如果**ICommand** 改变它的话。Prism的**InvokeCommandAction**监测ICommand的**CanExecuteChanged**事件，并相应的更新控件的可用状态。你可以用它来代替Expression Blend的**InvokeCommandAction**：

<TextBox>

<i:Interaction.Triggers>

<i:EventTrigger EventName="KeyUp">

<prism:InvokeCommandAction Command="{Binding MyCommand}" />

</i:EventTrigger>

</i:Interaction.Triggers>

</TextBox>

## 应用 ClearChildViewsRegionBehavior

在某些情况下，View包含的嵌套Region，当父View从一个Region中移除的同时，需要把它们从Region管理器中移除。为了支持这一需求，新增了一个可选的行为。该行为通过**RegionManager**的附加属性，强制将嵌套Region从Region管理器中移除。

虽然该行为已经注册，并对所有Region可用，但是它是可选的，通过使用附加属性**ClearChildViews**，使得它在指定View的嵌套Region上可用：

<UserControl x:Class="SomeModule.Views.RightView"

    ...

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

    RegionBehaviors:ClearChildViewsRegionBehavior.ClearChildViews="True">

...

## 同步指定Region

为了支持这样一种场景，父View中包含了指定Region，并且被指定的Region和View需要在父View被激活的时候，得到通知。需要解决这个问题，在指定Region中的View或ViewModel可以使用 **SyncActiveStateAttribute**确保获得激活通知。

[SyncActiveState]

public class MyViewModel : NotificationObject, IActiveAware

{

       (...)

}

# 快速入门

下面的文章将告诉你，在哪里可以找到Prism提供的指南和如何运行快速入门。

## 基础 MVVM快速入门

基础MVVM快速入门演示了如何创建一个非常简单的应用，实现MVVM模式。这将帮助你理解MVVM模式的基本概念。

基础MVVM快速入门实现了一个简单的调查应用。它可以显示不同类型的调查问题；完成调查问卷之后，可以提交；并且用户可以选择重置问题的答案。

**如何运行基础MVVM快速入门:**

1. 你必须完成 [安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism)。
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight Only - Basic MVVM QuickStart.bat**
3. 将BasicMVVMApp.Web工程设置为启动工程。
4. 将BasicMVVMAppTestPage.html设置为启动页面
5. 按下F5。

有关基础MVVM快速入门的详细信息，请查看[基础MVVM快速入门](http://msdn.microsoft.com/en-us/library/gg430857(v=pandp.40))。

## MVVM快速入门

MVVM快速入门演示了如何创建一个实现MVVM 用户界面(UI)的设计模式；它演示了一些更常见的编程任务，比如验证，交互和数据模版。

MVVM快速入门实现了一个简单的调查应用。它可以显示不同类型的调查问题；完成调查问卷之后，可以提交；并且用户可以选择重置问题的答案。

**如何运行MVVM快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight Only – MVVM QuickStart.bat**
3. 将MVVM工程设置为启动工程。
4. 按下F5。

有关MVVM快速入门的详细信息，请查看[MVVM快速入门](http://msdn.microsoft.com/en-us/library/gg430869(v=pandp.40))。

## 模块化快速入门

模块化快速入门演示了如何用Prism编写，发现，并初始化模块。此快速入门演示了一个模块化应用是如何发现和加载模块的，通过Prism支持的几种不同的方法，使用MEF和Unity作为组合容器。同时也演示了如何使用Prism 4.0的新的下载进度特性。

**如何运行使用MEF的WPF版模块化快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Desktop only - Open Modularity With Mef QuickStart.bat**
3. 将ModularityWithMef.Desktop工程设置为启动工程。
4. 按下F5。

**如何运行使用Unity的WPF版模块化快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Desktop only - Open Modularity With Unity QuickStart.bat**
3. 将ModularityWithUnity.Desktop工程设置为启动工程。
4. 按下F5。

**如何运行使用MEF的Silverlight版模块化快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight only - Open Modularity With Mef QuickStart.bat**
3. 将ModularityWithMef.Silverlight.Web工程设置为启动工程。
4. 按下F5。

**如何运行使用Unity的Silverlight版模块化快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight only - Open Modularity With Unity QuickStart.bat**
3. 将ModularityWithUnity.Silverlight.Web工程设置为启动工程。
4. 按下F5。

有关模块化快速入门的详细信息，请查看：

* [WPF模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921068(v=pandp.40))
* [Silverlight模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921163(v=pandp.40))

## 基于状态的导航快速入门

基于状态的导航快速入门演示如何使用视觉状态管理器实现应用导航。快速入门实现了一个聊天应用的子场景。应用启动后，显示用户的联系人列表。用户可以导航到他们联系人的其他视图：列表， 头像, 或者详细信息。当收到消息时，它们会通过显示通知。通过联系人详细信息视图，可以发送消息。

**如何运行基于状态的导航快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight only - Open QS - State-based Navigation QuickStart.bat**
3. 按下F5。

有关基于状态的导航快速入门的详细信息, 请查看[基于状态的导航快速入门](http://msdn.microsoft.com/en-us/library/gg405495(v=pandp.40)).

## 基于视图切换的导航快速入门

基于视图切换的导航快速入门演示如何使用Region导航APIs。快速入门实现了一个简单的集成了邮件，联系人和日历的应用。左边Region提供了导航到各个视图的功能。视图还演示了后退导航以及异步对话框交互。

|  |
| --- |
| **Gg405471.note(en-us,PandP.40).gif注意:** |
| 该快速入门没有演示如何整合Region导航和Silverlight导航API。但是， Silverlight导航API可以和Prism的Region导航API结合使用。 |

**如何运行基于视图切换的导航快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight only - Open QS - View-Switching Navigation QuickStart.bat**
3. 将ViewSwitchingNavigation.Web工程设置为启动工程。
4. 按下F5。

有关基于视图切换的导航快速入门的详细信息, 请查看[基于视图切换的导航快速入门](http://msdn.microsoft.com/en-us/library/gg430881(v=pandp.40)).

## UI组合快速入门

UI组合快速入门演示如何在UI组合中使用View发现和View注射。

**如何运行UI组合快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight only - Open QS - UI Composition QuickStart.bat**
3. 将**UIComposition.Web**工程设置为启动工程。
4. 按下F5。

有关UI组合快速入门的详细信息, 请查看[UI组合快速入门](http://msdn.microsoft.com/en-us/library/ff921174(v=pandp.40)).

## 命令快速入门

命令快速入门演示了如何使用Prism的命令，以解耦方式来处理WPF或Silverlight的UI操作。

**如何运行命令快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面其中一个批处理文件，在Visual Studio中打开工程： **Desktop & Silverlight - Open QS - Commanding QuickStart.bat** or **Desktop only - Open QS - Commanding QuickStart.bat**
3. 设置启动工程。要编译和运行WPF版快速入门,启动工程是Desktop解决方案文件夹下的**Commanding.Desktop**工程。要编译和运行Silverlight版快速入门,启动工程是Silverlight解决方案文件夹下的**Commanding.Silverlight**工程。
4. 按下F5。

有关命令快速入门的详细信息, 请查看[命令快速入门](http://msdn.microsoft.com/en-us/library/ff921082(v=pandp.40)).

## 事件聚合快速入门

事件聚合快速入门演示如何创建使用事件聚合服务的复合应用。这项服务可以建立应用中组件之间的松耦合通信。

**如何运行事件聚合快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面其中一个批处理文件，在Visual Studio中打开工程： **Desktop & Silverlight - Open QS - EventAggregator QuickStart.bat** 或 **Desktop only - Open QS - EventAggregator QuickStart.bat**
3. 设置启动工程。要编译和运行WPF版快速入门,启动工程是Desktop解决方案文件夹下的**EventAggregation.Desktop**工程。要编译和运行Silverlight版快速入门,启动工程是Silverlight解决方案文件夹下的**EventAggregation.Silverlight**工程。
4. 按下F5。

有关事件聚合快速入门的详细信息, 请查看[事件聚合快速入门](http://msdn.microsoft.com/en-us/library/ff921173(v=pandp.40)).

## 多目标快速入门

多目标快速入门演示了在WPF和Silverlight的多目标环境创建的项目的结构。它提供了桌面版 (WPF)和互联网富客户端版(RIA) (Silverlight)。

**如何运行多目标快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Desktop & Silverlight - Open QS - MultiTargeting QuickStart.bat**
3. 设置启动工程。要编译和运行WPF版快速入门,启动工程是Desktop解决方案文件夹下的**RealEstateListingViewer.Desktop**工程。要编译和运行Silverlight版快速入门,启动工程是Silverlight解决方案文件夹下的**RealEstateListingViewerHost** 工程。
4. 要编译和运行Silverlight版快速入门,右键点击位于**RealEstateListingViewerHost工程下的**RealEstateListingViewerTestPage.html，并**并设为启动页。**
5. 按下F5。

有关多目标快速入门的详细信息, 请查看[多目标快速入门](http://msdn.microsoft.com/en-us/library/ff921176(v=pandp.40)).

## Hello World 快速入门

Hello World 快速入门是动手做系列"[WPF:开始Prism](http://msdn.microsoft.com/en-us/library/ff921141(v=pandp.40))" 和"[Silverlight:开始Prism](http://msdn.microsoft.com/en-us/library/ff921096(v=pandp.40))"的起点解决方案。在该系列，你将学到Prism的基础概念，并学会如何运用他们创建一个Prism工程。可以以此为起点，创建一个复合的WPF或Silverlight应用。

**如何运行Hello World 快速入门:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Desktop only - Open QS - Hello World QuickStart.bat** or **Silverlight only - Open QS - Hello World QuickStart.bat**.
3. 按下F5。

# 参考实现

下面的文章将为你提供摘要，介绍如何运行参考实现(RI)。

## MVVM RI

MVVM RI应用是一个参考实现，演示了一个完整的调查应用，并演示如何解决开发人员在使用MVVM模式开发时面临的复杂挑战。主窗口列出了可用的调查问卷；当选中之后，将显示不同类型的问题。完成调查问卷后，可以提交。之后，用户返回到可用调查问卷列表。

**如何运行MVVM RI:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面的批处理文件，在Visual Studio中打开工程： **Silverlight Only - MVVM Reference Implementation.bat**
3. 将MVVM.Web工程设置为启动工程。
4. 按下F5。

有关MVVM RI的详细信息, 请查看[MVVM参考实现](http://msdn.microsoft.com/en-us/library/gg405492(v=pandp.40)).

## Stock Trader RI

Stock Trader RI应用是一个参考实现，演示了Prism应用的基准结构。在应用中，将看到如何处理开发人员在进行复合WPF应用开发时候所面临的挑战。比如通用，复用等。

Stock Trader RI应用演示了一个虚拟的，但是基于现实的金融投资场景。Contoso Financial Investments (CFI)是一个仿照真正的金融机构虚拟出的财务组织，CFI正在建立一个新的复合应用，提供给他们的股票交易员使用。

**如何运行Stock Trader RI:**

1. 你必须完成[安装Prism](http://msdn.microsoft.com/en-us/library/gg405471(v=pandp.40)#InstallingPrism).
2. 在Windows浏览器中，双击下面其中一个批处理文件，在Visual Studio中打开工程： **Desktop & Silverlight - Open RI - StockTrader Reference Implementation.bat**或
3. **Desktop only - Open RI - StockTrader Reference Implementaion.bat** (如果你没有安装Silverlight)
4. 如何运行WPF版Stock Trader RI, 将StockTraderRI工程 (位于Desktop解决方案文件夹) 设置为启动工程。
5. 如何运行Silverlight版Stock Trader RI, 将StockTraderRI工程(位于Silverlight解决方案文件夹)设置为启动工程。
6. 按下F5。

有关Stock Trader RI的详细信息, 请查看[Stock Trader 参考实现](http://msdn.microsoft.com/en-us/library/ff921074(v=pandp.40)).

# 更多

Prism社区位于<http://www.codeplex.com/Prism>。在社区里你可以提交问题，提供反馈意见，或者和其他用户共享想法。

## UpdatePrismBinaries.bat

UpdatePrismBinaries.bat批处理文件的目的是为了帮助你自动重编译Prism库，并把他们复制到{prism}\Bin文件夹。如果你用的是下载文件中的库，你不需要使用这个批处理文件。但是如果修改过源文件并想重编译并拷贝到{prism}\Bin文件夹，可以用这个批处理文件生成和复制程序集。

## 已知问题

* 用IE9 beta版运行Silverlight应用时，当应用第一次启动，有时IE9会报错。如果发生这种问题，点击IE9的刷新按钮，此后页面将会被正确加载。
* 运行RegisterAssemblies.bat批处理文件后,你可能需要重启Visual Studio，以便Prism的程序集引用显示出来。
* 在拥有Silverlight版和 WPF版的解决方案中, Visual Studio设计器会显示错误，如果没有选择正确的启动工程的话。当查看Silverlight代码时，设置Silverlight工程为启动工程。当查看WPF代码时，设置WPF工程为启动工程。
* 在Prism的Silverlight版工程上运行代码分析，会发生CA0055警告，如FxCop无法解析对旧版库的引用。请查看[connect issue 713608](http://connect.microsoft.com/VisualStudio/feedback/details/713608/ca0055-silverlight5-business-application-project)。Prism库引用的通用服务定位器和Unity仍指向Silverlight 4.

## 将Prism源码添加到工程

已编译的二进制包作为Prism的一部分发布， Prism库工程已经从所有的快速入门和参考实现解决方案的引用中删除。这是为了确保解决方案在重编译时，二进制包不会被不小心覆盖。如果需要将Prism源码添加到工程，需要完成下面几个步骤:

* **重新添加Prism库工程**.右键点击解决方案，选择**Add**,然后选择**Existing工程。**选择Prism库工程。然后为了防止无意识编译，在**Build**菜单中点击**Configuration Manager**，将所有Prism库工程的**Build复选框清除。包括**debug和release配置。
* **设置断点和逐句调试**.在你的应用的bootstrapper设置断点,然后逐句调试，进入基类的方法 (通常C#的快捷键是F11)。可能会被要求定位Prism库的源码,但通常情况下，应用的PDB文件是可用的，你只要在Prism库工程中打开源文件，直接在设置断点即可。

# Prism库和代码访问安全

Prism库使用所有默认的.NET框架的签名组件和代码访问安全方面的设置。因此建议工程的所有程序集都是强命名程序集, 包括Prism库程序集, shell程序集,和创建的所有模块。但这不是必须的。这样可以加载未签名的应用集到 (签名或未签名) Prism库应用。可以改变这种默认行为，通过应用.NET框架的安全策略，不允许使用未签名程序集，或者改变了信任等级的程序集。请注意.NET框架不允许加载部分受信任的程序集，除非你在Prism **库应用上追加AllowPartiallyTrustedCallers**属性

有关详细信息, 请查看MSDN上.NET框架开发指南的[代码访问安全](http://msdn.microsoft.com/en-us/library/930b76w0.aspx)。

Prism 4.0更新说明

|  |
| --- |
| **索引:** |
| [**新增教程**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec1) | [**更新教程**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec2) | [**Prism库的更新**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec3) | [**代码组织**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec4) | [**API 更新**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec5) - [*Bootstrapper API 的更新*](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec6), [*新增Prism 核心 API*](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec7) | [**示例**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec8)**代码更新** | [**非核心更新**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec9) | [**更多**](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40)#sec10) |

Prism 4.0包括几个部分的更新， Prism库中新的代码,新的和更新的快速入门,一个新的参考实现，以及文档的更新。部分Prism库从2.0升级到4.0，支持新的指南，解决已知问题，并应社区要求更新。

# 新增教程

Prism 4.0新增教程包括以下:

* 以Model-View-View Model (MVVM)模式创建Windows Presentation Foundation (WPF)和Microsoft Silverlight应用，在Chapter 5, "[实现MVVM模式](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))", 和Chapter 6, "[MVVM模式进阶](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))."进行了深入的讲解。
* 复合应用中的使用导航的两个方法：基于Region的导航和基于状态的导航在Chapter 7, "[UI组合](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))."进行了深入的讲解。
* 用来代替Unity的新依赖注入容器：Managed Extensibility Framework (MEF)在 Chapter 3, "[组件间的依赖管理](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40))."进行了深入的讲解。
* Prism的扩展点在Appendix E, "[扩展Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))."进行了讨论。
* Windows Phone 7的签名Prism库，包括Prism的一个子集 和 Prism.Interaction库，针对Windows Phone 7的开发。包括了Prism事件聚合, 命令, view model, 和交互请求。有关Prism Phone库的详细信息，请查看Windows Phone 7开发人员指南网站： <http://wp7guide.codeplex.com/>.

# 更新教程

Prism文档已经重新组织，所以你可以很容易的找到某个主题。Prism重要主题是按照章节组织的。每章包括你应该考虑的关键决策。关于参考实现，动手做，和快速入门的文档已经被移到附录。

# Prism库的更新

Prism 4.0包括了多项更新，包括新的机能，代码组织和API更新。

## 代码组织

对Prism库进行了以下代码组织上的更新:

* Composite Application库被重命名为Prism库。
* 命名空间**Composite和Composite.Presentation已被删除，这两个命名空间合并成了Microsoft.Practices.Prism**.
* Microsoft.Practices.Prism库，用于Silverlight和WPF， 命名空间**Microsoft.Practices.Prism.Regions**, **Microsoft.Practices.Prism.Commands**, and**Microsoft.Practices.Prism.ViewModel**被统一到http://www.codeplex.com/prism的**xmlns**定义.
* 几个基于UI的行为被提取到**Prism.Interactivity应用集，包括交互请求行为。**
* 现在已经可以用MEF作为依赖注入容器了。此功能需要两个新工程: **Prism.MefExtensions.Desktop**和**Prism.MefExtensions.Silverlight。这两个工程创建了一个新的程序集，** Microsoft.Practices.Prism.MefExtensions.dll,分为Desktop和Silverlight版。同时在解决方案中还包括了新功能的单元测试。

## API更新

Prism库的API有几个关键更新。Bootstrapper已经被大量修改，并且为了支持新功能，追加了部分可重用的代码。

### Bootstrapper API的更新

由于追加了MEF的支持，使得Prism的bootstrapper发生了一些变化。这些变化包括以下:

* 新添加了一个**Bootstrapper基类,位于Microsoft.Practices.Prism**命名空间下，分为**Prism.Desktop**和**Prism.Silverlight**版本.
  + **UnityBootstrapper**类(在**Microsoft.Practices.Prism.UnityExtensions.Desktop** 和**Microsoft.Practices.Prism.UnityExtensions.Silverlight**命名空间下)扩展自**Bootstrapper**基类.
  + **MefBootstrapper类** (在**Microsoft.Practices.Prism.MefExtensions.Desktop** 和**Microsoft.Practices.Prism.MefExtensions.Silverlight**命名空间下)扩展自**Bootstrapper**基类.
* 伴随**Bootstrapper的更改以下的**属性和方法被更改为统一命名:
  + **LoggerFacade属性被重命名为Logger**.
  + **获取Logger方式被移到了**bootstrappers之外的方法中,以**CreateLogger方法的返回值获得，而不是Logger的get。**
  + **Bootstrapper**中的**GetModuleCatalog方法被重命名为CreateModuleCatalog**.
* 增加了几个新的方法，以便概念更好的区分:
  + 作为引导过程的一部分，增加了虚拟方法**ConfigureModuleCatalog,**允许在创建之后修改模块分类。
  + 增加了虚拟方法**ConfigureServiceLocator,允许重写ServiceLocator的配置。**
* **ModuleManager现在增加了ModuleDownloadProgressChanged和LoadModuleCompleted事件，用来监控模块下载的进度，以及确定模块何时下载完毕。**

有关bootstrapper类的详细信息，请查看Chapter 2, "[初始化Prism应用](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40))."关于bootstrapper完整的API说明，请查看Prism 4 API 帮助文档的 **UnityBootstrapper**和**MEFBootstrapper**。

### 新增Prism核心API

以下命名空间被追加到Prism库中，以支持Prism 4.0的变化:

**加入Microsoft.Practices.Prism.ViewModel，以帮助开发基于**MVVM的应用。

**Microsoft.Practices.Prism.Regions包括几个新的类，**以帮助开发基于region的应用。

Prism引入新的行为**RegionMemberLifetimeBehavior，用于从region中移除已停用的view。**view或者view model可以选择使用**RegionMemberLifetimeAttribute**或者**IRegionMemberLifetime**接口来实现。有关更多详细信息,请查看Prism 4 API **帮助文档的RegionMemberLifetimeBehavior**。

# 示例代码更新

Prism 4.0提供了许多示例来演示指南教程。 几个Prism 2.0版的示例被删除或者更新，并新增了一些新的示例。

Prism 4.0新增了以下示例代码:

* **MVVM基础快速入门**. 该快速入门演示了一个非常简单的MVVM应用，并覆盖了MVVM的核心概念. 更多详细信息, 请查看[MVVM基础快速入门](http://msdn.microsoft.com/en-us/library/gg430857(v=pandp.40))**.**
* **MVVM快速入门**. 该快速入门扩展自MVVM基础快速入门，并演示了更加复杂的场景。 更多详细信息, 请查看[MVVM快速入门](http://msdn.microsoft.com/en-us/library/gg430869(v=pandp.40)).
* **MVVM参考实现**. 该参考实现演示了实际应用中面临的许多真实场景。该示例演示了一个更复杂更全面的MVVM场景。更多详细信息, 请查看[MVVM参考实现](http://msdn.microsoft.com/en-us/library/gg405492(v=pandp.40)).
* **基于视图切换的导航快速入门**. 该快速入门演示了如何使用Prism region导航API. 更多详细信息, 请查看[基于视图切换的导航快速入门](http://msdn.microsoft.com/en-us/library/gg430881(v=pandp.40)).
* **基于状态的导航快速入门**. 该快速入门演示了如何使用Silverlight的 Visual State Manager定义views (状态)和如何切换。更多详细信息,请查看[机遇状态的导航快速入门](http://msdn.microsoft.com/en-us/library/gg405495(v=pandp.40)).
* **UI组合快速入门**. 该快速入门演示如何用Prism 4版本的View注入快速入门和View 发现快速入门取代Prism 2.0版。所有的概念都包括在一个应用中。更多详细信息, 请查看[UI组合快速入门](http://msdn.microsoft.com/en-us/library/ff921174(v=pandp.40)).
* **模块化快速入门**. 有四个新的快速入门演示了如何为WPF版和Silverlight版实现同样的应用，通过使用Unity和MEF. 有四个新的快速入门覆盖了Prism 2.0版本的快速入门 (模块化配置快速入门, 通过编码定义模块快速入门, 查找目录中的模块快速入门, 和加载远程模块快速入门). 更多详细信息, 请查看[WPF模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921068(v=pandp.40)) 和 [Silverlight模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921163(v=pandp.40)).

Stock Trader参考实现已升级，使用了MVVM模式和导航的概念。并且已经用MEF取代了Unity作为依赖注入容器。

以下的示例已经从Prism中移除，因为他们已经被新的示例取代：UI组合快速入门, WPF模块化快速入门,和模块化快速入门：Silverlight版:

* View 注入快速入门
* View 发现快速入门
* 模块化配置快速入门
* 通过编码定义模块快速入门
* 查找目录中的模块快速入门
* 加载远程模块快速入门

# 非核心更新

Prism 4.0的另外一些更新,包括:

* Prism许可更新为Microsoft patterns & practices license.更多详细信息，请查看<http://compositewpf.codeplex.com/license>。该许可同样包括在位于Prism根目录的EULA.txt文件中。
* 具有强命名和验证码签名的Prism库现在包括在Prism 4.0中，并位于bin目录下。所有使用Prism库的工程，都在目录{prism}\Bin\{platform}下调用它。
* 你可以使用UpdatePrismBinaries.bat批处理文件重编译Prism库。该批处理文件位于你解压的Prism目录下。

|  |
| --- |
| **Gg430871.note(en-us,PandP.40).gif注意:** |
| 这将覆盖{prism}\Bin\ 文件夹下的文件。 |

* 你可以使用RegisterPrismBinaries.bat批处理注册Prism库到Visual Studio。该批处理文件位于你解压的Prism目录下。
* Moq库现在包含在Prism中，并位于以下目录:
  + Moq：WPF版位于{prism}\Lib\Desktop\Moq文件夹。
  + Moq：Silverlight版位于{prism}\Lib\Silverlight\Moq文件夹。

# 更多

更多详细信息关于如何将解决方案从2.x升级到4.0, 请查看[从Prism 2.x](http://msdn.microsoft.com/en-us/library/gg430859(v=pandp.40)) 升级.

更多详细信息关于Moq,请查看<http://code.google.com/p/moq/>.

# 1: 介绍

|  |
| --- |
| **索引:** |
| [**为什么要使用Prism?**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec1) | [**客户端应用开发面临的挑战**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec2) | [**综合方法**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec3) | [**Prism**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec4)**不能解决的部分** |[**开始Prism**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec5) | [**预备知识**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec6) | [**安装Prism**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec7) - [*Step 1: 安装系统环境*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec8), [*Step 2: 解压Prism 源码，库，和文档*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec9), [*Step 3: 注册Prism*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec10) | [**通览 Prism**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec11) – [*此发行版中的新功能?*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec12), [*安装包中的内容?*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec13), [*通览文档*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec14), [*通览快速入门*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec15), [*通栏参考实现*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec16) | [**从早期版本升级**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec17) | [**Prism概述**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec18) | [**Prism设计目标**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec19) | [**Prism关键概念**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec20) | [**使用Prism**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec21) – [*定义Shell*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec22), [*创建Bootstrapper*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec23), [*创建Module*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec24), [*添加View模块到Shell*](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec25) | [**更多**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec26) | [**社区**](http://msdn.microsoft.com/en-us/library/ff921153(v=pandp.40)#sec27) |

Prism提供指南，帮助你更轻松的设计和创造丰富的，灵活，易维护的WPF桌面应用， Silverlight富互联网应用(RIAs),和Windows Phone 7应用。使用设计模式，比如分离模式和使用松耦合模式。Prism可以帮助你设计和创建应用，使用可以独立开发维护的松耦合组件，但又可以轻松无缝的集成到一个应用中。这种类型的应用被称为复合应用。

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| Prism的前身是**WPF和Silverlight的复合应用指导**.为简洁起见，并应用户要求，该指南现在已经重命名为**Prism。** |

Prism用于开发者构建WPF或Silverlight应用。这些应用通常有多场景，丰富的交互体验和数据可视化，以及体现业务逻辑。这些应用通常有多个后台系统和服务，采用分层架构，可能跨多物理层部署。并且这些应用很有可能在其生命周期中，不断追加新的需求和业务要求。总之，这些应用是"变到死"和"建到死"的。如果应用没有这样的要求，可能使用Prism并不能带来好处。

Prism包括参考实现, 快速入门,可重用库(Prism库),和丰富的文档。该版本目标平台是Microsoft .NET Framework 4.0及Silverlight 4,包括新的指南，围绕MVVM模式,导航,和MEF。因为Prism是基于.NET Framework 4.0 (包括WPF)和Silverlight 4,熟悉这些技术对使用和熟悉Prism非常有用。

但是应注意到:虽然Prism不难学,但是开发者应该理解和接受新的开发模式和方法。使用Prism时，管理是非常重要的。在项目期限之前，前期就必须适应这些新的开发模式和方法。

# 为什么要使用Prism?

设计和开发易扩展，易维护的WPF或Silverlight的富客户端会很困难。本章将演示一些在开发WPF或Silverlight的富客户端时会面临的挑战，并演示如何使用Prism来应付这些挑战。

## 客户端开发面临的挑战

通常情况下，客户端应用开发人员面临着不少挑战。随着时间的推移，应用的需求会改变。新的商机会带来挑战，新的技术运用会带来挑战，或者在开发过程中客户的反馈同样会带来许许多多的需求变更。因此，开发使应用易扩展，易维护是非常重要的。设计这种易扩展的应用很难做到，它要求有一个架构，允许独立开发和测试应用的各个部分，并且在随后的更新或修改都是独立的，不会影响应用的其它部分。

大多数企业应用足够复杂，需要多名开发人员，甚至是大型团队的配合。包括UI设计师和本地化团队。如何设计应用，令开发人员或小组可以有效的开发应用独立部分，并确保模块无缝集成到应用中，是一项重大的挑战。

以一体化 风格设计和开发应用会导致应用开发效率低下，维护困难。在这种情况下，"一体化"中的组件是紧耦合的，没有明确的区分各个组件。通常这种设计和开发会令开发人员非常难过。它很难向应用中追加新的功能或者替换现有功能。很容易在解决一个Bug的同时导致另一部分产生另一个Bug。很难测试和部署。同事，它还很难让开发者和设计者有效的协同工作。.

## 综合方法

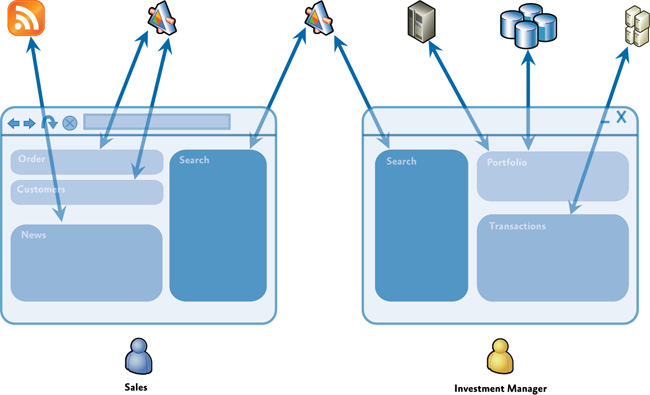
对这些挑战的有效补救办法是把应用划分为多个独立的，松散的，半独立的组件。以便轻松的集成到应用的"shell"，组成一个解决方案。设计和创建应用的这种方法通常被称为复合应用。

复合应用有很多好处，包括以下：

* 允许模块的独立开发，测试和发布;同样可以更容易的追加新方法，修改或者扩展，从而使应用更易于扩展和维护。注意：这是单人项目的经验，对于创建一个复合应用所获得的更容易测试和维护的好处。
* 提供了一个通用的应用壳，UI组件可以松耦合的方式结合在一起。这降低了多个开发人员之间在追加新功能到应用的时候，对共用同一UI所产生的冲突，并且这样可以提供一个统一的外观。
* 促进重用和使得应用在划分功能时更清晰。比如水平划分，按日志，身份验证。或者垂直划分的按特定业务机能划分。这可以更轻松的管理组件之间的依赖和交互。
* 有助于保持角色分离，允许不同的个人或团队，根据他们的经验和特长，专注于一个特定的任务或功能。特别是UI和业务逻辑之间的分离，这意味着UI设计师可以专注于创建更丰富的用户体验。

符合应用非常适合用于客户端应用开发。比如，希望创建一个用户体验丰富的前端，而后台使用不同的系统。下图显示了这种类型的复合应用的例子。

**多后台系统的复合应用**



在这类应用中，用户可以获得一个丰富灵活的用户体验，面向任务的，跨多后台系统，服务，和数据存储。各部分由一个或多个独立模块提供。应用UI和业务逻辑之间划分清晰，允许应用提供统一风格又可以定制的外观。

此外，复合应用当发生如下情况时非常有用，UI中有组件发生重大改变，并且这些往往是不通团队开发维护，需要融合在一起时。下图显示了这种类型的应用的屏幕截图。每个高亮区域代表组成的不同UI组件。

**Stock Trader参考实现复合应用**



这种情况下，复合应用允许UI动态组成。这提供了一个灵活的用户体验。比如它可以让新功能在运行时动态追加，使用户可以自由的定制和扩展终端。

## Prism不能解决的部分

虽然Prism可以帮助解决许多在创建WPF或Silverlight应用时面临的问题，但是根据应用场景和要求，可能会面临更多的挑战。比如， Prism没有覆盖以下主题:

* 脉冲链接和数据同步
* 服务和消息传递基础结构设计
* 身份验证和授权
* 应用性能
* 应用版本
* 错误处理和容错能力

# 开始Prism

## 预备知识

Prism假设你有使用WPF或Silverlight的实际经验。有几个重要概念在Prism中大量使用。你应该熟悉它们。这些概念包括以下的:

* **XAML (可扩展应用程序标记语言)**。 语言以声明方式定义和初始化WPF和Silverlight应用的用户界面。
* **数据绑定**. WPF和Silverlight应用中，UI元素与控件或数据进行链接的方式。
* **资源**. WPF和Silverlight应用中，用于创建和管理样式，数据模板，控件模板的方式。
* **命令**. 用于控件和用户手势或输入进行链接的方式。
* **用户控件**. 用于提供自定义行为或自定义外观。
* **依赖属性**. 公共语言运行库(CLR)的扩展，用于支持数据绑定，路由命令和事件的设置和监测。
* **行为.** 行为是封装了的交互功能，可以轻松应用到UI的控件的对象。

## 安装Prism

本章介绍如何安装Prism。包括下列三个步骤:

1. 安装系统环境。
2. 解压Prism源码，库，和文档。
3. 注册Prism。

### Step 1:安装系统环境

Prism目标平台是Microsoft Windows 7, Windows Vista,或Windows Server 2008。该版本已经在Windows XP Professional和Windows Server 2003上做过可用性测试,但并没有完成全面测试。生成本指南WPF应用需要.NET Framework 4.0, Silverlight应用需要Silverlight 5。

在你能使用Prism之前,必须安装以下:

* Microsoft .NET Framework 4.0 (随Visual Studio 2010安装)
* Microsoft Visual Studio 2010 Professional, Premium, 或Ultimate版

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| Visual Studio 2010 Express版同样能使用Prism库开发Prism应用。 |

如果你开发的是Silverlight应用,必须安装以下:

* [Visual Studio 2010的Silverlight 4工具](http://go.microsoft.com/fwlink/?LinkID=177428) (开发Silverlight必须;包括Silverlight开发运行时)

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| 虽然Visual Studio 2010的Silverlight工具不是必须的,但建议所有的WPF和Silverlight开发人员下载和使用最新版Visual Studio 2010的Silverlight工具 Visual Studio的WPF和Silverlight设计器也随Silverlight开发运行时及SDK一起更新, 这也包括在了下载中。并且更新也追加了一些新特性以及修复了Bug。 |

可选，你可以考虑安装以下:

* [Microsoft Expression Blend 4](http://www.microsoft.com/expression/products/Blend_Overview.aspx). 为WPF和Silverlight创建引人注目的用户体验的专业设计工具。
* [Windows Phone Developer Tools SDK](http://go.microsoft.com/fwlink/?LinkId=185968). 用于Windows Phone 7.1开发。

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| 有关在Windows Phone 7上使用Prism的更多信息，请参阅[Windows Phone 7开发人员指南](file:///C:\Users\615331\Dropbox\eBook\Patterns%20&%20Practices\Prism\Windows%20Phone%207开发人员指南) 位于CodePlex (<http://wp7guide.codeplex.com/>.) |

### Step 2: 解压Prism源码，库，和文档

要安装Prism,对Prismv41.exe点击鼠标右键,然后选择**以管理员身份运行**。它将解压源码，库，和文档到你选择的文件夹中。

### Step 3: 注册Prism

注册Prism到Visual Studio并不是必须的,但是对于大多数开发者，这样做简化了在他们工程中引用Prism库的操作。如果你注册了Prism，它将在追加引用时，显示在Visual Studio追加引用对话框中**。**如果你不注册Prism，你需要手动将它追加引用到工程中。Prism签名程序集可以在下列文件夹中找到:

* {prism}\Bin\Desktop
* {prism}\Bin\Silverlight
* {prism}\Bin\Phone

要注册Prism,启动位于解压文件夹下的RegisterPrismBinaries.bat批处理文件。此批处理文件创建一个临时的.reg文件。此文件包括了注册到Desktop，Silverlight和Phone所需要的所有信息，以及使用它更新注册表。由于更新注册表需要权限，如果没有权限，在更新之前会出现一个UAC警告。想要获得更多关于UAC的信息,参阅"[什么是用户账户控制(UAC)](http://windows.microsoft.com/en-US/windows7/What-is-User-Account-Control)."

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| 通常，用此脚本只会注册一个副本；如果注册了多份Prism副本，只有最后一个注册副本会在Visual Studio中可用。 |

## 通览Prism

### 此发行版本中的新功能?

此发行版本的Prism已将目标WPF和Silverlight版本更新为4,并且它包含了一些新的和更新的指南，包括以下:

* **Managed Extensibility Framework (MEF)**. Prism现在已经支持使用MEF管理依赖及松耦合组件间的协作。如何使用MEF请查阅Chapter 3, "组件间依赖管理."
* **Model-View-View Model (MVVM) 模式**. 旧版本的Prism已经提供了基于分离separated presentation模式的开发指南。此版本已升级为MVVM模式. Chapter 5演示了MVVM模式概念和描述如何实现它。Chapter 6介绍MVVM模式进阶场景。
* **导航**. Prism现在提供如何在WPF和Silverlight应用中实现导航。该指南包括了基于状态的导航,用于更新单一view状态的场景，以及基于视图切换的导航，用于在多view间切换的场景。两种方法都位于Chapter 8, "导航."

Prism现在提供了二进制版本。这使得想在自己应用中使用Prism的开发人员无需编译而直接使用。此外，所有 Visual Studio 项目 （Prism的参考实现和快速入门）也已迁移到Visual Studio 2010 和 Silverlight 4上。

有关新功能，资源和API的更新列表，请查阅 [Prism 4.0](http://msdn.microsoft.com/en-us/library/gg430871(v=pandp.40))更新内容.

### 安装包中的内容?

Prism包括以下：

* **Prism库源码**. Prism库源码包括核心Prism功能，以及Unity和MEF扩展。使得[Unity](http://msdn.microsoft.com/en-us/library/dd203101.aspx)和[MEF](http://msdn.microsoft.com/en-us/library/dd460648.aspx)可以用于Prism。
* **Prism二进制包**. Prism签名二进制包位于bin文件夹下，让开发者可以很容易的使用Prism库。Prism库也可以重编译和注册到Visual Studio，通过提供的脚本文件。二进制包同时也包括了[Unity Application Block](http://msdn.microsoft.com/en-us/library/dd203101.aspx) 和[Service Locator](http://commonservicelocator.codeplex.com/)包。
* **参考实现**. 综合参考实现说明如何使用Prism实现真实的应用场景。参考实现并不是完整的，但是说明了如何使用Prism模式合成一个统一的应用。Prism提供了两个参考实现: Stock Trader RI和MVVM RI.
* **快速入门**. Prism包括几个小型的专题示例源码。包括MVVM模式,导航, UI组合,模块化,命令,事件聚合,以及多目标。
* **文档**. Prism 4文档提供了一个关于Prism的目标和概念的概述。以及详细描述了如何使用Prism提供的每个功能及设计模式。下一节将提供一个逐章的概述。

### 通览文档

Prism文档涵盖了广泛的话题,包括：共同开发及复合应用开发的挑战的概述，Prism库和它实现的设计模式的概述，以及如何在开发中使用Prism库的分步解说的概述。文档希望能使读者很好的理解及将Prism应用到他们的应用中。文档包括以下:

* Chapter 2, "[初始化Prism](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40))." 本章讨论如何让一个模块化Prism应用运行起来。
* Chapter 3, "[组件间依赖管理](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40))." 基于Prism的应用需要一个依赖注入容器。虽然在Prism中可以使用任何依赖注入容器，但是Prism库提供了两个默认的选择: Unity或MEF.本章讨论了使用不通的依赖注入容器时可获得的功能以及需要考虑的问题。
* Chapter 4, "[模块化应用开发](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40))." 本章讨论了在创建一个使用Prism的模块化客户端应用时，需要考虑的核心概念，关键决策，以及核心场景。
* Chapter 5, "[实现MVVM模式](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))." 使用MVVM模式，你可以将应用的UI，底层，以及业务逻辑分为三个独立的类:view, model,和view model。本章讨论了MVVM模式背后的核心概念，并介绍了如何在你的Prism应用中实现。
* Chapter 6, "[MVVM进阶模式](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))." 本章提供了使用MVVM模式实现更复杂场景的指南,包括如何实现复合命令(命令群集),和如何处理异步Web服务和用户交换。本章还提供使用依赖注入容器(比如Unity或MEF)创建和包装成MVVM模式的指南。
* Chapter 7, "[UI组合](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))". Regions是占位符，允许开发者在应用中指定视图显示的位置。在Prism中,有两种方式指定视图显示在region中:view发现和view注入。本章讨论了如何使用region和UI。同样关于UI设计中，如何理解复合应用的指南。
* Chapter 8, "[导航](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))." 导航是用户和应用交互或者应用内部状态变更的结果，在应用UI上的体现。本章提供了相应的指南，包括如何实现基于状态的导航，当视图的状态发生改变时在UI导航上的反映。以及基于视图切换的导航,创建一个新的视图并显示在region中。
* Chapter 9, "[松耦合组件间的通讯](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40))." 本章讨论了在不同模块之间的组件间通讯，通过使用命令，事件聚合，Region上下文，和共享服务。
* Chapter 10, "[Silverlight和WPF之间共享代码](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40))." 多目标代码是指针对两个不同平台，具有大致相同的代码。本章指的两个平台是WPF和Silverlight。本章帮助你理解什么是多目标，它的优点和缺点。Prism提供了[项目连接器](http://visualstudiogallery.msdn.microsoft.com/en-us/5e730577-d11c-4f2e-8e2b-cbb87f76c044) 工具帮助你在WPF和Silverlight平台上，自动创建和维护链接，来自源项目和目标项目的共通代码，。
* Chapter 11, "[部署Prism应用](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40))." 本章涉及部署使用Prism的 WPF和Silverlight应用的注意事项。
* Appendix A, "[词汇表](http://msdn.microsoft.com/en-us/library/ff921135(v=pandp.40))." 本附录提供Prism库中使用的术语，概念，设计模式和功能的简明摘要。
* Appendix B, "[Prism中的设计模式](http://msdn.microsoft.com/en-us/library/ff921146(v=pandp.40))." 本附录描述了Prism及Stock Trader RI中应用的软件设计模式。本章主要针对架构师和开发人员，使之理解和熟悉Prism中的设计模式，用于解决在创建复合应用时面临的挑战。
* Appendix C, "[Prism库](http://msdn.microsoft.com/en-us/library/gg405476(v=pandp.40))." 本附录提供了对Prism库的概述。

下列主题均包括源码和在线资源:

* Appendix D, "[从早期版本升级](http://msdn.microsoft.com/en-us/library/ff921073(v=pandp.40))." 本附录讨论了如果你是从早期版本升级的话，需要知道的事情。
* Appendix E, "[扩展Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))." 本附录讨论如何扩展Prism模块，行为和导航。
* Appendix F, "[参考实现](http://msdn.microsoft.com/en-us/library/gg405483(v=pandp.40))." 本附录介绍了Prism中的参考实现。更多详细信息,请查阅"通览参考实现."
* Appendix G, "[快速入门](http://msdn.microsoft.com/en-us/library/gg430879(v=pandp.40))." Prism包括几个关键概念的快速入门。更多详细信息, 请查阅下一节, "通览快速入门."
* Appendix H, "[动手做Prism](http://msdn.microsoft.com/en-us/library/gg405475(v=pandp.40))." 本附录演示了如何创建一个简单的复合应用，循序渐进的，在WPF和Silverlight平台上。本附录主要针对希望理解Prism库基础概念的开发者。同样，也演示了如何通过ClickOnce发布和更新WPF应用。

### 通览快速入门

快速入门很小,但是说明了Prism相关重点概念。如果你想理解一个概念，快速入门是一个理想的起点。通过源代码学习一个新的技术，是最合适不过的了。Prism包括以下的快速入门:

* [模块化快速入门：WPF](http://msdn.microsoft.com/en-us/library/ff921068(v=pandp.40))和[模块化快速入门：Silverlight](http://msdn.microsoft.com/en-us/library/ff921163(v=pandp.40)). 该快速入门演示如何创建WPF和Silverlight应用中的模块。这些模块可以静态加载,当shell包括该模块的引用，或者动态加载,当模块在运行时被动态发现和加载。该快速入门演示了如何使用Unity和MEF容器。
* [MVVM基础快速入门](http://msdn.microsoft.com/en-us/library/gg430857(v=pandp.40))和[MVVM快速入门](http://msdn.microsoft.com/en-us/library/gg430869(v=pandp.40)). MVVM基础快速入门演示如何生成一个非常简单的MVVM模式的应用。MVVM快速入门演示如何生成一个更复杂的MVVM模式的应用，演示开发者实际面临的更常见的挑战，比如验证，UI交互，和数据模板。
* [UI组合快速入门](http://msdn.microsoft.com/en-us/library/ff921174(v=pandp.40)). 该快速入门演示如何创建WPF和Silverlight的UI组合，将不同的视图加载到region中，演示了视图之间解耦的方式。阐述了如何在UI组合中使用view发现和view注入功能。
* [基于状态的导航快速入门](http://msdn.microsoft.com/en-us/library/gg405495(v=pandp.40)). 该快速入门演示了定义一个简单应用的导航的方法。在该快速入门中使用的方法是：使用Silverlight Visual State Manager (VSM)，定义应用的不通状态以及这些状态对应的动画，然后在不同状态之间切换。
* [基于视图切换快速入门](http://msdn.microsoft.com/en-us/library/gg430881(v=pandp.40)). 该快速入门演示如何使用Prism Region Navigation API.该快速入门演示多导航场景，包括：导航到某Region中的视图，导航到某Region中包含在另一个视图中的子视图(嵌套导航)，导航日志支持，限时存在视图创建，导航到视图时传递上下文信息,在导航中的View和View Model,以及通过模块化和UI组合，将导航作为应用的一部分。
* [命令快速入门](http://msdn.microsoft.com/en-us/library/ff921082(v=pandp.40)). 该快速入门演示如何创建WPF或Silverlight UI，通过使用Prism提供的命令，以分离方式处理UI操作。
* [事件聚合快速入门](http://msdn.microsoft.com/en-us/library/ff921173(v=pandp.40)). 该快速入门演示如何创建WPF或Silverlight应用，通过使用事件聚合服务。该服务可在应用组件间创建松耦合通讯。
* [多目标快速入门](http://msdn.microsoft.com/en-us/library/ff921176(v=pandp.40)). 该快速入门演示WPF和Silverlight环境下创建的多目标项目的结构。演示了在桌面(WPF)和富互联网应用(RIA) (Silverlight)上的使用。

### 通览参考实现

Prism的参考实现，是基于现实中客户所面临的挑战制作的应用。当查看这些应用时，应当知道是使用Prism构建实现的。 Prism参考实现包括:

* [Stock Trader参考实现](http://msdn.microsoft.com/en-us/library/ff921074(v=pandp.40)). Stock Trader RI演示了如何使用Prism库的基准体系结构实现复合应用。
* [MVVM参考实现](http://msdn.microsoft.com/en-us/library/gg405492(v=pandp.40)). MVVM RI演示了使用MVVM模式时，开发者所要面临的复杂挑战。

## 从早期版本升级

如果你是从早期版本升级而来，你应该查看Appendix D 的"[从早期版本升级](http://msdn.microsoft.com/en-us/library/ff921073(v=pandp.40))"，以了解此发行版本和早期版本之间的主要区别。

如果你是从复合UI应用块(针对WinForms)升级到Prism,你应该查看Appendix D的"[从复合UI应用块升级](http://msdn.microsoft.com/en-us/library/ff921081(v=pandp.40))"，以了解如何从复合UI应用块映射到Prism库。

# Prism概述

## Prism设计目标

Prism旨在帮助你设计丰富，灵活，且易于维护的WPF和Silverlight应用。Prism库实现的设计模式，体现了关键功能的松耦合分离。使用设计模式和Prism提供的功能，可以设计和构建使用松耦合组件的应用，不但可以独立开发维护某一部分，又可以轻松无缝的集成到一个应用中。

Prism的设计是围绕着关键功能的松耦合分离的设计原则来实现的。这使得Prism提供了许多便利功能，包括以下:

* **重用**. Prism的组件和服务可以很方便的开发，测试及集成到一个或多个应用中，从而促进重用。在组件级别，那些通过了单元测试的组件，在运行时，通过依赖注入可以很容易的实现发现和注入，从而实现重用。在应用级别，通过封装模块，可以实现跨应用的重用。
* **可扩展性**. Prism可以开发易扩展的应用，通过管理组件依赖，允许组件在运行时更容易的集成或者替换。并且提供分解应用成模块的能力，使得模块可以独立更新和部署。Prism库中的许多模块，本身也能扩展或替换。
* **灵活性**. Prism帮助创建灵活的应用，使他们更容易的更新，通过新功能的开发和集成。Prism还允许WPF和Silverlight应用的开发中，使用共通服务和组件，允许以最优方式部署和使用。还允许应用提供不通的体验，基于角色或者配置。
* **团队开发**. Prism促进团队开发，允许不同的团队独立的开发，甚至部署应用的不同部分。 Prism有助于最大限度的减少跨团队的依赖，允许团队专注于不通的功能(比如UI设计，业务逻辑实现，和核心代码)，或者不同的业务(比如档案，销售，库存或者物流)。
* **质量**. Prism可以帮助提高应用的质量，通过给提供给团队使用的共通服务和组件提供完整测试。此外，Prism对通用设计模式的实现提供了完整的测试，以及完整的指南，因此， Prism允许开发团队专注于他们的应用，而不是执行测试基础结构代码。

值得注意: Prism的设计是，你可以单独，或者一起使用Prism的功能和设计。由需求和应用场景所决定。并且，可以循序渐进的使用，根据不同的开发时期，而不需要对应用进行大规模的结构性转变。

最后，因为软件测试应该作为开发启动之后的首要任务来考虑，并且和开发进程紧密结合，Prism提供各种软件测试的广泛支持，使得设计和应用可以非常容易的测试。Prism本身的开发，也是测试先行，它的开发满足多个严格的质量测试，以确保符合Microsoft安全标准，并确保它可以在多个操作系统上，多个版本的Visual Studio上，和使用多重编程语言结合正常运作。每次代码签入后，都会进行单元测试。此外，Prism库的测试，针对额外的质量等级进行了测试，如下：

|  |  |
| --- | --- |
| 测试 | 描述 |
| 验收测试 | 测试应用功能，通过用户场景驱动测试。测试可以手动或者自动进行。 |
| 应用编译 | 团队成员编译提交应用。 |
| 黑盒测试 | 从用户角度手动执行验收测试。 |
| 跨浏览器测试 | 在多个浏览器上运行所有的自动测试。 |
| 跨平台测试 | 在多个平台上运行所有的自动测试。 |
| 全球化测试 | 在多语言环境下运行所有的自动测试。 |
| 性能测试 | 对系统特定功能进行性能测试。 |
| 安全审查 | 微软内部安全审核标准，涵盖线程模型，识别攻击因素和使用安全分析工具覆盖代码。 |
| 压力测试 | 进行系统稳定性的极端测试；具体到驱动问题，比如内存泄漏和线程处理问题。 |
| 白盒测试 | 深入源码分析验证编码标准，结构和如何映射到整体体系结构。 |

Prism库源码包括单元测试和UI自动化测试，如下表所示。你可以使用这些作为学习的资源，或者针对Prism库本身进行这些测试。这是允许自定义，重编译，测试，和部署的修改版本Prism的质量，能达到Prism团队的水平。

|  |  |
| --- | --- |
| 测试 | 描述 |
| UI自动测试 | 限定范围的测试；从用户角度测试应用。 |
| 单元测试 | 验证类的实现 |

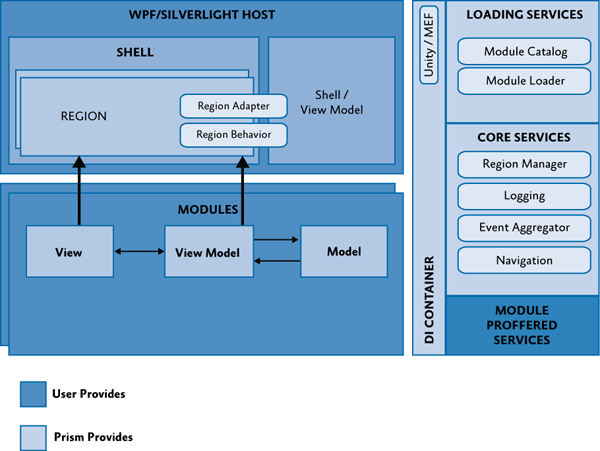
## Prism关键概念

Prism可能提供了你不熟悉的功能和模式，尤其是如果你刚开始接触设计模式和复合应用开发。本章简要概述Prism的主要概念以及一些你在阅读文档和代码时，会遇到的术语。

* **Modules**. 模块是可以独立开发，测试和(可选)部署的功能包。许多情况下，模块是由独立的团队开发和维护的。典型的Prism应用是由多模块组成的。模块可以用来表示特定的业务相关功能(比如，配置文件管理)和封装实现该功能所需的所有视图，服务和数据模型。模块还可以用于封装通用应用基础架构或服务 (比如，日志记录和异常管理服务)，可以跨应用重用。
* **Module catalog**. 在一个复合应用中，模块应该由宿主在运行时发现和加载。Prism中，模块目录是用来制定哪些模块应该被加载，什么时候加载，以何种顺序加载。模块目录用在**ModuleManager**和**ModuleLoader**组件中，如果模块是远程的话，负责下载模块，加载模块应用集到应用中，并初始化模块。Prism允许模块目录指定不通的加载方法，包括使用编程方式，在XAML中生命，或者配置文件。你可以实现一个自定义目录，如果需要的话。
* **Shell**. Shell是宿主应用的已加载模块。Shell定义了应用的整体布局和结构,但是它通常是不知道确切的模块的。它通常实现共通应用服务和基础架构，但是大多数应用的功能和内容，是在模块内实现的。Shell还提供顶级窗口或者视觉元素，用于装载模块提供的不通UI组件。
* **Views**. Views是应用中封装了特定功能或者特定区域的UI控件。Views用在 MVVM或Model-View-Presenter (MVP)模式中，用来提供UI和应用逻辑，及数据之间的分离。View被用于封装UI和定义用户交互行为，允许View独立的更新或者替换，而对底层的业务逻辑不产生影响。View通过数据绑定来关联View model和presenter类。
* **View Model及Presenter**. View Model类封装了应用的表现逻辑和状态。他们是MVVM的一部分。View Model封装了应用大部分的功能。Presenters和View Model类似，都是封装了表现逻辑和状态。他们是MVP模式的一部分。View Model及Presenter用于定义属性，名利和事件，使得可以在View中通过数据绑定调用。
* **Models**. Model封装了应用的数据和业务逻辑。是MVVM或MVP模式的一部分。模块封装了数据，验证，及业务规则，确保数据的统一性和完整性。
* **命令**. 命令用来封装应用的功能，让他们可以定义和测试独立于应用的UI。他们可以定义成命令对象或者命令方法，在ViewModel或者Presenter中。Prism提供了**DelegateCommand类和CompositeCommand类。后者用来整合命令群。**
* **Region**. Region是为了在应用UI中(Shell或者View中)显示View而定义的逻辑占位符。Region可以更新应用UI的布局，而无需影响应用逻辑。许多通用控件可以当成Region使用，在其中自动显示View，比如**ContentControl**, **ItemsControl**，**ListBox**，或**TabControl**。View可以通过编码方式或者自动显示。Prism支持在Region中导航。Region可以被另外组件定位，通过**RegionManager的RegionAdapter和RegionBehavior组件，指定View显示在特定Region。**
* **导航**. 导航是应用根据用户的交互或应用内部状态变更，响应变更结果对UI进行更新的一个过程。Prism支持两种风格的导航:基于状态的导航，View根据状态的变化而更新的简单导航场景，以及基于视图切换的导航，在应用UI中实现新旧View更换。基于视图切换的导航使用基于URI的导航机制，允许灵活的导航。
* **事件聚合**. 在以松耦合模式构建的复合应用中，组件通常需要与其它组件和服务通讯。为了支持这一特性，Prism提供了**EventAggregator**组件，实现了pub-sub事件机制。从而允许组件发布事件，另外的组件可以订阅这些事件，而无需引用组件本体。**EventAggregator通常允许定义在不同模块中，以使得组件可以相互通讯。**
* **依赖注入容器**. 整个Prism用的是依赖注入模式，使得组件间的依赖关系可以管理起来。依赖注入允许在运行时完成组件依赖，并且支持扩展和测试。Prism的默认依赖注入容器是Unity或MEF，或者通过使用**ServiceLocator调用**其他依赖注入容器。
* **服务**. 服务用来封装非UI相关的功能。比如日志，异常管理，和数据访问。服务可以由应用或者模块定义。服务通常由依赖注入容器注册，定位或者以请求方式创建，然后用于与他们相关的组件。
* **控制器**. 控制器用来创建或初始化应用UI中Region上显示的View。控制器封装了显示逻辑，决定了哪个View需要显示。控制器可以用Prism提供的基于视图切换的导航机制，该机制提供了一个可扩展的，基于URI的，协调Region中View的创建和布局的模式.应用控制模式定义了一个映射到这类功能的抽象。
* **Bootstrapper**. **Bootstrapper**是应用用来初始化Prism的各种组件和服务的。用来初始化依赖注入容器，注册任何应用级别的组件和服务。用于配置和初始化模块目录和shell的view和view model或者presenter。
* **多目标**. Prism支持针对WPF和Silverlight的应用开发。通过分离文档模式，比如MVVM或者MVP模式，你可以将UI和业务逻辑中分离出来。View model, presenter, 和 model类可以再WPF和Silverlight版本中重用。然后为WPF和Silverlight封装特定的界面。

Prism旨在让你可以单独或者一起使用上述功能或设计模式，这取决于需求及应用设计。你可以单独使用MVVM模式，模块化，Regions，命令，或者事件任意一项或组合。当然，如果你希望得到关注分离及松耦合提供的全部好处，需要更多的使用Prism提供的功能和设计模式来协调。下面的插图显示了一个典型的Prism应用体系结构，并显示如何将Prism各项功能以多模块的方式结合在一起。

**使用Prism的典型复合应用体系结构**



大多数Prism应用包括一个Shell应用，定义了Region，用来显示顶级View和共享服务，以便加载的模块可以访问。Shell中定义了模块目录，指定哪些模块在启动时加载，哪些根据需求加载。依赖注入容器也在这时定义，允许组件组件在运行时完成依赖。通过应用启动时的**Bootstrapper，**共享服务和组件被注册到容器中。

独立模块封装应用功能的一部分，使用分离模式，比如MVVM，定义View，View Model，Model和服务组件。当模块被加载，定义在模块中的View被显示到Shell定义的Region中。完成初始化之后，则会通过基于状态或者基于视图切换的导航，更新可视区或创建新视图，将用户导航到应用指定的Region。

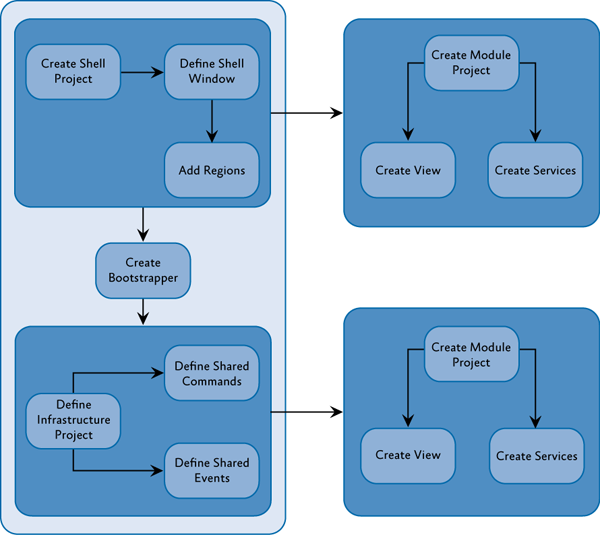
## 使用Prism

到此，你已经了解了Prism提供的主要功能和设计模式了，那么我们来看看如何在开发一个新的应用时，是怎样使用Prism的。本章概述了创建一个基本的Prism应用所需的几个基本步骤。你可以根据需要，扩展此基础应用，使之使用Prism提供的功能和设计模式。

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| Prism库不但可以在创建新的WPF或Silverlight应用时候使用(或者以两者为目标)，你也可以在现有的应用上引入Prism库，使用Prism的部分功能或设计模式。 |

Prism应用通常可以看成一个shell工程加上多个模块工程。下图显示开发复合应用时候，使用Prism所需要的流程。

**创建一个复合应用的流程**



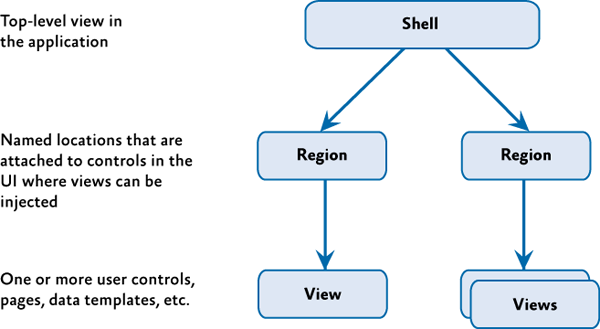
一个典型的Prism应用，使用了绝大部分Prism提供的功能和设计模式，如前面所述。可以充分体现关注分离和松耦合设计模式的好处。然而，对于此例，创建一个最基本的Prism应用，是在一个单一的模块中创建一个单一的视图。

|  |
| --- |
| **Ff921153.note(en-us,PandP.40).gif注意:** |
| **Prism库引用**  Prism应用需要引用Prism库。Prism提供了签名二进制包，以及一个允许你将二进制包注册到Visual Studio的脚本文件，注册后，可以在Visual Studio的**添加引用**对话框中选择Prism引用。如果你决定不注册Prism库，你需要手动将Prism库添加到你的工程。你也可以直接将Prism工程添加到解决方案，然后使用工程引用。然后在调试中，可以使用“转到定义”进入Prism内部查看源码，就像在你自己的工程中一样便利。 |

### 定义Shell

应用的Shell提供了应用的基本布局。布局使用模块可以放置View的Region来定义。View，像Shell一样，可以用Region定义可发现区域，可以将内容添加到其中，如下图所示。Shell通常设置整个应用的外观，以及整个应用的全局样式。

**Shell, views, 和 regions**



### 创建Bootstrapper

Bootstrapper是Prism库和Unity及MEF容器之间的黏合剂。每个应用创建一个别的Bootstrapper，从**UnityBootstrapper**或**MefBootstrapper继承而来。如下插图所示，你要决定**,你希望用来填充模块目录的方法。至少，每个应用要提供一个模块目录及一个Shell。

默认情况下，Bootstrapper日志使用.NET Framework **Trace类。**大多数应用希望支持他们自己的日志服务，比如Enterprise Library logging。应用可以在Bootstrapper中支持自定义日志服务。

默认情况下，**UnityBootstrapper**和**MefBootstrapper启动Prism服务。可以通过设置应用的BootStrapper关闭或者替换这些服务**。

**链接到Prism库示意图**



### 创建Module

Module封装了应用特定功能，包括View和服务。通常，模块是作为独立的程序集和由独立团队开发。模块是由一个类实现**IModule接口实现的。这些模块在初始化的时候，将他们的View和服务注册，并且将一个或多个View添加到Shell中。**根据模块发现方法，你可能需要将特定的属性应用到模块的类，或者定义模块之间的依赖关系。

### 添加View模块到Shell

Module利用Shell的Region放置内容。在初始化时，模块使用**RegionManager定位Region在Shell中的为止，并添加一个或多个**View到该Region或者注册一个或多个View类型，用来在该Region中创建View。**RegionManager是一个从**bootstrapper初始化的核心服务，**负责跟踪整个应用的Region。**

本指南中其余章节将提供关于Prism关键概念的更多详细信息。

# 更多

Prism假设你有WPF或Silverlight的实际经验。如果你需要有关WPF和Silverlight的一般信息，请查阅以下资源:

* [Windows Presentation Foundation](http://msdn2.microsoft.com/en-us/library/ms754130.aspx)在MSDN.
* MacDonald, Matthew. Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4, Apress, 2010.
* Nathan, Adam. WPF 4 Unleashed. Sams Publishing, 2010.
* Bugnion, Laurent. Silverlight 4 Unleashed, Sams Publishing, 2010.
* Brown, Pete. Silverlight 4 in Action, Manning Publications, 2010.

如果你需要关于Silverlight的一般信息，请查阅以下:

* [Microsoft Silverlight主页](http://www.microsoft.com/silverlight/).
* [Microsoft Silverlight资源](http://www.microsoft.com/silverlight/resources/default.aspx).
* [Microsoft Silverlight社区](http://silverlight.net/default.aspx).
* [Silverlight文档](http://msdn.microsoft.com/en-us/library/cc838158(vs.95).aspx)在MSDN.
* [Silverlight书籍](http://www.silverlightbooks.net/).
* [Project Rosetta](http://visitmix.com/labs/rosetta/).

## 社区

Prism的社区是<http://www.codeplex.com/Prism>. 在该社区，可以发布问题，提供反馈，或者和其他用户共享灵感。社区成员还可以帮助微软策划和测试未来产品和下载内容，比如扩展和培训材料。

# 2: 初始化Prism

|  |
| --- |
| **索引:** |
| [**什么是Bootstrapper?**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec1) | [**依赖注入**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec2) | [**创建Shell**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec3) | [**关键决策**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec4) | [**核心方案**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec5) | [**为你的应用创建Bootstrapper**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec6) – [*实现CreateShell方法*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec7), [*实现InitializeShell方法*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec8) | [**创建并配置模块目录**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec9) | [**创建并配置容器**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec10) – [*核心服务*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec11), [*示例应用服务*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec12), [*创建并配置容器:UnityBootstrapper*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec13), [*创建并配置容器:MefBootstrapper*](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec14) | [**更多**](http://msdn.microsoft.com/en-us/library/gg430868(v=pandp.40)#sec15) |

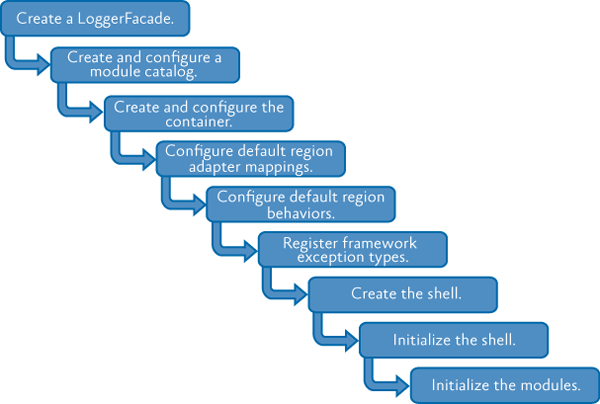
This chapter addresses what needs to happen to get a Prism application up and running. A Prism application requires registration and configuration during the application startup process—this is known as bootstrapping the application.

# 什么是Bootstrapper?

Bootstrapper是一个类，负责构建使用Prism库的应用的初始化。通过使用Bootstrapper，你有更多的控制权，决定Prism库如何链接到应用。

Prism库提供了一个默认的抽象基类**Bootstrapper**，可以定制使用各种容器。Bootstrapper类中的大多数方法都是虚方法(virtual method)。你可以根据需要在自定义的Bootstrapper实现中重写这些方法。

**Bootstrapping的基础流程**



Prism库提供额外的基类，扩展自**Bootstrapper**，拥有默认的实现，适合大多数应用。唯一的需要在应用的实现的Bootstrapper中实现的，是创建和初始化Shell。

## 依赖注入

由Prism生成的应用，依赖于容器所生成的依赖注入。Prism库提供的应用集可以和Unity或MEF，并且允许使用其他依赖注入容器。启动的过程，其实就是配置容器和注册的过程。

Prism库包括**UnityBootstrapper**和**MefBootstrapper**类，实现了使用Unity或者MEF作为应用的依赖注入容器所需要的大多数功能。除了上图所示的必要阶段，每个Bootstrapper对为它的容器添加了一些特殊的步骤。

## 创建Shell

传统的WPF应用中，在App.xaml文件中指定启动主窗口的URI。在Silverlight应用中，在App.xaml的后台代码中，设置**RootVisual**属性。

在使用Prism的应用中，这是Bootstrapper的责任，决定创建Shell或者MainWindow。这是因为Shell的依赖服务，比如像Region Manager，需要在Shell显示之前注册。

# 关键决策

当你决定在应用中使用Prism后，还要作出几个其他的决定：

* 需要确定是否使用MEF，Unity，或其他的依赖注入容器。这将决定是否需要实现一个自定义的Bootstrapper。
* 需要考虑应用的特定服务，这需要将它们注册到容器中。
* 决定是否使用内建日志服务，或者重新指定一个日志服务。
* 决定让应用如何发现模块：通过显式代码加载，代码属性，或者通过目录扫描，配置文件，或XAML。

本章的其余部分提供了更多细节。

# 核心场景

创建一个启动序列是创建Prism应用中非常重要的一部分。本章介绍如何创建一个Bootstrapper并自定义创建Shell，配置依赖注入容器，注册应用级别服务，和如何加载和初始化模块。

## 为你的应用创建Bootstrapper

如果你选择使用Unity或MEF作为你的依赖注入容器，为应用创建一个简单的Bootstrapper非常简单。只需要创建一个新类，从**MefBootstrapper**或**UnityBootstrapper**扩展。然后实现**CreateShell**方法。以及，如果需要，重写**InitializeShell**来实现Shell的特定初始化。

### 实现CreateShell方法

**CreateShell**方法允许开发者为Prism应用指定顶级窗口。Shell通常使用**MainWindow**或**MainPage**。实现该方法，返回应用的Shell类的一个实例。在Prism应用中，可以创建Shell对象，或者在容器中生成，根据应用需求而定。

下面的示例代码演示了如何使用**ServiceLocator**创建一个Shell对象。

**C#**

protected override DependencyObject CreateShell()

{

return ServiceLocator.Current.GetInstance<Shell>();

}

|  |
| --- |
| **Gg430868.note(en-us,PandP.40).gif注意:** |
| 你会经常看到用**ServiceLocator**来实例化Shell，而不是使用特定的依赖注入容器。**ServiceLocator**是通过调用容器来实现的，因此它是容器和代码无关的一个不错的选择。当然也可以直接引用容器，而不是使用**ServiceLocator**。 |

### 实现InitializeShell方法

当创建了Shell之后，需要运行初始化，确保Shell已经可以运行。根据应用是WPF或Silverlight工程， **InitializeShell**的实现是不同的。Silverlight应用中，需要将Shell设置为应用的Root节点，如下：

**C#**

protected override void InitializeShell()

{

Application.Current.RootVisual = Shell;

}

WPF应用，需要创建的Shell应用对象和将其设置为应用的MainWindow，如下：(来自模块化快速入门：WPF版).

**C#**

protected override void InitializeShell()

{

Application.Current.MainWindow = Shell;

Application.Current.MainWindow.Show();

}

**InitializeShell**的基本实现什么也不做，不会调用积累的实例化，因此很安全。

## 创建并管理模块目录

如果开发的是模块化应用，需要创建和管理模块目录。Prism使用**IModuleCatalog**实例跟踪哪些模块是提供给应用使用，哪些模块需要下载，哪些模块需要驻留。

**Bootstrapper**提供受保护的**ModuleCatalog**属性来引用模块目录，并且已有基础实现，通过虚方法**CreateModuleCatalog**，该基础实现返回**ModuleCatalog**实例；但是这个方法可以重写，实现返回新的**IModuleCatalog**实例，如下：(来自**Bootstrapper快速入门** 在使用MEF的Silverlight版模块化快速入门一章)

**C#**

protected override IModuleCatalog CreateModuleCatalog()

{

// When using MEF, the existing Prism ModuleCatalog is still

// the place to configure modules via configuration files.

return ModuleCatalog.CreateFromXaml(new Uri(

"/ModularityWithMef.Silverlight;component/ModulesCatalog.xaml",

UriKind.Relative));

}

在**UnityBootstrapper**和**MefBootstrapper**类中， **Run**方法调用**CreateModuleCatalog**方法，然后使用返回值设置类的**ModuleCatalog**属性。如果你重写该方法，不需要调用基类的实现，因为你需要替换所提供的功能。更多详细信息关于模块，请查阅Chapter 4, "[模块化应用开发](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40))."

## 创建并配置容器

容器在Prism应用中扮演一个关键角色。Prism库和它构建的应用，依赖于容器注入相关性和服务。在容器配置阶段，几个核心服务会被注册。除了这些核心服务，你可以根须需要提供额外的功能。

### 核心服务

下表列出了Prism库中与应用无关的服务。

|  |  |
| --- | --- |
| 服务接口 | 描述 |
| **IModuleManager** | 定义模块管理服务的接口，用于检索并初始化应用程序的模块。 |
| **IModuleCatalog** | 包含了有关在应用中的模块的元数据。Prism库提供了几种不同的目录。 |
| **IModuleInitializer** | 初始化模块。 |
| **IRegionManager** | 注册并搜索Region，是布局的容器。 |
| **IEventAggregator** | 发布和订阅之间的松耦合事件聚合。 |
| **ILoggerFacade** | 日志机制的包装器。以便可以选择自己的日志机制。Stock Trader参考实现(Stock Trader RI)中使用的是企业库日志应用模块，通过**EnterpriseLibraryLoggerAdapter类**，该例子作为如何使用自己的日志机制的示例。在Bootstrapper的Run方法中，日志服务被注册到容器中，通过使用**CreateLogger方法的返回值。在容器中注册另外的日志服务将不起作用，除非重写**Bootstrapper**中的CreateLogger**方法。 |
| **IServiceLocator** | 允许Prism库访问容器，如果希望自定义或者扩展Prism库的话，这个非常有用。 |

### 示例应用服务

下表列出了在Stock Trader RI中用到的特定的应用服务。可以用作示例，以了解你应用可以提供的服务的类型。

|  |  |
| --- | --- |
| Stock Trader RI中的服务 | 描述 |
| **IMarketFeedService** | 提供实时 (虚拟)市场数据。**PositionSummaryPresentationModel**更新屏幕位置，基于它接收到来自此服务的通知。 |
| **IMarketHistoryService** | 提供市场历史数据，用于现实所选基金的趋势线。 |
| **IAccountPositionService** | 天哦共基金投资组合列表。 |
| **IOrdersService** | 提供提交的买/卖订单。 |
| **INewsFeedService** | 提供所选基金新项目列表 |
| **IWatchListService** | 当新项目添加到监事列表时处理新项目 |

有两个**Bootstrapper的派生类在Prism中，UnityBootstrapper**和**MefBootstrapper**。创建和配置不同容器虽然概念类似，但是实现去完全不同。

### 创建并配置容器:UnityBootstrapper

**UnityBootstrapper**类的**CreateContainer**方法简单的创建和返回一个新的**UnityContainer**实例。在大多数情况下，无需改变此功能；然而，该方法是虚方法，因此允许重写。

容器创建后，可能需要根据应用进行配置。**UnityBootstrapper** 的**ConfigureContainer**实现注册了Prism的大多数核心服务，如下：

|  |
| --- |
| **Gg430868.note(en-us,PandP.40).gif注意:** |
| 此处演示的是当一个模块在**Initialize** 方法中注册模块级别的服务。 |

**C# UnityBootstrapper.cs**

protected virtual void ConfigureContainer()

{

...

if (useDefaultConfiguration)

{

RegisterTypeIfMissing(typeof(IServiceLocator), typeof(UnityServiceLocatorAdapter), true);

RegisterTypeIfMissing(typeof(IModuleInitializer), typeof(ModuleInitializer), true);

RegisterTypeIfMissing(typeof(IModuleManager), typeof(ModuleManager), true);

RegisterTypeIfMissing(typeof(RegionAdapterMappings), typeof(RegionAdapterMappings), true);

RegisterTypeIfMissing(typeof(IRegionManager), typeof(RegionManager), true);

RegisterTypeIfMissing(typeof(IEventAggregator), typeof(EventAggregator), true);

RegisterTypeIfMissing(typeof(IRegionViewRegistry), typeof(RegionViewRegistry), true);

RegisterTypeIfMissing(typeof(IRegionBehaviorFactory), typeof(RegionBehaviorFactory), true);

}

}

Bootstrapper的**RegisterTypeIfMissing**方法确定是否一个服务已经被注册——它不会被注册两次。Prism允许你通过配置重写默认注册方式。默认情况下，可以关闭注册任何服务；要执行此操作，使用重载的**Bootstrapper.Run**方法，传入**false**。你也可以重写**ConfigureContainer**方法和关闭你不想使用的服务，比如事件聚合。

|  |
| --- |
| **Gg430868.note(en-us,PandP.40).gif注意:** |
| 如果你关闭默认的注册机制，你需要手动注册所需的服务。 |

要扩展**ConfigureContainer**默认行为，只需重载应用的Bootstrapper并选择是否调用基类实现，如来自使用Unity的WPF版模块化快速入门中的**QuickStartBootstrapper中**。该实现调用基类的实现，注册了**IModuleTracker** 的具体实现**ModuleTracker**类型，注册了**callbackLogger**实例作为Unity 的**CallbackLogger**。

**C#**

protected override void ConfigureContainer()

{

base.ConfigureContainer();

this.RegisterTypeIfMissing(typeof(IModuleTracker), typeof(ModuleTracker), true);

this.Container.RegisterInstance<CallbackLogger>(this.callbackLogger);

}

### 创建并配置容器:MefBootstrapper

**MefBootstrapper**类的**CreateContainer**方法做了几件事情。首先，他会创建一个**AssemblyCatalog**和一个**CatalogExportProvider**。**CatalogExportProvider**允许**MefExtensions**程序集提供对Prism类型的默认浏览，同样允许你重写注册的默认类型。然后**CreateContainer**创建和返回一个新的**CompositionContainer**实例，通过使用**CatalogExportProvider**。大多数情况下，你不需要改变此功能；然而，该方法是虚方法，因此可以重写。

|  |
| --- |
| **Gg430868.note(en-us,PandP.40).gif注意:** |
| 在 Silverlight中，因为安全限制，不允许通过类型检索程序集，但是Prism提供了另一种方法，调用**Assembly.GetCallingAssembly**方法。 |

创建容器后，需根据应用进行配置。**MefBootstrapper** 中的**ConfigureContainer** 实现了Prism的大多数核心服务，如下所示。如果重载该方法，请考虑是否需要调用基类的实现，以注册Prism的核心服务，或者你可以使用自定义的实现。

**C#**

protected virtual void ConfigureContainer()

{

this.RegisterBootstrapperProvidedTypes();

}

protected virtual void RegisterBootstrapperProvidedTypes()

{

this.Container.ComposeExportedValue<ILoggerFacade>(this.Logger);

this.Container.ComposeExportedValue<IModuleCatalog>(this.ModuleCatalog);

this.Container.ComposeExportedValue<IServiceLocator>(new MefServiceLocatorAdapter(this.Container));

this.Container.ComposeExportedValue<AggregateCatalog>(this.AggregateCatalog);

}

|  |
| --- |
| **Gg430868.note(en-us,PandP.40).gif注意:** |
| 在**MefBootstrapper**中，由于Prism的核心服务都作为单例添加到容器中，因此他们可以在整个应用中定位使用。 |

除了**CreateContainer**和**ConfigureContainer**方法， **MefBootstrapper**额外提供了两个方法用于创建和配置MEF中使用的**AggregateCatalog**。**CreateAggregateCatalog**方法简单的创建并返回**AggregateCatalog**对象。和**MefBootstrapper**其他方法一样，**CreateAggregateCatalog**是虚方法，如果需要，可以进行重载。

**ConfigureAggregateCatalog**方法允许强制追加注册类型到**AggregateCatalog**中。比如在来自使用MEF的Silverlight版模块化快速入门中的**QuickStartBootstrapper**，显式添加ModuleA和ModuleC到**AggregateCatalog**，如下所示。

**C#**

protected override void ConfigureAggregateCatalog()

{

base.ConfigureAggregateCatalog();

// Add this assembly to export ModuleTracker

this.AggregateCatalog.Catalogs.Add(

new AssemblyCatalog(typeof(QuickStartBootstrapper).Assembly));

// Module A is referenced in in the project and directly in code.

this.AggregateCatalog.Catalogs.Add(

new AssemblyCatalog(typeof(ModuleA.ModuleA).Assembly));

// Module C is referenced in in the project and directly in code.

this.AggregateCatalog.Catalogs.Add(

new AssemblyCatalog(typeof(ModuleC.ModuleC).Assembly));

}

# 更多

更多详细信息关于MEF，**AggregateCatalog**，和**AssemblyCatalog**，请查看MSDN 上的"Managed Extensibility Framework Overview"：  
<http://msdn.microsoft.com/en-us/library/dd460648.aspx>.

# 3: 组件间依赖管理

|  |
| --- |
| **索引:** |
| [**关键决策:选择一个依赖注入容器**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec1) | [**使用容器**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec2)**的注意事项** | [**核心场景**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec3) |[**注册**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec4) – [*为Unity容器注册类型*](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec5), [*为MEF*](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec6)*容器注册类型* | [**解析**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec7) – [*为Unity*](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec8)*解析实例*,[*为MEF*](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec9)*解析实例* | [**在Prism中使用依赖注入容器和服务**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec10) | [**IServiceLocator**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec11) | [**使用IServiceLocator**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec12)**的注意事项** | [**更多**](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40)#sec13) |

基于Prism的应用通常是由许多松耦合的类型和服务组成的复合应用。他们需要互动，根据提供的内容和接收到的用户操作的通知。因为他们是松耦合的，他们需要相互之间通讯的一种业务功能。

为了将各种功能合成到一起，基于Prism的应用依靠的是依赖注入容器。依赖注入容器降低依赖对象间的耦合，通过在容器的配置，实例化及管理实例的生命周期。当对象被创建时， 容器为对象注入它所需的所有依赖。如果这些依赖尚未创建，则容器先为其创建并解析。有些时候，容器本身被作为一个依赖来解析。比如，当使用Unity作为容器，模块被注入容器，使它们可以注册自己的View和服务到容器中。

使用容器有以下几个优点：

* 容器移除了组件，定位依赖或者管理生命周期的需要。
* 容器允许在不影响组件的情况下交换实现依赖。
* 容器提高可测试性，允许提供模拟的依赖。
* 容器提高可维护性，允许新组件很简单的添加到系统中。

在使用基于Prism的应用中，使用容器有独特的优点:

* 容器在模块加载的时候将依赖关系注入模块。
* 容器用于注册和解析View Model及View。
* 容器可以创建View Model并注入View。
* 容器注入组合服务，比如Region管理和事件聚合。
* 容器用于注册特定模块的服务，它们具有特定模块的功能服务。

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| Prism指南中的某些示例使用Unity作为容器。另一些示例，比如模块化快速入门，使用MEF。由于Prism库并没有特定的容器，因此你可以使用它的服务及模式，与其他容器一起工作。比如Castle Windsor, StructureMap,和Spring.NET。 |

# 关键决策:选择一个依赖注入容器

Prism库提供了两个可选的依赖注入容器: Unity或MEF。Prism是可扩展的，因此只需要稍作改动，就可以使用其他容器。Unity和MEF都提供了依赖注入的基本功能，尽管他们工作方式非常不同。两个容器提供的功能包括以下:

* 都可以注册类型到。
* 都可以注册示例到容器。
* 都可以强制为已注册类型创建示例。
* 都将已注册类型的实例注入到结构体。
* 都将已注册类型的实例注入到属性。
* 都有标记类型和依赖为需管理类型的声明属性。
* 都解析在对象图中的依赖关系。

Unity提供但MEF不提供的功能：

* 可解析未注册具体类型。
* 可解析范型。
* 使用拦截来俘获对对象的调用，并对目标对象添加额外功能。

MEF提供但Unity不提供的功能:

* 可以发现在文件夹中的应用集。
* 可以使用XAP文件格式下载和发现应用集。
* 可以重组属性和集合成为用于被发现的新类。
* 自动导出派生类型。
* 已包含到.NET框架。

容器在功能及工作方式上有不同，但是Prism库可以和他们协同工作，并提供类似的功能。当考虑应使用哪个容器的时候，请牢记并确定，适合你的方案是最好的。

## 使用容器的注意事项

在使用容器之前，应该仔细考虑以下的事项：

* 考虑是否需要使用容器注册和解析组:
  + 考虑使用容器从注册到解析出实例的性能开销是否适合你的应用场景。比如，你需要调用一个渲染方法，在区域中创建10,000个多边形，并渲染表面。通过容器解析所有的多边形实例，在性能上将是很大的开销，因为容器会通过反射创建所有的实例。
  + 如果有过多或过深的依赖，性能开销将会显著提升。
  + 如果组件没有任何依赖项，或不是其他类型的依赖项，把它放到容器中可能毫无意义。
  + 如果组件有一部分依赖项是该项的组成部分，并且永远也不会改变，把它放到容器中可能毫无意义。
* 考虑是否需要将一个组件注册为单例或实例:
  + 如果组件是全局服务，作为资源管理器，比如日志服务，你也许希望把它注册为单例。
  + 如果组件为多个消费者提供共享状态，你也许希望把它注册为单例。
  + 如果对象在每次注入它的依赖项的时候，需要一个新的实例，将它注册为非单例。比如，每个View可能需要一个新的View Model。
* 考虑是否需要通过配置文件或编码方式配置容器:
  + 如果你希望集中管理所有不同的服务，在配置文件中配置容器。
  + 如果你希望有条件的注册特定的服务，通过编码方式配置容器。
  + 如果你有模块级别的服务，考虑通过编码方式配置容器，以便只有在模块被加载的时候才注册服务。

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 某些容器，比如MEF，不能通过配置文件来配置，只能通过使用编码方式配置。 |

# 核心场景

容器主要用于两个目的，即注册和解析。

## 注册

可以将依赖关系注入一个对象之前，需要将依赖项的类型注册到容器。注册一个类型通常涉及向容器传入一个接口及一个实现该接口的具体类型。有两种手段实现：通过编码方式或配置文件。具体方法根据容器而异。

通常，以编码方式，注册类型和对象到容器中有两种方法:

* 你可以注册一个类型或将其映射到容器。在适当时机，容器会创建你所指定的类型的实例。
* 你可以注册一个既存对象实例，作为单例到容器中，容器会返回对现有对象的引用。

### 为Unity容器注册类型

在初始化过程中，一个类型可以注册其他类型，比如View和服务。注册可以通过容器提供他们的依赖，并允许其他类进行访问。为做到这点，该类将需要将容器注入模块构造。下面的代码演示**OrderModule**类型在命令快速入门中如何注册类型。

public class OrderModule : IModule

{

public void Initialize()

{

this.container.RegisterType<IOrdersRepository, OrdersRepository>(new ContainerControlledLifetimeManager());

...

}

...

}

根据你使用的容器，注册可以通过配置文件来执行。此示例请查阅Chapter 4, "[模块化应用开发](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40))."的"通过配置文件注册模块"

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 相对于使用配置文件的优点，使用代码配置容器，只有在模块加载时进行注册。 |

### 为MEF 容器注册类型

在MEF中，注册类型到容器使用的是基于属性的机制。因此，添加该容器的注册类型很简单:它只需要为类型添加 **[Export]属性**。如下所示。

[Export(typeof(ILoggerFacade))]

public class CallbackLogger: ILoggerFacade

{

}

使用MEF的另一选择是，创建一个类的实例，并在容器中注册该特定实例。在使用MEF的Silverlight版模块化快速入门中的**QuickStartBootstrapper**的**ConfigureContainer**方法中，演示了这种方法。如下所示。

protected override void ConfigureContainer()

{

base.ConfigureContainer();

// Because we created the CallbackLogger and it needs to

// be used immediately, we compose it to satisfy any imports it has.

this.Container.ComposeExportedValue<CallbackLogger>(this.callbackLogger);

}

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 当选择MEF作为容器，建议使用属性来注册类型。 |

## 解析

类型注册之后，可以作为依赖关系解析或者注入。当一个类型被解析后，容器需要创建一个新的实例，并将依赖关系注入该实例。

通常，当类型被解析后，下列三件事情其中之一会发生:

* 如果该类型未被注册，容器会抛出异常。

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 某些容器，包括Unity，允许解析尚未注册的具体类型。 |

* 如果类型被注册为单例，容器返回该单例的实例。如果这是该类型第一次调用，容器创建，并并保持该实例。以便之后的调用。
* 如果类型未被注册为单例，容器返回一个新的实例。

|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 默认情况下，MEF中的类型都作为单例注册，容器管理对象的引用。再Unity中，默认是创建对象的新实例，容器并不管理对象的引用。 |

### 为Unity解析实例

来自命令快速入门的代码，演示了从容器中解析**OrdersEditorView**和**OrdersToolBar**，再将其关联到相应的区域中。

public class OrderModule : IModule

{

public void Initialize()

{

this.container.RegisterType<IOrdersRepository, OrdersRepository>(new ContainerControlledLifetimeManager());

// Show the Orders Editor view in the shell's main region.

this.regionManager.RegisterViewWithRegion("MainRegion",

() => this.container.Resolve<OrdersEditorView>());

// Show the Orders Toolbar view in the shell's toolbar region.

this.regionManager.RegisterViewWithRegion("GlobalCommandsRegion",

() => this.container.Resolve<OrdersToolBar>());

}

...

}

**OrdersEditorPresentationModel**构造函数中包括下列依赖项(订单存储库和订单命令代理)，在解析时候将其注入。

public OrdersEditorPresentationModel( IOrdersRepository ordersRepository, OrdersCommandProxy commandProxy )

{

this.ordersRepository = ordersRepository;

this.commandProxy = commandProxy;

// Create dummy order data.

this.PopulateOrders();

// Initialize a CollectionView for the underlying Orders collection.

#if SILVERLIGHT

this.Orders = new PagedCollectionView( \_orders );

#else

this.Orders = new ListCollectionView( \_orders );

#endif

// Track the current selection.

this.Orders.CurrentChanged += SelectedOrderChanged;

this.Orders.MoveCurrentTo(null);

}

除上述代码演示的，通过构造函数注入外， Unity同样允许属性注入。任何使用**[Dependency]** 标记的属性，当对象被解析的时候，都会自动解析和注入。

### 为MEF解析实例

下面来自使用MEF的Silverlight版模块化快速入门中的Bootstrapper 代码，演示了如何获取Shell的实例的。无需具体类型，代码可以通过一个接口获取实例。

protected override DependencyObject CreateShell()

{

return this.Container.GetExportedValue<Shell>();

}

在MEF解析的类中，你可以使用构造体注入，下面来自使用MEF的Silverlight版模块化快速入门中的ModuleA代码演示了如何注入ILoggerFacade和IModuleTracker。

[ImportingConstructor]

public ModuleA(ILoggerFacade logger, IModuleTracker moduleTracker)

{

if (logger == null)

{

throw new ArgumentNullException("logger");

}

if (moduleTracker == null)

{

throw new ArgumentNullException("moduleTracker");

}

this.logger = logger;

this.moduleTracker = moduleTracker;

this.moduleTracker.RecordModuleConstructed(WellKnownModuleNames.ModuleA);

}

另一个选择是使用属性注入，下面来自使用MEF的Silverlight版模块化快速入门中的ModuleTracker类的代码演示了如何注入ILoggerFacade。

[Export(typeof(IModuleTracker))]

public class ModuleTracker : IModuleTracker

{

// Due to Silverlight/MEF restrictions, this must be public.

[Import] public ILoggerFacade Logger;

}

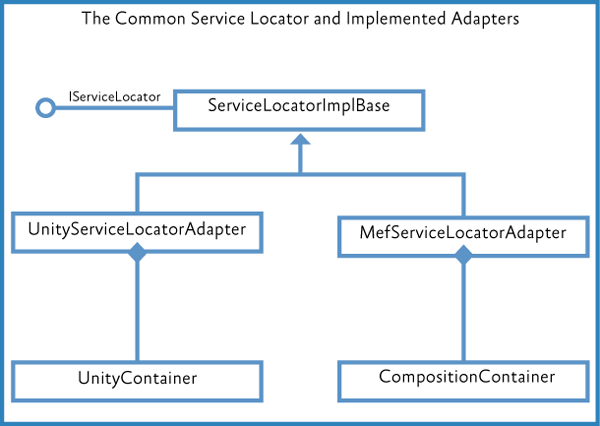
|  |
| --- |
| **Ff921140.note(en-us,PandP.40).gif注意:** |
| 在Silverlight中， 用于导入的属性和字段必须是公共的。 |

# 在Prism中使用依赖注入容器和服务

依赖注入容器，通常指的是"容器"，用于满足组件间的依赖关系；满足这些依赖关系通常涉及注册和解析。Prism库提供Unity和MEF容器，但是它并不依赖特定容器。因为库通过**IServiceLocator接口访问容器，因此容器可以置换。若要置换容器，你的容器必须实现IServiceLocator**接口。 通常，如果你使用自定义容器，你同时需要提供自定义的Bootstrapper。**IServiceLocator**接口被定义在通用服务定位器库中。这是一个开放源码，奴隶提供一个抽象的IoC(反转控制)容器，比如依赖注入容器，和服务定位器。使用该库的目的是利用IoC和服务定位，不需要依赖具体的实现。

Prism库提供了**UnityServiceLocatorAdapter**和**MefServiceLocatorAdapter。这两个适配器都实现了ISeviceLocator**接口，统括对**ServiceLocatorImplBase**类型扩展。下图显示了类的阶层。

**Prism中的通用服务定位器的实现**



虽然Prism库不引用或依赖一个特定的容器，但是应用是依赖于特定容器的。这意味着应用需要合理的引用容器，但是Prism库不直接引用容器。比如，再Stock Trader RI和几个快速入门中，Prism库使用了Unity作为容器，而另一些使用MEF。

# IServiceLocator

下列代码演示了**IServiceLocator**接口。

public interface IServiceLocator : IServiceProvider

{

object GetInstance(Type serviceType);

object GetInstance(Type serviceType, string key);

IEnumerable<object> GetAllInstances(Type serviceType);

TService GetInstance<TService>();

TService GetInstance<TService>(string key);

IEnumerable<TService> GetAllInstances<TService>();

}

服务定位器在Prism库中被扩展，通过如下扩展方法。你可以看到， **IServiceLocator只被用于解析，这意味着它只用于获取实例；它并不用于注册。**

public static class ServiceLocatorExtensions

{

public static object TryResolve(this IServiceLocator locator, Type type)

{

try

{

return locator.GetInstance(type);

}

catch (ActivationException)

{

return null;

}

}

public static T TryResolve<T>(this IServiceLocator locator) where T: class

{

return locator.TryResolve(typeof(T)) as T;

}

}

**TryResolve**扩展方法——Unity容器不支持——如果一个类型已被注册，将返回该类型的实例；否则将返回**null**。

**ModuleInitializer**使用**IServiceLocator**解析模块，在模块加载过程中，如下代码所示：

IModule moduleInstance = null;

try

{

moduleInstance = this.CreateModule(moduleInfo);

moduleInstance.Initialize();

}

...

protected virtual IModule CreateModule(string typeName)

{

Type moduleType = Type.GetType(typeName);

if (moduleType == null)

{

throw new ModuleInitializeException(string.Format(CultureInfo.CurrentCulture, Properties.Resources.FailedToGetType, typeName));

}

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

}

# 使用IServiceLocator的注意事项

**IServiceLocator并**不是通用容器。容器有不同的语义，会根据应用不同有不同的选择。这点需要铭记于心。在Stock Trader RI中，直接使用依赖注入容器而不是**IServiceLocator**。这是应用开发的推荐方法。

在下列情况中，可能适合使用**IServiceLocator**:

* 你是独立软件供应商 (ISV)，设计一个第三方服务，需要支持多容器。
* 你为一个正在使用多容器的组织提供服务。

# 更多

有关容器相关信息，请查阅以下:

* MSDN上的"Unity Application Block":   
  <http://www.msdn.com/unity>.
* CodePlex上的Unity社区:  
  <http://www.codeplex.com/unity>.
* MSDN上的"Managed Extensibility Framework Overview":  
  <http://msdn.microsoft.com/en-us/library/dd460648.aspx>.
* CodePlex上的MEF社区:  
  <http://mef.codeplex.com/>.
* Martin Fowler的网站上的"反转控制容器和依赖注入模式":  
  <http://www.martinfowler.com/articles/injection.html>.
* MSDN Magazine的"设计模式：依赖注入":  
  <http://msdn.microsoft.com/en-us/magazine/cc163739.aspx>.
* MSDN Magazine的"Loosen Up: Tame Your Software Dependencies for More Flexible Apps":  
  <http://msdn.microsoft.com/en-us/magazine/cc337885.aspx>.
* Castle工程:  
  <http://www.castleproject.org/container/index.html>.
* StructureMap:  
  <http://structuremap.sourceforge.net/Default.htm>.
* Spring.NET:  
  <http://www.springframework.net/>.

# 4: 模块化应用开发

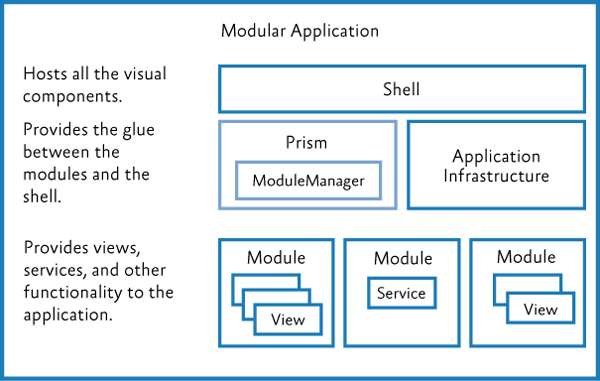
|  |
| --- |
| **索引:** |
| [**构建模块化应用的好处**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec1) | [**Prism对模块化应用开发的支持**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec2) | [**核心概念**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec3) | [**IModule:模块化应用的构造块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec4) | [**模块生命周期**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec5) | [**模块目录**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec6) | [**控制加载模块时机**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec7) | [**在应用中集成模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec8) | [**模块之间的通信**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec9) | [**依赖注入和模块化的应用**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec10) | [**关键决策**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec11) | [**把你的应用划分成模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec12) – [*决定模块的粒度*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec13), [*XAP和模块封装*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec14) | [**松耦合中**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec15)**使用依赖注入** | [**核心场景**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec16) | [**定义模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec17) | [**注册和发现模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec18) – [*通过代码注册模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec19), [*通过XAML注册模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec20), [*通过配置文件注册模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec21), [*发现文件夹下的模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec22) | [**加载模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec23) | [**初始化模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec24) | [**指定模块依赖关系**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec25) – [*通过代码指定依赖关系*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec26) | [**通过XAML指定依赖关系**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec27) – [*通过配置文件指定依赖关系*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec28) | [**按需加载模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec29) – [*通过代码指定按需加载*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec30), [*通过XAML指定按需加载*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec31), [*通过配置文件指定按需加载*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec32), [*加载指定模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec33) | [**在后台下载远程模块**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec34) – [*为远程下载准备一个模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec35), [*跟踪下载进度*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec36) | [**检测模块是否加载完毕**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec37) | [**MEF**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec38)**中的模块** – [*使用MEF*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec39)*通过代码注册模块*, [*使用MEF*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec40)*发现文件夹下的模块*, [*使用MEF*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec41)*通过代码指定依赖关系*, [*使用MEF指定按需加载*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec42)*模块*, [*使用MEF为远程下载准备一个模块*](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec43) | [**更多**](http://msdn.microsoft.com/en-us/library/gg405479(v=pandp.40)#sec44) |

一个模块化应用被划分为若干功能集单元(称为模块)，然后集成到一个大的应用中的。客户端模块封装应用程序的整体功能的一部分，通常代表一系列相关的概念。它可以包含相关的组件，比如应用功能，包括用户界面和业务逻辑，或者应用基础结构件，比如日志和用户验证这些应用级别服务。模块彼此独立，但可以松耦合方式相互通讯。模块化应用可以更容易开发，测试，部署和扩展你的应用。

比如，考虑一个个人银行业务应用程序。用户可以多种功能，如账户间转账，付账单，以及从单用户界面(UI)更新个人信息。然而，在后台，每个功能都封装在一个独立的模块内。这些模块相互之间可以通讯，通过后段系统，比如数据库和Web服务。应用服务将每个不同的模块内的各种组件集成，以及处理拥护间的交流。用户可以看到一个像整体一样的应用。

下图显示了一个有多个模块的模块化应用的设计。

**模块组成**



# 创建模块化应用的好处

你可能已经准备好创建一个架构良好的应用，使用组件，接口和类，并且采用良好的面向对象设计原则。即便如此，除非非常谨慎，否则应用设计仍然可能是“铁板一块” (以紧耦合方式实现应用的所有功能)，这会使应用非常难开发，测试，扩展和维护。

另一方面，模块化应用的方法，可以帮助你确定确定应用的主要功能，并允许你独立的开发和测试。这可以让开发和测试更容易，并使应用在未来变更时更加灵活和易于扩展。模块化方法的好处是，可以让你的应用总体架构更灵活和可维护，因为它允许应用划分为若干可管理部分。每部分封装特定的功能，每部分干净独立，通过松耦合方式通讯。

## Prism对模块化应用开发的支持

Prism提供对模块化应用开发的支持，以及在运行时管理模块。使用Prism的模块化开发功能可以节省时间，因为你不需要实现和测试自己的模块化框架。Prism提供以下模块化应用开发功能：

* 模块目录，用于注册命名模块及管理模块定位；你可以创建模块目录，通过以下方式：
  + 通过编码方式或在XAML中定义
  + WPF：可以发现指定目录下的模块，因此可以加载所有模块，无需在目录中显式定义。
  + WPF：通过配置文件定义模块
* 模块的元数据声明属性支持初始化模式和依赖模式。
* 与依赖注入容器集成，支持模块的松耦合方式集成。
* 针对模块加载：
  + 依赖关系管理，包括重复和循环引用检测，以确保模块是按照正确的顺序加载，及只加载并初始化一次。
  + 支持按需和后台下载模块，以最小化应用启动时间；模块的其余部分可以当使用的时候，才将其在后台加载和初始化。

# 核心概念

本章介绍了Prism中模块化的核心概念，包括**IModule**接口，模块加载流程，模块目录，模块间通讯，和依赖注入容器。

## IModule:模块化应用的构造块

模块是一个逻辑集合，将功能和资源以可以独立开发，测试，发布的方式打包，然后集成到一个应用中。一个包可以是一个或多个组件，无论是作为一个松散集合还是作为XAP文件捆绑在一起。每个模块都有一个中心类，负责模块初始化和集成模块功能到应用。该类实现**IModule**接口。拥有了实现**IModule**接口的类，也就可以将一个应用集看成一个模块。**IModule**接口有一个**Initialize**方法，你可以在其中实现所有逻辑，关于模块初始化及将模块功能集成到应用。根据模块的目的，可以注册View到UI组件，为应用提供额外服务，或者扩展应用的功能。下面的代码演示了如何以最简模式实现一个模块。

**C#**

public class MyModule : IModule

{

public void Initialize()

{

// Do something here.

}

}

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| Stock Trader RI并没有使用**IModule**接口所提供的初始化机制，取而代之的是使用声明，基于属性的方法来注册View，服务和类型。 |

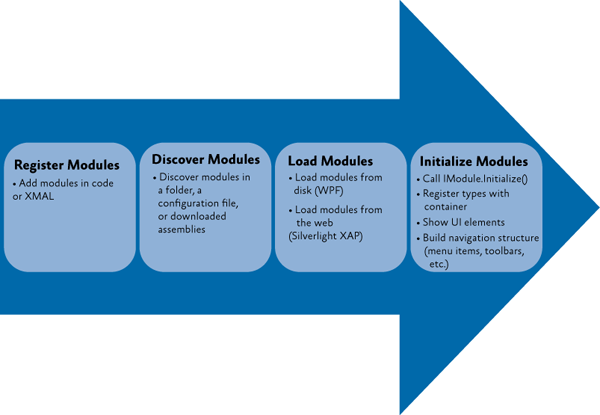
## 模块生命周期

Prism的模块加载流程包含以下：

1. **注册/发现模块**. 特定应用在运行时加载模块，是在模块目录中定义的。目录包含被加载模块信息，他们的位置，以及他们被加载的顺序。
2. **加载模块**. 包含模块的应用集被加载到内存。这可能需要将模块从Web上，或者其他远程位置，或者本地目录下载。
3. **初始化模块**. 模块初始化。意味着创建模块实例，并通过**IModule**接口调用**Initialize**方法。

下图显示模块加载流程。

**模块加载流程**



## 模块目录

**ModuleCatalog**管理应用可以使用的关于模块的信息。该目录本质上是**ModuleInfo**类的集合。每个模块由**ModuleInfo**类描述，记录着名城，类型，位置以及模块的其他属性。有几种典型的方法，使用**ModuleInfo**实例填充**ModuleCatalog**：

* 通过代码注册模块
* 通过XAML注册模块
* 通过配置文件注册模块 (仅针对WPF)
* 通过发现注册本地目录中的模块 (仅针对WPF)

注册和发现机制应该根据应用需求而使用。使用配置文件或者XAML文件，允许你的应用不引用这些模块。使用本地目录发现可以允许应用无需指定任何配置文件。

## 控制加载模块时机

Prism应用可以以最快方式初始化模块，称之为"可用时"加载，或者当应用需要他们的时候加载，称之为"按需"加载。针对Silverlight应用，模块可以和应用一起下载，或者在应用启动后，在后台下载。选择加载模块方式，请考虑以下准则：

* 应用启动时必须的模块，应该和应用一起下载，并在应用启动时初始化。
* 包含应用常用功能的模块，可以在后台下载，并在可用时初始化。
* 包括不常用功能的模块， (或者是可选的依赖模块)可以在后台下载，并在需要时初始化。

综合考虑应用实际，共通功能，启动时间，以及下载大小。然后决定如何配置模块的下载和初始化。

## 在应用中集成模块

Prism提供下面两个类用于引导应用： **UnityBootstrapper**或者**MefBootstrapper**。这些类可以用来创建和配置模块管理，以查找和加载模块。你可以重写配置方法，使得注册模块可以通过XAML文件，或者配置文件，或者查找本地目录。这只需要几行代码。

使用模块的**Initialize**方法将模块和应用的其他部分集成到一起。根据应用结构和模块内容的不同，实现方法各不相同。以下是集成模块到应用时，通常需要考虑的事项：

* 添加模块的View到应用的导航中。这常见于构建使用View发现或View注入的复合UI应用时。
* 订阅应用级别事件或服务。
* 与应用的依赖注入容器注册共享服务。

## 模块之间的通信

尽管模块需要尽降低相互之间的耦合性，他们之间仍需通讯，这里是常见的集中松耦合通讯模式，每个都有自己的长处。通常情况下，结合他们的优点，建立最终解决方案。下面是这些模式：

* **松耦合事件**.模块可以广播某一特定事件。另外一些模块可以订阅这些事件，他们会在事件发生的时候收到通知。松耦合事件是两个模块间通讯的轻量级方式。因此很容易实现。但是，如果设计过分依赖事件，就会变得很难维护，尤其是多事件要协同一起完成单个任务时。这个时候，共享服务是更好的选择。
* **共享服务**. 共享服务是一个类，可以通过共用接口访问。通常，共享服务位于共享应用集中，提供全系统服务，比如身份验证，日志，或者配置。
* **共享资源**. 如果你不希望模块间直接通讯，可以间接的通过共享资源通讯，比如数据库或者Web服务。

## 依赖注入和模块化应用

容器，比如Unity和MEF使你可以轻松的使用反转控制 (IoC)和依赖注入，他们是强大的设计模式，以帮助以松耦合方式编写组件。允许组件可以获取另外组件的依赖，而不必通过硬编码方式引用这些模块，从而促进更好的代码重用和更好的灵活性。依赖注入在创建松耦合，模块化应用时非常有用。Prism是面向非特定依赖注入容器的。因此容器的使用取决于你的应用需求和偏好。不过这里有来自Microsoft的两个主要依赖注入框架 – Unity和MEF。

patterns & practices的Unity提供了一个全功能的依赖注入容器。它支持基于属性和基于构造函数的注入以及Policy注入，允许在组件间透明的注入行为和Policy；此外，还支持其他功能，是典型的依赖注入容器。

MEF (现在是.NET Framework 4和Silverlight 4的一部分) 为创建可扩展的.NET应用提供支持，通过支持基于依赖注入的组件，并提供其他功能，支持模块化应用开发。它允许应用在运行时发现组件，然后将这些组件以松耦合模式集成到应用程序。MEF是一个非常棒的扩展开发框架。它包括应用集和类型发现，类型依赖项解析，依赖注入，和一些不错的应用集和XAP下载功能。Prism支持MEF的功能，及以下优势：

* 关联模块类型到XAP
* 通过XAML和编码属性注册模块，适用于WPF和Silverlight
* 通过配置文件和扫描本地目录注册模块，适用于WPF
* 模块加载后状态跟踪
* 使用MEF时自定义模块的声明元数据

Unity和MEF依赖注入容器和Prism无缝集成工作。

# 关键决策

你需要做的第一个决定是，是否需要开发模块化解决方案。建立模块化应用，如之前所述，有很多好处，但同时你需要花费时间和精力去真正获得这些好处。一旦你决定开发模块化解决方案，以下这些问题是需要考虑的：

* **确定将要使用的框架**. 你可以创建自己的模块化框架，使用Prism，MEF，或其他框架。
* **确定如何组织解决方案**. 通过定义每个模块的边界，包括模块中应该包含哪些应用集。你可以决定为每个开发使用模块化，以及控制如何部署应用或是应用否支持插件或扩展体系。
* **确定如何划分模块**. 模块可以根据不同需求进行划分，比如，按功能，供应商，开发团队或开发需求。
* **确定应用需要对所有模块提供的核心服务**. 比如说异常处理服务，或者身份验证服务，或者授权服务。
* **如果使用Prism，确定在模块目录中使用注册模块的方式**. 针对WPF，你可以通过编码，XAML，配置文件，扫描本地目录来注册模块。针对Silverlight，你可以通过编码或者XAML来注册模块。
* **确定模块间通讯和依赖策略**.模块之间需要相互通讯，并且需要解决模块间的依赖关系。
* **确定依赖注入容器**.通常，模块化系统需要依赖注入，反转控制，或者服务定位，以允许松耦合及动态的加载创建模块。Prism可以在使用Unity，MEF，或其他容器中选择，并且提供用于基于Unity或MEF的应用的库。
* **最小化应用启动时间**.考虑将模块按需加载和后台下载，以达到应用启动时间最小化。
* **确定部署要求**.你需要考虑将打算以何种方式部署应用。这将决定你放在XAP包中的应用集数量。你也许还需要共享库，比如在Silverlight中，Prism会使用应用集缓存这些高级特性。

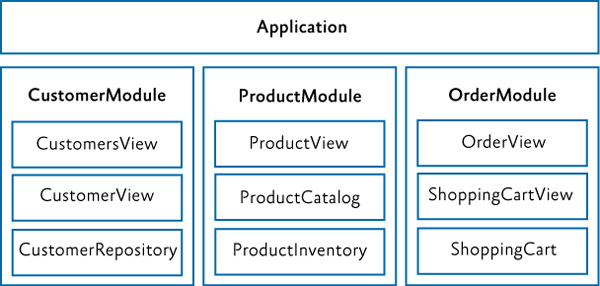
下一章提供了如何确定这些问题的细节。

## 把你的应用划分成模块

当你以模块化方式开发应用的时候，你会将应用结构划分为独立的客户端模块，以便独立开发，测试和部署。每个模块将封装应用的总体功能的一部分。因此在设计应用时第一个决策就是，如何将应用功能划分到离散的模块。

一个模块封装一组相关的问题，并有一套独立的职责。一个模块可以是应用的垂直切片或者水平的服务层。大型应用一般都会有两种类型的模块。

**围绕垂直功能切分应用组织模块**



**围绕水平功能切分应用组织模块**



一个大型应用模块的划分方式可能包括垂直或水平。下面是一些划分的示例：

* 一个包含了特定的应用功能的模块，比如在Stock Trader引用实现(Stock Trader RI)中的新闻模块。
* 一个包含了特定子系统或者一组相关功能的模块，比如采购，票据或总账功能。
* 一个包含了基础服务的模块，比如日志，缓存，和认证服务，或者Web服务。
* 一个包含了调用生产线 (LOB)系统的模块，比如Siebel CRM和SAP，以及其他内部系统。

模块和模块之间的依赖应当达到最小化。当一个模块与另一模块存在依赖项时，应使用松耦合模式，使用定义在共享服务中的接口，而不是直接使用该模块，或者使用**EventAggregator**，通过**EventAggregator**事件类型和其他模块进行通讯。

模块化的目标是令应用保持灵活，易维护，稳定，甚至在往应用中追加或删除的某些技术的时候。完成这一任务的最好方式是，设计应用的时候，模块尽可能保持独立，有语义明确的接口。

### 决定模块的粒度

有几种方法用来创建和打包模块。推荐和常用的方法是为每个模块创建一个单独的应用集。这有助于保持独立的逻辑单元，并促进适当封装。这还更清晰的表述以应用集作为模块边界，并有助于部署模块。然而，没有什么可以防止一个应用集中加入多个模块，有时候这是首选，尽量减少解决方案中的工程数目。对于大行业能够用来说，有10-50个模块很正常。把每个模块都分割为一个工程，会增加解决方案的复杂度，也会让Visual Studio变慢。如果，你坚持让每个模块成为一个独立的应用集/Visual Studio工程，可以考虑将模块或者关联的模块集划分为独立的解决方案。

### XAP和模块封装

Silverlight应用中，模块通常打包在单独的XAP文件中，但是在某些情况下，可能需要在一个XAP中包含多个模块。应该考虑XAP文件的个数，以便在应用启动和启动新功能的时候，尽量减少下载请求的XAP文件的大小和个数。如果你选择把每个模块分割成它自己的工程/应用集，你需要确定是否把每个模块放到它自己部署的XAP里面，或者在一个XAP中包含多个应用集。

下面是一些影响你选择是否要在单个XAP中包含多个模块，或者将他们分开的因素：

* **下载大小和共享依赖关系**. 每个XAP文件由于需要管理清单及.zip文件打包，存在少量的额外开销。此外，如果模块间的通用依赖没有被划分成独立的模块，或者缓存库，每个XAP都会包括这些依赖库，将会大大提高下载大小。
* **应用需要多个模块的时机**. 如果多模块同时被加载和使用，比如在应用启动的时候显示的View，把他们打包到一个单独的XAP文件，可能会使下载速度稍快，并有助于确保这两个模块可以同时在应用中使用。Prism的模块化功能可以确保模块及依赖项能以正确的顺序加载，因此不会发生模块被划分为多个XAP之后，加载顺序不正确的问题。但是使用多个XAP文件可能会带来少量的额外性能开销，即使多个XAP的总下载大小和一个是一样。
* **模块版本**. 如果不同的模块有单独的开发及部署时间线，你可能需要把他们放到不同的XAP文件中，以更清晰的标记为不同的版本，并可以独立更新。

为了避免同一个应用集被下载两次，这里有两个办法：

* 共享依赖到单独的基础结构模块，其他模块使用该共享模块。
* 使用Silverlight中的 Assembly Library Caching，将共享类型放到只下载一次的共享库中，并使用Silverlight将它缓存，而不使用Prism的模块下载器。

## 松耦合中使用依赖注入

一个模块可能需要依赖宿主应用或者其他模块提供的组件和服务。Prism支持注册模块间依赖关系，以便他们按正确的顺序加载和初始化。Prism也支持模块加载到应用之后初始化。在模块初始化时，模块可以检索它需要的附加组件或服务，并/或 注册它包含的其他组件或服务，使其可被其他模块调用。

模块应当使用一个独立的机制来获取外部接口，而不是直接使用具体类，比如通过依赖注入容器或工厂服务。依赖注入容器，如Unity或MEF，允许通过依赖注入自动获取接口或类型的实例。Prism集成了Unity和MEF允许模块自由的使用依赖注入。

下面的关系图显示了典型的操作序列，当模块被加载时，请求或者注册组件和服务的引用。

**依赖注入示例**



该实例中， **OrdersModule**应用集定义一个**OrdersRepository**类(用于其他View和类实现预定功能)。**CustomerModule**应用集定义一个**CustomersViewModel**类，需要依赖**OrdersRepository**，基于服务所暴露的接口。应用在启动的时候，引导过程包括以下步骤：

1. Bootstrapper启动模块初始化，模块加载器加载和初始化**OrdersModule**。
2. 在**OrdersModule**初始化时，它将**OrdersRepository**注册到容器。
3. 模块加载器然后加载**CustomersModule**。模块加载顺序可以通过模块元数据指定。
4. **CustomersModule**使用容器解析生成**CustomerViewModel**实例。**CustomerViewModel**依赖**OrdersRepository** (通常基于它的接口) 通过构造函数或者属性注入。容器通过ViewModel的构造函数注入依赖关系，基于由**OrdersModule**注册的类型。最终结果是，从**CustomerViewModel**到**OrderRepository**的引用是通过接口的引用，而没有类之间的紧耦合。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 用于暴露**OrderRespository** (**IOrderRepository**)的接口可以放在一个独立的"共享服务"应用集或一个"请求服务"应用集中，应用集中仅包含需要暴露的服务的接口和请求类型。这样，在**CustomersModule**and和**OrdersModule**间就不存在硬依赖关系了。 |

1. 注意: 所有的模块和依赖注入容器有一个隐式的依赖关系。这种依赖关系在模块构造时被通过模块加载器注入。

# 核心场景

本章介绍在应用中使用模块时所遇到的常见场景。这些场景包括定义模块，注册和发现模块，加载模块，初始化模块，指定模块依赖关系，按需加载模块，在后台下载远程模块，检测模块是否加载完毕。你可以注册和发现模块，通过编码，XAML文件或者应用配置文件，或者扫描本地目录。

## 定义模块

模块是一组功能或者资源的集合，可以独立开发，测试，部署和集成到应用中。每个模块有一个中心类，用于初始化模块，以及将模块功能集成到应用去。这个类实现**IModule**接口，如下所示。

**C#**

public class MyModule : IModule

{

public void Initialize()

{

// Initialize module

}

}

实现**Initialize**方法根据应用而定。模块类的类型，初始化模式，及其他的依赖模块被定义在模块目录中。在目录中的每个模块，模块加载器都会为该模块类创建一个实例,然后调用**Initialize**方法。模块按照模块目录中指定的顺序进行处理。运行时初始化顺序是基于模块的下载，可用性，及是否满足依赖关系的。

根据应用所使用的模块目录，模块依赖关系可以通过在模块类本身使用声明属性，或者通过模块目录文件进行配置。下面章节将提供更多详细信息。

## 注册和发现模块

应用能加载的模块都被定义在模块目录中。Prism模块加载器通过模块目录决定模块的可用性，以及加载的顺序。

模块目录是实现**IModuleCatalog**接口的类。在应用初始化时，由应用Bootstrapper类创建。根据你的选择，Prism提供了各种不同的模块目录实现。你也可以从另外的数据源发行一个目录模块，通过调用**AddModule**方法或通过继承**ModuleCatalog**自定义一个模块目录。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 通常情况下，Prism中的模块使用模块注入容器和通用服务定位器来检索模块初始化时请求的类型的实例。Prism支持Unity和MEF容器。虽然大体流程上一致，都包括注册，发现，下载和初始化，但是具体实现细节需要根据使用Unity或MEF有所不同。本章主题解释容器的特定方法之间的差异。 |

### 通过代码注册模块

由**ModuleCatalog**类提供了最基本的模块目录。你可以是用此模块目录以编码方式，通过指定模块类的类型，来注册模块。此外，你也可以以编码方式，指定模块名和初始化方式。要直接将模块和**ModuleCatalog**类注册，在应用的**Bootstrapper**类中，调用**AddModule**方法。下面是示例代码。

**C#**

protected override void ConfigureModuleCatalog()

{

Type moduleCType = typeof(ModuleC);

ModuleCatalog.AddModule(

new ModuleInfo()

{

ModuleName = moduleCType.Name,

ModuleType = moduleCType.AssemblyQualifiedName,

});

}

上述的例子中，模块被Shell直接引用，模块类的类型被定义并可以用在**AddModule**中。这就是为什么在例子中使用**typeof(Module)**，将模块添加到目录。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 如果你的应用是直接引用模块类型，你可以像上面演示那样添加类型；否则你需要完整的类型名称和应用集的位置。 |

通过代码定义模块目录的另一个示例，请查阅Stock Trader引用实现(Stock Trader RI)的StockTraderRIBootstrapper.cs。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| **Bootstrapper**基类提供**CreateModuleCatalog**方法，协助创建**ModuleCatalog**。默认情况，该方法创建一个**ModuleCatalog**实例，但是这个方法可以被扩展类重写，以创建不同的模块目录。 |

### 通过XAML注册模块

你可以通过在XAML中声明的方式定义一个模块目录。XAML文件指定创建什么样的模块目录，以及添加什么模块到其中。通常，.xaml文件是作为资源文件添加到Shell工程中的。模块目录是由Bootstrapper调用**CreateFromXaml**方法来创建的。从技术角度看，这种方法和通过编码方式定义**ModuleCatalog**非常相似，因为XAML文件只是定义需要实例化对象的层次结构。

下面的代码示例演示了如何使用XAML文件指定模块目录。

**XAML ModularityWithUnity.Silverlight\ModulesCatalog.xaml**

<Modularity:ModuleCatalog xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

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

xmlns:sys="clr-namespace:System;assembly=mscorlib"

xmlns:Modularity="clr-namespace:Microsoft.Practices.Prism.Modularity;assembly=Microsoft.Practices.Prism">

<Modularity:ModuleInfoGroup Ref="ModuleB.xap" InitializationMode="WhenAvailable">

<Modularity:ModuleInfo ModuleName="ModuleB" ModuleType="ModuleB.ModuleB, ModuleB, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

</Modularity:ModuleInfoGroup>

<Modularity:ModuleInfoGroup InitializationMode="OnDemand">

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

<Modularity:ModuleInfo Ref="ModuleF.xap" ModuleName="ModuleF" ModuleType="ModuleF.ModuleF, ModuleF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" >

<Modularity:ModuleInfo.DependsOn>

<sys:String>ModuleE</sys:String>

</Modularity:ModuleInfo.DependsOn>

</Modularity:ModuleInfo>

</Modularity:ModuleInfoGroup>

<!-- Module info without a group -->

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

</Modularity:ModuleCatalog>

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| **ModuleInfoGroups** 提供一种简便的组织模块的方式。用于同一个.xap文件或者应用集，或者以同样的初始化方式，或者有同样的依赖关系的模块。可以定义在相同**ModuleInfoGroup**中的模块的相互依赖关系，但不能定义在不同的**ModuleInfoGroups**中的模块间的依赖关系。把模块放到组内是可选的。设置组的属性，该属性将应用于组内所有模块。注意:模块不放在组内也可以注册。 |

在应用的**Bootstrapper**类中，需要指定XAML文件作为**ModuleCatalog**的源，如下所示。

**C#**

protected override IModuleCatalog CreateModuleCatalog()

{

return ModuleCatalog.CreateFromXaml(new Uri("/MyProject.Silverlight;component/ModulesCatalog.xaml",

UriKind.Relative));

}

### 通过配置文件注册模块

在WPF中，可以通过App.config文件指定模块信息。这种方法的优点是，不会把该文件编译到应用中。这使得它在运行时非常容易的添加或者移除模块，而无需重编译应用。

下面的代码演示了如何使用配置文件指定模块目录。如果你想模块被自动加载，设置**startupLoaded="true"**。

**XML ModularityWithUnity.Desktop\app.config**

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

<configuration>

<configSections>

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

</configSections>

<modules>

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

<module assemblyFile="ModularityWithUnity.Desktop.ModuleF.dll" moduleType="ModularityWithUnity.Desktop.ModuleF, ModularityWithUnity.Desktop.ModuleF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" moduleName="ModuleF" startupLoaded="false">

<dependencies>

<dependency moduleName="ModuleE"/>

</dependencies>

</module>

</modules>

</configuration>

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 不管你的应用集是在全局应用集缓存还是和应用放在同一文件夹下， **assemblyFile**属性是必须的。该属性的作用是将**moduleType**映射到**IModuleTypeLoader**。 |

在应用的**Bootstrapper**类中，你需要指定配置文件作为**ModuleCatalog**的源。为了做到这一点，你需要使用**ConfigurationModuleCatalog**类，如下所示。

**C#**

protected override IModuleCatalog CreateModuleCatalog()

{

return new ConfigurationModuleCatalog();

}

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 你仍可以在**ConfigurationModuleCatalog**中，通过编码模式添加模块。你可以这样用，比如，确保你应用绝对需要的功能被定义到目录中。 |

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| Silverlight不支持使用配置文件。如果你希望在Silverlight中使用这种配置文件风格，推荐方式是创建自己的**ModuleCatalog**，从服务器上的Web服务读取模块的配置文件。 |

### 发现文件夹下的模块

在WPF中，Prism的**DirectoryModuleCatalog**类允许指定一个本地文件夹作为模块目录。该模块目录会扫描指定的文件夹，并检索可定义到应用中的应用集。为了做到这一点，需要对模块类使用声明属性，指明模块的名称，和它们的依赖关系。下面代码演示了如何发现本地目录的应用集，并填充模块目录。

**C#**

protected override IModuleCatalog CreateModuleCatalog()

{

return new DirectoryModuleCatalog() {ModulePath = @".\Modules"};

}

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| Silverlight不支持此功能，因为Silverlight安全机制不允许你从本地目录加载应用集。 |

## 加载模块

填充完**ModuleCatalog**之后，模块随时可以加载和初始化。加载模块意味着模块应用集从硬盘文件转为内存存储。如果应用集不存在硬盘，则需要先进行检索。比如使用Silverlight .xap 文件，需要先从网络下载。**ModuleManager**用于协调加载和初始化过程。

## 初始化模块

模块加载后，开始初始化。意味着模块类的实例被创建，然后调用**Initialize**方法。初始化方法是将模块的功能集成到应用的地方，考虑模块初始化时有可能要做的几个事情：

* **将模块的View注册到应用**.如果模块参与UI组合，通过View发现或者View注入，模块需要把它的View或者View Model关联到相应Region。这使得View可以动态的显示在菜单，工具栏，或者应用其他的可视化区域中。
* **订阅应用级别的事件或服务**. 通常情况下，应用公开模块有可能用到的特定的应用服务和/或事件。通过使用**Initialize**方法，将模块的功能添加到这些应用级别的服务和事件中。

比如，应用会在它关闭的时候触发一个事件。模块希望对该事件做出反应。也有可能模块需要提供些数据给应用级别的服务，比如，如果你创建了一个**MenuService** (负责添加和删除菜单项)，模块的**Initialize**方法就是你添加模块菜单项的地方。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 默认情况下，模块生命周期被设置最小化。当**Initialize**在加载过程中被调用之后，模块的实例引用就被释放了。如果模块实例没有强引用链的话，它会被垃圾回收器正确回收。该行为在调试的时候可能会有问题，如果你订阅了事件并保有弱引用到你的模块，因为你的模块在垃圾回收器工作的时候， "消失"了。 |

* **注册类型到依赖注入容器**. 如果你使用依赖注入容器模式，比如使用Unity或MEF，模块会为应用或其他用到的模块注册类型。它可能会要求容器解析一个它需要的类型的实例。

## 指定模块依赖关系

模块可能需要依赖别的模块。如果模块A依赖模块B，模块B必须在模块A前初始化。**ModuleManager**跟踪这些依赖项并以此初始化模块。根据定义模块目录的方式，你可以通过编码，配置文件，或XAML定义模块依赖关系

### 通过代码指定依赖关系

WPF应用通过代码注册模块或者在本地目录中扫描模块， Prism提供了创建一个模块时所需要用到的声明属性，如下所示。

**C# (使用Unity)**

[Module(ModuleName = "ModuleA")]

[ModuleDependency("ModuleD")]

public class ModuleA: IModule

{

...

}

## 通过XAML指定依赖关系

下面的XAML演示了依赖模块E的模块F。

**XAML ModulesCatalog.xaml**

<Modularity:ModuleInfo Ref="ModuleF.xap" ModuleName="ModuleF" ModuleType="ModuleF.ModuleF, ModuleF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" >

<Modularity:ModuleInfo.DependsOn>

<sys:String>ModuleE</sys:String>

</Modularity:ModuleInfo.DependsOn>

</Modularity:ModuleInfo>

### 通过配置文件指定依赖关系

下面的App.config文件演示了依赖模块B的模块D。

**XML App.config**

<modules>

<module assemblyFile="Modules/ModuleD.dll" moduleType="ModuleD.ModuleD, ModuleD" moduleName="ModuleD">

<dependencies>

<dependency moduleName="ModuleB"/>

</dependencies>

</module>

## 按需加载模块

若要按需加载模块，你需要指定加载模式**InitializationMode**为**OnDemand**。之后，在应用中编写关于请求加载模块的代码。

### 通过代码指定按需加载

模块使用属性指定为按需加载，如下所示。

**C# Bootstrapper.cs**

protected override void ConfigureModuleCatalog()

{

Type moduleCType = typeof(ModuleC);

this.ModuleCatalog.AddModule(new ModuleInfo()

{

ModuleName = moduleCType.Name,

ModuleType = moduleCType.AssemblyQualifiedName,

InitializationMode = InitializationMode.OnDemand

});

}

### 通过XAML指定按需加载

你可以在使用XAML定义模块目录的时候，使用**InitializationMode.OnDemand**，如下所示。

**XAML ModulesCatalog.xaml**

...

<Modularity:ModuleInfoGroup InitializationMode="OnDemand">

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

...

### 通过配置文件指定按需加载

你可以在使用App.config定义模块目录的时候，使用**InitializationMode.OnDemand**，如下所示。

**XML App.config**

...

<module assemblyFile="Modules/ModuleC.dll" moduleType="ModuleC.ModuleC, ModuleC" moduleName="ModuleC" startupLoaded="false"/>

...

### 加载指定模块

在模块被指定为按需加载之后，应用可能请求加载模块。想要启动加载的代码，需要获得被Bootstrapper注册到容器中的**IModuleManager**服务的引用。

**C#**

private void OnLoadModuleCClick(object sender, RoutedEventArgs e)

{

moduleManager.LoadModule("ModuleC");

}

## 在后台下载远程模块

当应用启动之后或者只有在用户使用的时候，才通过后台下载模块，可以改善应用的启动时间。

### 为远程下载准备一个模块

在Silverlight应用中，模块被打包到.xap文件中。若要下载一个从应用中分离出的模块，需要创建一个独立的.xap文件。你可以选择把多个模块放到单个.xap文件中，以优化下载请求数及每个.xap文件的大小。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 对于每个.xap文件，你需要创建一个新的Silverlight应用工程。在Visual Studio 2008 和2010中，只有应用工程会生成独立的.xap文件。在这些工程中，不需要App.xaml或者MainPage.xaml文件。 |

### 跟踪下载进度

**ModuleManager**类提供了一个事件，使应用可以跟踪模块下载进度。除下载百分比外，它提供了已下载字节数及需下载的总字节数。你可以据此给用户显示模块下载进度。

**C#**

this.moduleManager.ModuleDownloadProgressChanged += this.ModuleManager\_ModuleDownloadProgressChanged;

**C#**

void ModuleManager\_ModuleDownloadProgressChanged(object sender,

ModuleDownloadProgressChangedEventArgs e)

{

...

}

## 检测模块是否加载完毕

**ModuleManager**服务提供了一个事件，使应用可以跟踪模块下载是否完成或者失败。你可以通过**IModuleManager** 的依赖赖注入获取该服务的引用。

**C#**

this.moduleManager.LoadModuleCompleted += this.ModuleManager\_LoadModuleCompleted;

**C#**

void ModuleManager\_LoadModuleCompleted(object sender, LoadModuleCompletedEventArgs e)

{

...

}

为保持应用和模块松耦合，应用应当避免使用该事件来集成模块到应用中。相反，模块的**Initialize**方法应该处理和应用集成的事件。

**LoadModuleCompletedEventArgs** 包含了一个**IsErrorHandled**属性。如果模块文件加载失败，并且应用希望防止**ModuleManager**记载该错误及抛出异常，需要将此属性设置为**true**。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 当模块被加载并被初始化之后，模块应用集不能被卸载。模块实例引用不会被Prism库使用，因此在模块实例在初始化结束之后，会被垃圾回收器清理。 |

## MEF中的模块

本章仅针对使用MEF作为依赖注入容器所要注意的差异。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| 当使用MEF时， **MefBootstrapper**调用的是**MefModuleManager**。它扩展自**ModuleManager**并实现了**IPartImportsSatisfiedNotification**接口，确保**ModuleCatalog**在新的类型被MEF导入的时候被更新。 |

### 使用MEF通过代码注册模块

当使用MEF时，可以应用**ModuleExport**属性到模块类，使MEF可以自动发现该类型。以下是示例。

**C#**

[ModuleExport(typeof(ModuleB))]

public class ModuleB : IModule

{

...

}

你也可以使用MEF的**AssemblyCatalog**类发现并加载模块，该类可以发现一个应用集中所有被导，出的模块类。以及**AggregateCatalog**类，该类允许多目录合并为一个逻辑目录。默认情况下， Prism的**MefBootstrapper**类创建一个**AggregateCatalog**实例。你可以重写**ConfigureAggregateCatalog**方法注册应用集，如下示例代码所示。

**C#**

protected override void ConfigureAggregateCatalog()

{

base.ConfigureAggregateCatalog();

//Module A is referenced in in the project and directly in code.

this.AggregateCatalog.Catalogs.Add(

new AssemblyCatalog(typeof(ModuleA).Assembly));

this.AggregateCatalog.Catalogs.Add(

new AssemblyCatalog(typeof(ModuleC).Assembly));

}

Prism的**MefModuleManager**保证MEF的**AggregateCatalog**和Prism的**ModuleCatalog**同步，从而允许Prism发现通过**ModuleCatalog**或者**AggregateCatalog**添加的模块。

|  |
| --- |
| **Gg405479.note(en-us,PandP.40).gif注意:** |
| MEF使用**Lazy<T>**扩展，防止类型实例的导出导入，直到**Value**属性被调用为止。 |

### 使用MEF发现文件夹下的模块

MEF提供一个**DirectoryCatalog**用于检查目录中的应用集是否包含模块 (以及其他被MEF标记为导出的类型)。这种情况下，你可以重写**ConfigureAggregateCatalog**方法注册目录。这种方式只适用于WPF。

要使用这种方法，你首先需要使用**ModuleExport**属性，将模块的名字和依赖关系应用于你的模块，如下代码所示。这允许MEF导入模块，并允许Prism保持**ModuleCatalog**更新。

**C#**

protected override void ConfigureAggregateCatalog()

{

base.ConfigureAggregateCatalog();

DirectoryCatalog catalog = new DirectoryCatalog("DirectoryModules");

this.AggregateCatalog.Catalogs.Add(catalog);

}

### 使用MEF通过代码指定依赖关系

对于使用MEF的WPF应用，使用**ModuleExport**属性，如下所示。

**C# (使用MEF)**

[ModuleExport(typeof(ModuleA), DependsOnModuleNames = new string[] { "ModuleD" })]

public class ModuleA : IModule

{

...

}

由于MEF允许你在运行时发现模块，你也可以在运行时发现新的模块间的依赖关系。虽然你可以混合使用MEF和**ModuleCatalog**，但务必要记住，当使用**ModuleCatalog**验证依赖关系链，从XAML或者配置文件加载时 (在模块加载之前)，如果模块先被**ModuleCatalog**加载，然后使用MEF加载，会使用**ModuleCatalog**的依赖关系，**DependsOnModuleNames**属性会被忽略。混合使用MEF及**ModuleCatalog**，通常出现在Silverlight应用中，独立的XAP文件包含模块的情况。

### 使用MEF指定按需加载模块

如果你使用MEF和**ModuleExport**属性指定模块及模块依赖关系，你可以使用**InitializationMode**属性指定模块是否按需加载，如下所示。

**C#**

[ModuleExport(typeof(ModuleC), InitializationMode = InitializationMode.OnDemand)]

public class ModuleC : IModule

{

}

### 使用MEF为远程下载准备一个模块

默认机制下，Prism应用使用MEF，使用MEF的**DeploymentCatalog**类来下载.xap文件并发现该.xap文件中的应用集和类型。**MefXapModuleTypeLoader**添加每个**DeploymentCatalog**到**AggregateCatalog**。

如果两个不同的.xap文件被追加，并且包含相同的共享应用集，相同的类型被二次导入。如果该类型是准备作为单例，并且在两个模块之间共享的话，这会导致重组异常。**Microsoft.Practices.Prism.MefExtensions.dll**就是这样一个例子。

为避免导入冲突，打开每个模块工程的引用，把共享DLLs设置为**'Copy Local'=false**.这可以避免应用集被打包到模块的.xap文件，防止二次导入。同样，这也可以降低每个.xap文件的大小。你需要确保应用对该共享应用集的引用或者包含该共享应用集的包，在模块的.xap文件使用之前被下载。

# 更多

更多关于程序集缓存的详细信息，请查看MSDN的"How to: Use Assembly Library Caching"：<http://msdn.microsoft.com/en-us/library/dd833069(VS.95).aspx>.

若要了解更多关于Prism的模块化设计，请查看使用MEF的WPF版模块化快速入门或者使用Unity的WPF版模块化快速入门。更多关于这些快速入门的详细信息，请查阅 [WPF模块化快速入门](http://msdn.microsoft.com/en-us/library/ff921068(v=pandp.40)).

若要了解如何扩展Prism库中的模块化功能，请查阅在"[扩展Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))"中的"[模块](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40)#Modules)"。

# 5: 实现MVVM模式

|  |
| --- |
| **索引:** |
| [**类的职责和特点**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec1) | [**View 类**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec2) | [**View Model类**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec3) | [**Model类**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec4) | [**类的交互**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec5) |[**数据绑定**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec6) – [*实现INotifyPropertyChanged*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec7), [*实现INotifyCollectionChanged*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec8), [*实现ICollectionView*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec9) |[**命令**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec10) – [*实现命令对象*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec11), [*从View中调用命令对象*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec12), [*从View中调用命令方法*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec13) |[**数据验证和异常报告**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec14) – [*实现IDataErrorInfo*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec15), [*实现INotifyDataErrorInfo*](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec16) | [**组装**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec17) |[**通过XAML创建View Model**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec18) | [**通过代码创建View Model**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec19) | [**创建View并定义为数据模版**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec20) | [**关键决策**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec21) | [**更多**](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#sec22) |

MVVM模式可以帮助你彻底把应用的业务逻辑和表现逻辑从它的UI中分离出来。维护一个分离干净的应用逻辑及UI可以帮助解决许多开发和设计问题，可以使应用更加容易测试，维护和扩展。也可以大大提高代码的复用机会，允许开发人员和UI设计师在专注他们各自负责的领域同时，更容易的合作。

使用MVVM模式，应用的UI，表现逻辑及业务逻辑被分城三个不同的类:View，封装了UI和UI逻辑；View Model，封装了表现逻辑和状态；Model，封装了应用的业务逻辑和数据。

Prism包括了示例以及引用实现，关于如何实现MVVM模式，在Silverlight或WPF应用。Prism库同样提供了帮助你在应用中应用该模式的功能。这些功能体现了实现MVVM模式最常见的做法，并被设计为支持可测试性及可与Expression Blend和Visual Studio很好的协同工作。

本章提供了MVVM模式的概述，并介绍了如何实现其基本特征。Chapter 6介绍了如何使用Prism库实现更高级的MVVM场景。

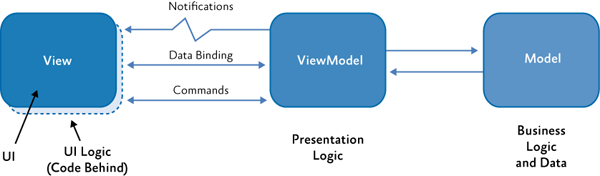
# 类的职责和特点

MVVM模式是Presentation Model模式的变体，优化了一些WPF和Silverlight的核心功能，比如数据绑定，数据模板，命令，和行为。

在MVVM模式中，View封装了UI以及其他UI逻辑，View Model封装了表现逻辑和状态，Model封装了业务逻辑和数据。View和View Model之间的交互是通过数据绑定，命令，和状态变更的通知事件。View Model查询，观察，和协调更新Model，转换，验证，和聚合View中显示所需要的数据。

下图描述了MVVM类和他们的之间的交互。

**MVVM类和他们之间的交互**



如其他所有的分离表示模式一样，有效使用MVVM模式的关键在于将应用划分为正确的层次，以适当的方法和理解进行编码，并且再各种场景中正确使用这些类。以下各节描述了MVVM模式中各个类的职责和特点。

## View类

View的职责是定义用户再屏幕上看到的结构和外观。理想情况下，View的后台代码，仅仅包括构造函数，及构造函数中调用的**InitializeComponent**方法。在某些情况下，后台代码可能包含在XAML中难以实现，或者低效的部分实现，比如复杂的动画，或者当代码需要直接操作View的一部分视觉元素的逻辑。你不应该把任何需要做单元测试的逻辑代码放到View中。通常，View的后台代码中的逻辑会通过UI自动测试方法进行测试。

在Silverlight和WPF中，View中的数据绑定是依赖于数据上下文的。在MVVM中，View的数据上下文被设置为View Model。View Model实现属性和命令，用于View的绑定，或当状态发生变化时，通过通知事件通知View进行变更。通常View和View Model的关系是一对一的。

通常，View是**Control**-派生类或**UserControl**-派生类。然而，一些情况下，View可以是数据模板，用于指定UI元素的可视化表示。通过使用数据模板，可视化设计器可以很容易的定义一个ViewModel如何显示，或者可以在不改变底层对象或者行为的前提下，改变控件默认的视觉效果。

数据模板可以看作是没有任何后台代码的View。它们被设计为绑定到一个特定的将会被显示在UI上的ViewModel。在运行时，View，定义的数据模板，将会被自动实例化，并且他的数据上下文被设置为响应的ViewModel。

WPF中，可以将数据模版和应用级别的ViewModel类型相关联。WPF将自动的将数据模版应用到任何需要表示在UI上的指定类型的ViewModel对象上。这就是所谓的隐式数据模板。Silverlight中，你必须显式的为需要显示的控件的ViewModel指定一个数据模版。不管哪种方式，数据模版既可以定义在使用它的控件中，也可以定义在View外的资源字典中然后通过声明方式将其结合到View的资源字典。

简而言之，View有以下主要特点：

* View是视觉元素，如窗口，页面，用户控件，或者数据模版。View定义了在View中的控件及他们的可视布局和样式。
* View通过**DataContext**属性引用ViewModel。View中的控件绑定ViewModel公开的属性和命令。
* View可以自定义View和ViewModel间的数据绑定行为。比如，View可以使用值转换器格式化用于显示在UI的数据，或者使用验证规则为用户提供额外的数据验证。
* View定义和处理UI视觉行为，比如动画或者ViewModel状态变化引发的改变，或者用户和UI交互带来的改变。
* View的后台代码可以定义UI逻辑，实现那些难以在XAML中定义或者需要直接操作指定UI控件的可视化行为。

## View Model类

MVVM模式中的ViewModel封装了显示逻辑和View中的数据。它没有直接引用View或者任何关于View的特定实现或者类型。ViewModel实现了属性和命令，为View提供绑定的数据及通过通知事件提供给View的状态变更通知。这些由ViewModel提供的属性和命令，定义了提供给UI的功能，但是如何使用这些功能，是由View决定的。

ViewModel是负责协调View和所需的Model之间的互动。通常，ViewModel和Model之间的关系是一对多的。ViewModel可能会选择性直接暴露Model给View，使View中的控件可以进行数据绑定。这种情况下，Model类需要设计为支持数据绑定和支持相关的通知事件。更多关于此方面的详细信息，请查阅[数据绑定](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#DataBinding)，在本章的后半部分。

由于ViewModel可以转换或者控制Model数据，因此View可以很方便的使用他们。ViewModel可以为指定的View定义附加的属性；这些属性通常不会是 (或者不能添加到)Model的一部分。比如，ViewModel会结合两个字段，使之更容易被View使用，或者它可以计算可以输入的剩余字段长度。ViewModel可以实现数据验证逻辑确保数据正确性。

ViewModel可以定义逻辑状态，用于提供UI的可视化变更。View可以定义布局或者样式来反应ViewModel中变更的逻辑状态。比如，ViewModel可以定义一个状态，用于反映数据异步提交到Web服务。View可以在该状态时显示一个动画，以向用户提供可视化反馈。

通常情况下，ViewModel可以定义在UI上使用或者用户可以调用的命令或者操作。一个命令的例子是，ViewModel提供了**Submit**命令，允许用户提交数据到Web服务或者数据仓库。View可以选择用一个按钮来表示该命令，用户可以点击该按钮提交数据。通常，当命令变为不可用时，相关的UI表现为禁用。命令提供了一种方法封装用户操作以及彻底的把他们从相关的UI可视化中分离出来。

简而言之，ViewModel有以下主要特点：

* ViewModel是一个非可视类，且并非WPF或者Silverlight任何基类。它封装了应用中用于支持的显示逻辑。ViewModel可以独立于View和Model进行测试。
* ViewModel通常不直接引用View。它实现属性和命令，用于View的数据绑定。它通过通知事件通知View进行状态改变，通知事件是通过**INotifyPropertyChanged**和**INotifyCollectionChanged**接口实现。
* ViewModel协调View和Model之间的交互。它可以转换和操作数据，以便View更容易的使用，并且可以实现Model中不能实现的附加属性，也可以通过**IDataErrorInfo**或**INotifyDataErrorInfo**接口实现数据验证。
* ViewModel可以定义View用于反馈给用户的可视化信息所对应的逻辑状态。

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| **View还是View Model?**  很多时候，确定在何处实现何种功能，并不是很明显。一般的经验法则是：任何屏幕上的可视化元素，以及在稍后会被重设样式的(即使现在没有计划进行重设样式)都应该放到View中；任何关于应用的逻辑行为都应该放到ViewModel中。此外，由于ViewModel不需要知道View中特定的视觉元素的信息，以编码方式控制View中的视觉元素的代码应当放在View的后台代码中，或者封装在行为中。同样，检索或操作那些通过数据绑定显示在View中的数据项的代码，应当放在ViewModel中。  比如，列表框中被选中的项的高亮颜色，应当定义在View中，但是列表中的显示的项目，及被选中项所指向的数据，应当定义在ViewModel中。 |

## Model类

MVVM模式中的Model封装了业务逻辑以及数据。业务逻辑被定义为应用的逻辑，有关检索或者管理应用的数据，确保任何业务规则和确保数据的一致性和有效性的逻辑。为了最大话重用，模型不应当包含任何使用例相关或者用户事件相关的行为或者应用逻辑。

通常，Model代表应用的客户端域模型。它可以定义数据结构基于应用的数据模型和所有支持的业务和验证逻辑。Model可以包括数据访问和缓存的代码，但通常是一个独立的数据仓库或者服务被用来提供这些。通常，Model和数据访问层被作为数据访问或者服务策略的一部分，比如ADO.NET Entity Framework, WCF Data Services,或者WCF RIA Services。

通常，Model的实现让你轻松将其绑定到视图。意味着它支持属性和集合变更通知，通过实现接口**INotifyPropertyChanged**和**INotifyCollectionChanged**。Model类代表对象类集合，通常从从**ObservableCollection<T>**类派生，它提供了对**INotifyCollectionChanged**接口的实现。

Model同时支持数据验证和错误报告，通过实现**IDataErrorInfo**(或**INotifyDataErrorInfo**)接口。这些接口允许WPF和Silverlight数据绑定值放声变化时通知UI发生变化。他们还支持在UI层中的数据验证及错误报告。

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| **如果你的Model类不实现所需要的接口的话？**  有时候，你需要Model对象不实现**INotifyPropertyChanged**，**INotifyCollectionChanged**，**IDataErrorInfo**，或**INotifyDataErrorInfo**接口。在这些情况下，ViewModel就需要封装Model对象和暴露需要的属性给View。这些属性直接由Model对象提供。ViewModel需要为这些属性实现所需的接口，以便View可以方便的数据绑定它们。 |

简而言之，Model有以下主要特点:

* Model类是非可视类，封装了应用的数据和业务逻辑。它们负责管理应用数据和确保数据的一致性及有效性，通过封装所需的业务规则和验证逻辑。
* Model类不直接引用View或者ViewModel类，并不依赖它们的实现。
* Model类通常提供属性和集合的变更通知事件，通过实现**INotifyPropertyChanged**和**INotifyCollectionChanged**接口。这允许它们可以容易的被View数据绑定使用。Model类是对象的集合，通常从**ObservableCollection<T>**类派生。
* Model类通常提供数据验证和错误报告，通过实现**IDataErrorInfo**或**INotifyDataErrorInfo**接口。
* Model通常使用服务或者库封装数据访问和缓存。

# 类的交互

MVVM模式使得应用的UI，显示逻辑，以及业务逻辑和数据被清楚的划分成单独的类。因此，当你使用MVVM是，非常重要的因素是将你的应用的代码放到正确的类中，如上一节所述。

设计良好的View，ViewModel，和Model类不但封装了正确的类型和行为；同时被设计成可以很容易的通过数据绑定，命令，数据验证等接口，进行交互。

View和它的ViewModel间的交互也许是最重要的，但是Model和ViewModel之间的交互同样重要。以下各章节描述了各种相互交互的场景，并演示了在应用中是如何实现MVVM模式的。

## 数据绑定

数据绑定在MVVM模式中扮演了一个非常重要的角色。WPF和Silverlight都提供了强大的数据绑定功能。你的ViewModel和 (理想情况下)你的Model类被设计成支持数据绑定，因此他们可以充分利用这些功能。通常情况下，这意味着他们必须实现正确的接口。

Silverlight和WPF数据绑定支持多中模式，单向数据绑定，UI可以绑定到ViewModel中那些希望在显示时候进行渲染的底层数据。双向数据绑定可以在用户更新UI时，自动更新底层数据。

为确保UI在ViewModel的数据变更时，能保持最新状态，它需要实现适当的变更通知接口。如果它定义了可以被数据绑定的属性，那它应该实现**INotifyPropertyChanged**接口。如果ViewModel是一个集合，它应该实现**INotifyCollectionChanged**接口，或者从**ObservableCollection<T>** 类派生。所有这些接口都定义了一个在后台数据发生变化时候触发的事件。所有绑定了数据的控件会在该事件触发的时候进行更新。

许多情况下，ViewModel会定义具有返回值的属性(以及，有可能定义返回其他类型的对象)。WPF和Silverlight数据绑定支持通过使用**Path**属性，绑定到嵌套属性。因此，View的ViewModel返回其他ViewModel或者Model类的引用是很常见的。所有View可以访问的的ViewModel和Model类都应当实现**INotifyPropertyChanged**或**INotifyCollectionChanged**接口，如果需要。

以下各节描述了如何实现所需的接口，以便在MVVM模式中支持数据绑定。

### 实现INotifyPropertyChanged

在ViewModel或者Model中实现**INotifyPropertyChanged**接口，使它们能够在底层数据发生变化时，为绑定他们的控件提供变更通知。实现该接口非常简单，如下所示 (请查阅MVVM基础快速入门中的**Questionnaire**类)。

**C#**

public class Questionnaire : INotifyPropertyChanged

{

private string favoriteColor;

public event PropertyChangedEventHandler PropertyChanged;

...

public string FavoriteColor

{

get { return this.favoriteColor; }

set

{

if (value != this.favoriteColor)

{

this.favoriteColor = value;

if (this.PropertyChanged != null)

{

this.PropertyChanged(this,

new PropertyChangedEventArgs("FavoriteColor"));

}

}

}

}

}

在多个ViewModel类中实现**INotifyPropertyChanged**接口容器出错，且复用性不好，因为需要再事件的参数中指定属性的名字。Prism库提供了一个方便的基类，你可以从该类以类型安全方式，实现**INotifyPropertyChanged** 派生出你的ViewModel类，如下所示。

**C#**

public class NotificationObject : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

...

protected void RaisePropertyChanged<T>(

Expression<Func<T>> propertyExpression )

{...}

protected virtual void RaisePropertyChanged( string propertyName )

{...}

}

派生的ViewModel类可以出发属性变更时间，通过指定属性名，调用**RaisePropertyChanged**，或者使用lambda表达式，如下所示。

**C#**

public string CurrentState

{

get { return this.currentState; }

set

{

if ( this.currentState != value )

{

this.currentState = value;

this.RaisePropertyChanged( () => this.CurrentState );

}

}

}

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| 使用lambda表达式会有很小的额外性能开销，因为每次调用lambda表达式的时候都需要进行验证。好处是，这种方式提供了编译时类型安全，在你对属性进行重命名时也不会有影响。虽然性能开销很小，一般不会影响你的应用，但是如果有很多变更通知的情况下，该开销是可积累的。这种情况下，你应该考虑使用非lambda方式进行重载。 |

通常，你的Model或者ViewModel会使用来自其他的Model或者ViewModel的属性。当处理属性的变更通知时，一定还会引发其他属性的变更通知。

### 实现INotifyCollectionChanged

你的ViewModel或者Model类可能代表了一个集合，或者它可以定义返回集合的一个或多个属性， 这种情况下，你也许希望现实把该集合显示在View的一个**ItemsControl**中，比如**ListBox**，或者**DataGrid**中。这些控件可以绑定到这些代表一个集合或者属性是集合的ViewModel中，通过**ItemSource**属性。

**XAML**

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

为使属性支持变更通知请求，ViewModel或者Model类，如果它封装的是集合，需要实现**INotifyCollectionChanged**接口(除了**INotifyPropertyChanged**接口)。如果ViewModel或者Model类定义了一个属性返回的是集合的引用，集合类需要实现**INotifyCollectionChanged**接口。

然而，实现**INotifyCollectionChanged**接口是个很大的挑战，当集合中项目被增删改的时候。为了避免直接实现这个接口，通常从已经实现了该接口的集合类派生新类。**ObservableCollection<T>**类提供了该接口的实现，并且通常作为基类或者返回集合的属性。

如果你提供了一个集合给View做数据绑定，并且你不需要跟踪用户的选择，或者支持过滤，排序，或者分组集合中的项目，你可以简单的定义ViewModel中的属性，返回一个引用**ObservableCollection<T>**的实例。

**C#**

public class OrderViewModel : INotifyPropertyChanged

{

public OrderViewModel( IOrderService orderService )

{

this.LineItems = new ObservableCollection<OrderLineItem>(

orderService.GetLineItemList() );

}

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

}

如果你得到一个集合类的引用 (比如，从另一个没有实现**INotifyCollectionChanged**的组件或服务)，你可能需要封装集合到**ObservableCollection<T>**实例，使用包含**IEnumerable<T>**或**List<T>**参数的构造函数。

### 实现ICollectionView

上述代码演示了如何实现一个简单的ViewModel属性，它返回一个集合，并且该集合可以通过数据绑定显示在控件中。由于**ObservableCollection<T>**类实现了**INotifyCollectionChanged**接口，View中的控件将会自动更新，以反应当前列表中的项目被追加或者被删除。

然而，你通常需要更好的从ViewModel自身，控制在View中如何更好的显示项目的集合，或者跟踪用户和显示项目之间的交换。比如，你可能需要允许集合过滤或者排序，通过在ViewModel中实现的显示逻辑，或者你需要保持跟踪View中的当前选中项，以便在ViewModel中实现的命令可以作用于当前选定的项目。

WPF和Silverlight支持这些场景，通过各种实现了**ICollectionView**接口的类。该接口提供属性和方法允许一个集合被过滤，排序和分组，并且允许跟踪和改变当前选中项。Silverlight和WPF都提供了该接口的实现——Silverlight提供了**PagedCollectionView**类， WPF提供了**ListCollectionView**类。

集合视图类的底层封装了集合，因此可以提供自动的选择跟踪和排序，过滤，分页。该类的实例可以通过编码方式或者在XAML中定义来实现，通过使用**CollectionViewSource**类。

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| 在WPF中，不管是否绑定集合，集合视图类都会被默认的创建。在Silverlight中，只有在集合支持**ICollectionViewFactory**接口的时候，才会自动创建。 |

集合视图类可以用于ViewModel跟踪底层集合的重要状态信息，而在View的UI层和Model的数据层间保持一个干净的分离。实际上， **CollectionViews**是专门设计用于支持集合的。

因此，如果你在ViewModel中需要实现过滤，排序，分组或者跟踪选择项，你的ViewModel需要为暴露给View的集合创建实例。你可以订阅选择变更事件，比如**CurrentChanged**事件，或者控件过滤，排序，或者分组，通过在你的ViewModel中的集合视图类提供的方法。

ViewModel应当实现一个只读属性，返回一个**ICollectionView**的引用，以便在View中使用数据绑定或者并它交互。WPF和Silverlight中，从**ItemsControl**基类派生出的类可以自动和实现**ICollectionView**的类进行交互。

下面代码演示了如何在Silverlight中使用**PagedCollectionView**跟踪当前选中用户。.

**C#**

public class MyViewModel : INotifyPropertyChanged

{

public ICollectionView Customers { get; private set; }

public MyViewModel( ObservableCollection<Customer> customers )

{

// Initialize the CollectionView for the underlying model

// and track the current selection.

Customers = new PagedCollectionView( customers );

Customers.CurrentChanged +=

new EventHandler( SelectedItemChanged );

}

private void SelectedItemChanged( object sender, EventArgs e )

{

Customer current = Customers.CurrentItem as Customer;

...

}

}

在View，你可以绑定一个**ItemsControl**，比如**ListBox**，到ViewModel的**Customers**属性，通过它的**ItemsSource**属性，如下所示。

**XAML**

<ListBox ItemsSource="{Binding Path=Customers}">

<ListBox.ItemTemplate>

<DataTemplate>

<StackPanel>

<TextBlock Text="{Binding Path=Name}"/>

</StackPanel>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

当用户在UI选择一个用户的时候，ViewModel会被通知，以便它可以应用命令到当前所选择的用户。ViewModel也可以通过编码方式更改当前选择的用户界面，通过调用集合视图类的方法。如下代码所示。

**C#**

Customers.MoveCurrentToNext();

当在集合视图中的选择被改变时，UI自动更新被选中项目的视觉状态。在WPF中实现非常简单，前面所说的**PagedCollectionView**通常用**ListCollectionView**或者**BindingListCollectionView**类代替，如下所示。

**C#**

Customers = new ListCollectionView( \_model );

Customers.CurrentChanged += new EventHandler( SelectedItemChanged );

## 命令

除了对显示在View中的数据的提供访问和编辑权限之外，ViewModel还可能定义一个或多个动作或者操作提供给用户。在WPF和Silverlight中，动作和操作，用户可以通过UI来使用的，通常定义为命令。命令提供一个方便的方式来表示动作或操作，可以轻松的绑定到UI的控件。他们封装了动作和操作的实际代码实现，并且有利于和View的视觉元素保持分离。

命令可以通过可视化表现形式，被用户通过不同的方式调用。在许多情况下，他们被作为一个动作结果调用，如鼠标点击，键盘事件，触摸手势，或者其他的输入事件。在View中，控件被数据绑定到ViewModel的命令，因此用户可以通过任何控件定义的动作使用他们。View中的UI控件和命令之间的交互可以是双向的。这种情况下，命令可以命令可以作为UI的用户交互服务，UI也可以通过命令的可用不可用切换，自动显示当前状态。

ViewModel可以实现命令通过**Command Method**或者作为**Command Object** (一个实现了**ICommand**接口的对象)。这两种情况下，View和命令的交互，可以不需要在View的后台代码中使用复杂的事件处理代码。比如，WPF和Silverlight中某些控件本身支持命令，提供了一个**Command**属性，可以绑定到ViewModel提供的实现了**ICommand**的对象上。其他情况下，可以使用命令行为来处理命令方法或者是ViewModel提供的命令对象。

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| Behaviors are a powerful and flexible extensibility mechanism that can be used to encapsulate interaction logic and behavior that can then be declaratively associated with controls in the view. Command behaviors can be used to associate command objects or methods with controls that were not specifically designed to interact with commands. |

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

### 实现命令对象

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

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

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

**C# DelegateCommand.cs**

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

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

**C#**

public class QuestionnaireViewModel { public QuestionnaireViewModel() { this.SubmitCommand = new DelegateCommand<object>( this.OnSubmit, this.CanSubmit ); } public ICommand SubmitCommand { get; private set; } private void OnSubmit(object arg) {...} private bool CanSubmit(object arg) { return true; } }

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

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

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

Other implementations of the **ICommand** interface are available. The **ActionCommand** class provided by the Expression Blend SDK is similar to Prism's **DelegateCommand** class described earlier, but it supports only a single **Execute** method delegate. Prism also provides the **CompositeCommand** class, which allows **DelegateCommands** to be grouped together for execution. For 更多 about using the **CompositeCommand** class, see "[Composite Commands](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#CompositeCommands)" in Chapter 6, "[Advanced MVVM Scenarios](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))."

### 从View中调用命令对象

There are a number of ways in which a control in the view can be associated with a command object proffered by the view model. Certain WPF and Silverlight 4 controls, notably **ButtonBase** derived controls, such as **Button** or **RadioButton**, and **Hyperlink**, or**MenuItem**derived controls, can be easily data bound to a command object through the **Command** property. WPF also supports binding view model**ICommand** to a **KeyGesture**.

**XAML**

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

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

An alternative approach is to use Expression Blend interaction triggers and **InvokeCommandAction** behavior.

**XAML**

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

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

Unlike controls that can be bound directly to a command, **InvokeCommandAction** does not automatically enable or disable the control based on the command's **CanExecute** value. To implement this behavior, you have to data bind the **IsEnabled** property of the control directly to a suitable property on the view model, as shown earlier.

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| **Command-Enabled Controls vs. Behaviors** WPF and Silverlight 4 controls that support commands allow you to declaratively hook up a control to a command. These controls will invoke the specified command when the user interacts with the control in a specific way. For example, for a **Button** control, the command will be invoked when the user clicks the button. This event associated with the command is fixed and cannot be changed.  Behaviors also allow you to hook up a control to a command in a declarative fashion. However, behaviors can be associated with a range of events raised by the control, and they can be used to conditionally invoke an associated command object or a command method on the view model. In other words, behaviors can address many of the same scenarios as command-enabled controls, and they may provide a greater degree of flexibility and control. You will need to choose when to use command-enabled controls and when to use behaviors, as well as which kind of behavior to use. If you prefer to use a single mechanism to associate controls in the view with functionality in the view model or for consistency, you should consider using behaviors, even for controls that inherently support commands. If you only need to use command-enabled controls to invoke commands on the view model, and if you are happy with the default events to invoke the command, behaviors may not be required. Similarly, if your developers or UI designers will not be using Expression Blend, you may favor command-enabled controls (or custom attached behaviors) because of the additional syntax required for Expression Blend behaviors. |

### 从View中调用命令方法

An alternative approach to implementing commands as **ICommand** objects is to implement them simply as methods in the view model and then to use behaviors to invoke those methods directly from the view.

This can be achieved in a similar way to the invocation of commands from behaviors, as shown in the previous section. However, instead of using **InvokeCommandAction**, you use the **CallMethodAction**. The following code example calls the (parameter-less) **Submit** method on the underlying view model.

**XAML**

<Button Content="Submit" IsEnabled="{Binding CanSubmit}"> <i:Interaction.Triggers> <i:EventTrigger EventName="Click"> <i:CallMethodAction TargetObject="{Binding}" Method="Submit"/> </i:EventTrigger> </i:Interaction.Triggers> </Button>

The **TargetObject** is bound to the underlying data context (which is the view model) by using the **{Binding}** expression. The **Method**parameter specifies the method to invoke.

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| **CallMethodAction** does not support parameters; if you need to pass parameters to the target method, you have to provide the values as properties on the view model, switch to using a command with an **InvokeCommandAction**, or write your own version of the**CallMethodAction** that will pass parameters. |

## 数据验证和异常汇报

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

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

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

### 实现IDataErrorInfo

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

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

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

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

**XAML**

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

### 实现INotifyDataErrorInfo

The **INotifyDataErrorInfo** interface is more flexiblethan the**IDataErrorInfo** interface. It supports multiple errors for a property, asynchronous data validation, and the ability to notify the view if the error state changes for an object. However, **INotifyDataErrorInfo** is currently only supported in Silverlight 4 and is not available in WPF 4.

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

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

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

**C#**

public abstract class DomainObject : INotifyPropertyChanged, INotifyDataErrorInfo { private ErrorsContainer<ValidationResult> errorsContainer = new ErrorsContainer<ValidationResult>( pn => this.RaiseErrorsChanged( pn ) ); public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged; public bool HasErrors { get { return this.ErrorsContainer.HasErrors; } } public IEnumerable GetErrors( string propertyName ) { return this.errorsContainer.GetErrors( propertyName ); } protected void RaiseErrorsChanged( string propertyName ) { var handler = this.ErrorsChanged; if (handler != null) { handler(this, new DataErrorsChangedEventArgs(propertyName) ); } } ... }

In Silverlight, any controls data bound to properties on the view model will automatically subscribe to the **INotifyDataErrorInfo** event and display error information on the control if the property contains an error.

# 组装

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

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| Choosing an appropriate strategy to manage this step is especially important if you are using a dependency injection container in your application. The Managed Extensibility Framework (MEF) and the Unity Application Block (Unity) both provide the ability to specify dependencies between the view, view model, and model classes and to have them fulfilled by the container. For more advanced scenarios, see Chapter 6, "[MVVM进阶模式](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))." |

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

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

|  |
| --- |
| **g405484.note(en-us,PandP.40).gif注意:** |
| However, it should be 注意:d that the view will implicitly depend on specific properties, commands, and methods on the view model because of the data bindings it defines. If the view model does not implement the required property, command, or method, a run-time exception will be generated by the data binding engine, which will be displayed in the Visual Studio output window during debugging. |

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

## 通过XAML创建View Model

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

The XAML-based approach is demonstrated in the QuestionnaireView.xaml file in the Basic MVVM QuickStart. In that example, the**QuestionnaireViewModel** instance is defined in the **QuestionnaireView**'s XAML, as shown here.

**XAML**

<UserControl.DataContext> <my:QuestionnaireViewModel/> </UserControl.DataContext>

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

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

## 通过代码创建View Model

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

**C#**

public QuestionnaireView() { InitializeComponent(); this.DataContext = new QuestionnaireViewModel(); }

The programmatic construction and assignment of the view model within the view's code-behind has the advantage that it is simple and works well in design-time tools like Expression Blend or Visual Studio. The disadvantage of this approach is that the view needs to have knowledge of the corresponding view model type and that it requires code in the view's code-behind. Using a dependency injection container, such as Unity or MEF, can help to maintain loose coupling between the view and view model. For 更多, see Chapter 3, "[Managing Dependencies Between Components](http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.40))."

## 创建View并定义为数据模版

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

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

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

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

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

**XAML**

<UserControl ...> <UserControl.Resources> <DataTemplate x:Key="CustomerViewTemplate"> <local:CustomerContactView /> </DataTemplate> </UserControl.Resources> <Grid> <ContentControl Content="{Binding Customer}" ContentTemplate="{StaticResource CustomerViewTemplate}" /> </Grid> </UserControl>

Here, the data template wraps a concrete view type. This allows the view to define code-behind behavior. In this way, the data template mechanism can be used to externally provide the association between the view and the view model. Although the preceding example shows the template in the **UserControl** resources, it would often be placed in application's resources for reuse. You can find an example of using data templates to instantiate views and associate them with their view models in the MVVM QuickStart file QuestionnaireView.xaml.

# 关键决策

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

* Decide on the approach to view and view model construction you will use. You need to decide if your application constructs the views or the view models first and whether to use a dependency injection container, such as Unity or MEF. You will usually want this to be consistent application-wide. For 更多, see the section, "[组装](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#ConstructionandWireUp)," in this chapter and the section "Advanced Construction and Wire-Up," in Chapter 6, "[MVVM模式进阶](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))."
* Decide if you will expose commands from your view models as command methods or command objects. Command methods are simple to expose and can be invoked through behaviors in the view. Command objects can neatly encapsulate the command and enabled/disabled logic and can be invoked through behaviors or via the **Command** property on **ButtonBase**-derived controls. To make it easier on your developers and designers, it is a good idea to make this an application-wide choice. For 更多, see the section, "[命令](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#Commands)," in this chapter.
* Decide how your view models and models will report errors to the view. Your models can either support **IDataErrorInfo** or, if using Silverlight, **INotifyDataErrorInfo**. Not all models may need to report error information, but for those that do, it is preferable to have a consistent approach for your developers. For 更多, see the section, "[Data Validation and Error Reporting](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#DataValidationandErrorReporting)," in this chapter.
* Decide whether Microsoft Expression Blend design-time data support is important to your team. If you will use Expression Blend to design and maintain your UI and want to see design time data, make sure that your views and view models offer constructors that do not have parameters and that your views provide a design-time data context. Alternatively, consider using the design-time features provided by Microsoft Expression Blend using design-time attributes such as **d:DataContext** and **d:DesignSource**. For 更多, see "Guidelines for Creating Designer Friendly Views" in Chapter 7, "[Composing the User Interface](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))."

# 更多

更多关于WPF中的数据绑定的详细信息，请查阅MSDN上的"数据绑定"：<http://msdn.microsoft.com/en-us/library/ms750612.aspx>.

更多关于Silverlight的数据绑定的详细信息，请查阅MSDN上的"数据绑定"：<http://msdn.microsoft.com/en-us/library/cc278072(VS.95).aspx>.

更多关于WPF集合绑定的详细信息，请查阅MSDN上"数据绑定概论"的"集合绑定"：<http://msdn.microsoft.com/en-us/library/ms752347.aspx>.

更多关于Silverlight集合绑定的详细信息，请查阅MSDN上"数据绑定"的"集合绑定"：<http://msdn.microsoft.com/en-us/library/cc278072(VS.95).aspx>.

更多关于Presentation Model模式的详细信息，请查阅Martin Fowler 网站的"Presentation Model"：<http://www.martinfowler.com/eaaDev/PresentationModel.html>.

更多关于数据模板的详细信息，请查阅MSDN上"数据模板概述"：<http://msdn.microsoft.com/en-us/library/ms742521.aspx>.

更多关于MEF的详细信息，请查阅MSDN上的 "MEF概述"：<http://msdn.microsoft.com/en-us/library/dd460648.aspx>.

更多关于Unity的详细信息，请查阅MSDN上的"Unity Application Block"：<http://www.msdn.com/unity>.

更多关于**DelegateCommand**和**CompositeCommand**的详细信息，请查阅Chapter 9，"[松耦合组件之间的通讯](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40))."

# 6: MVVM模式进阶

|  |
| --- |
| **索引:** |
| [**命令**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec1) | [**复合命令**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec2) – [*注册和注销子命令*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec3), [*在活动子Views*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec4)*上执行命令* |[**Collections**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec5)**中的命令** | [**命令行为**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec6) – [*扩展Prism命令行为*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec7) | [**处理异步交互**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec8) | [**检索数据并和Web服务交互**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec9) | [**交互模式**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec10) | [**使用交互服务**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec11) |[**使用交互请求对象**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec12) – [*处理从View Model*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec13)*发来的交互请求*, [*使用行为实现交互体验*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec14) | [**组装进阶**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec15) - [*通过MEF*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec16)*创建View和View Model*, [*通过Unity*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec17)*创建View和View Model* | [**通过外部类创建View和View Model**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec18) | [**测试MVVM应用**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec19) | [**测试INotifyPropertyChanged实现**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec20) – [*简单场景*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec21), [*依赖运行环境并且没有Set的属性*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec22), [*对象整体的通知*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec23) |[**测试INotifyDataErrorInfo实现**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec24) – [*测试验证规则*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec25) , [*测试INotifyDataErrorInfo实现的请求*](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec26) | [**测试异步服务调用**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec27) | [**更多**](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#sec28) |

The previous chapter described how to implement the basic elements of the Model-View-ViewModel (MVVM) pattern by separating your application's user interface (UI), presentation logic, and business logic into three separate classes (the view, view model, and model), implementing the interactions between those classes (through data binding, commands, and data validation interfaces), and by implementing a strategy to handle construction and wire-up.

Implementing the MVVM pattern using these basic elements will likely support many of the scenarios in your application. However, you may encounter more sophisticated scenarios that require the basic MVVM pattern to be extended or that require more advanced techniques to be applied. This is more likely to be true if your application is large or complex, but you may also encounter these scenarios in many smaller applications. The Prism Library provides components that implement many of these techniques, allowing you to more easily use them in your own applications.

This chapter describes some sophisticated scenarios and describes how the MVVM pattern can support them. The next section describes how commands can be chained together or associated with child views and how they can be extended to support custom requirements. The following sections then describe how to handle asynchronous data requests and subsequent UI interactions and how to handle interaction requests between the view and the view model.

The section, "[Advanced Construction and Wire-Up](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40)#AdvancedConstructionandWireUp)," provides guidance on handling construction and wire-up when using a dependency injection container, such as the Unity Application Block (Unity), or when using the Managed Extensibility Framework (MEF). The final section describes how you can test MVVM applications by providing guidance on unit testing your application's view model and model classes, and on testing behaviors.

# 命令

Commands provide a way to separate the command's implementation logic from its UI representation. Data binding or behaviors provide a way to declaratively associate elements in the view with commands proffered by the view model. The section, "[Commands](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#Commands)" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))," described how commands can be implemented as command objects or command methods on the view model, and how they can be invoked from controls in the view either by using behaviors or by using the built-in **Command** property provided by certain controls.

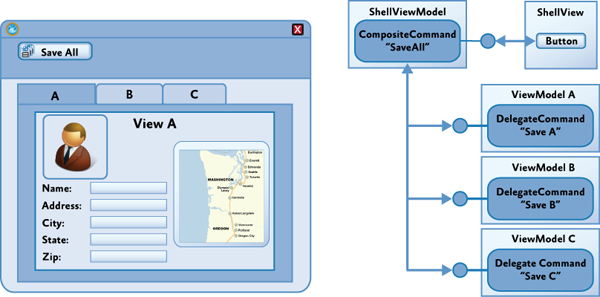
|  |
| --- |
| **g405494.note(en-us,PandP.40).gifWPF路由命令:** |
| It should be 注意:d that commands implemented as command objects or command methods in the MVVM pattern differ somewhat from WPF's built-in implementation of commands named routed commands (Silverlight does not have any routed command implementations). WPF routed commands deliver command messages by routing them through elements in the UI tree (specifically the [logical tree](http://msdn.microsoft.com/en-us/library/ms753391.aspx)). Therefore, command messages are routed up or down the UI tree from the focused element or to an explicitly specified target element; by default, they are not routed to components outside of the UI tree, such as the view model associated with the view. However, WPF-routed commands can use a command handler defined in the view's code-behind to forward the command call to the view model class. |

## 复合命令

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

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

**Implementing the SaveAll composite command**



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

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

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

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

### Registering and Unregistering Child Commands

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

**C# OrdersController.cs**

commandProxy.SubmitAllOrdersCommand.RegisterCommand( orderCompositeViewModel.SubmitCommand ); commandProxy.CancelAllOrdersCommand.RegisterCommand( orderCompositeViewModel.CancelCommand );

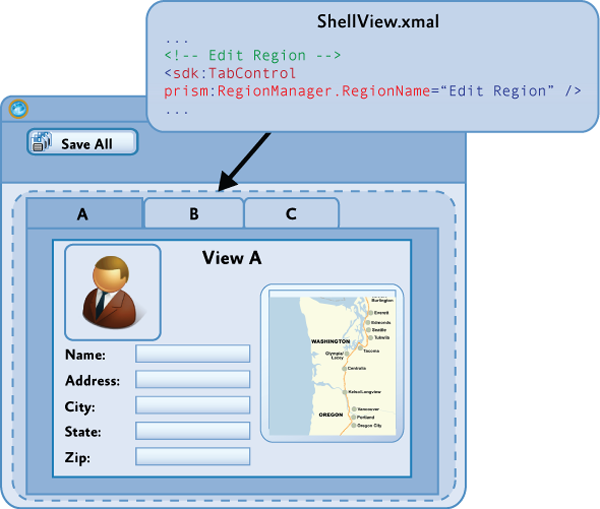
|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| The preceding **commandProxy** object provides instance access to the **Submit** and **Cancel** composite commands, which are defined statically. For 更多, see the class file StockTraderRICommands.cs. |

### Executing Commands on Active Child Views

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

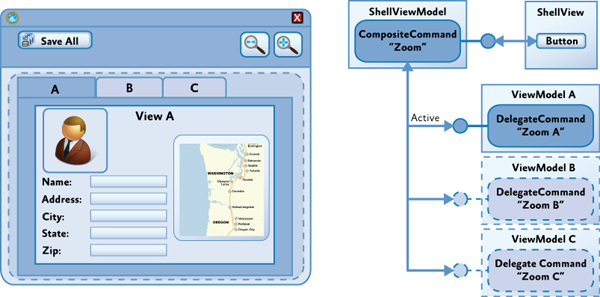
Prism regions (described in section, "Regions," in Chapter 7, "Composing the User Interface") provide a way for child views to be associated with logical placeholders in the application's UI. They are often used to decouple the specific layout of child views from their logical placeholder and its position in the UI. Regions are based on named placeholders that are attached to specific layout controls. The following illustration shows an example where each child view has been added to the region named **EditRegion**, and the UI designer has chosen to use a **Tab** control to lay out the views within that region.

**Defining the EditRegion using a Tab control**



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

**Defining the EditRegion using a Tab control**



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

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

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

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

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

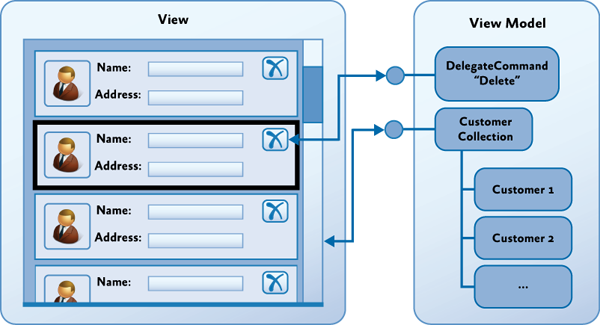
You can use this functionality to implement the example described earlier. By implementing the **IActiveAware** interface on your child view models, you will be notified when your child view becomes active or inactive with the region. When the child view's active status changes, you can update the active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

## Commands Within Collections

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

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

**Binding commands within collections**



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

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

**XAML**

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

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

## Command Behaviors

In Silverlight 3 and earlier versions, Silverlight did not offer controls that directly supported commands. The **ICommand** interface was available, but no controls implemented the **Command** property to allow them to be directly hooked up to an **ICommand** implementation. To overcome this limitation, and to support MVVM commanding patterns in Silverlight 3, the Prism Library (version 2.0) provided a mechanism to allow any Silverlight control to be bound to a command object using an attached behavior. This mechanism also worked in WPF, which allowed view model implementations to be re-used in both Silverlight and WPF applications.

The following example shows how the Prism command behaviors are used to bind a command object defined on a view model to a button's click event.

**XAML OrdersView.xaml**

<Button Content="Submit All" prism:Click.Command="{Binding Path=SubmitAllCommand}" prism:Click.CommandParameter="{Binding Path=TickerSymbol}" />

Silverlight 4 added support for the **Command** property to all **Hyperlink**-derived and **ButtonBase**-derived controls, allowing them to be bound directly to command objects in the same way as in WPF. The use of the **Command** property for these controls is described in the section, "[Commands](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#Commands)," in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))." However, Prism command behaviors remain in the Prism Library for backward compatibility reasons and to support the development of custom behaviors, as described later.

The behavior approach is a generally applicable technique for implementing and encapsulating interactive behavior in a way that can be easily applied to controls in the view. The use of behaviors to support commands as shown earlier is just one of the many scenarios that behaviors can support. Microsoft Expression Blend now provides a variety of behaviors, including the **InvokeCommandAction** and the**CallMethodAction** described in the section, "Invoking Command Methods from the View," in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))," and a software development kit (SDK) to allow the development of custom behaviors. Expression Blend provides drag-and-drop creation and property editing support for behaviors, which makes the task of adding behaviors very easy. For 更多 about developing custom Expression Blend behaviors, see "[Creating Custom Behaviors](http://msdn.microsoft.com/en-us/library/ff724708(v=Expression.40).aspx)" on MSDN.

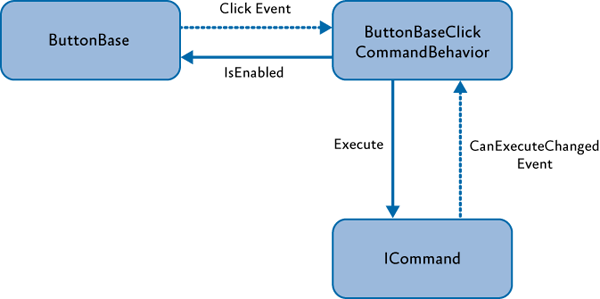
Although the introduction of support for command-enabled controls in Silverlight 4, and the introduction of the Expression Blend Behaviors SDK, obviates much of the need for the Prism command behaviors, you may find their compact syntax and implementation, and their ability to be easily extended, to be useful.

### Extending Prism Command Behaviors

Prism command behaviors are based on an attached behavior pattern. This pattern connects events raised by controls to the command objects provided by the view model. The Prism command behavior is comprised of two parts: an attached property and a behavior object. The attached property establishes a relationship between the target control and the behavior object. The behavior object monitors the target control and takes action based on events or state changes in the control or the view model.

The Prism command executes commands based on the **Click** event of **ButtonBase**-derived controls by providing the**ButtonBaseClickCommandBehavior** class and an attached property to attach it to the click event of a target control. The following illustration shows the relationship between **ButtonBase**, **ButtonBaseClickCommandBehavior**, and the **ICommand** object provided by the view model.

**Forwarding a ButtonClick event to an ICommand**



Your application may need to invoke commands from controls or events other than the **Click** event from **ButtonBase**, or you may need to customize the way in which the behavior interacts with the target control or the view model to which it is bound. In these cases, you will need to define your own attached property and/or behavior implementation.

The Prism Library provides the **CommandBehaviorBase<T>** class to make it easier to create behaviors that interact with **ICommand**objects. This class invokes the command and monitors changes in the command's **CanExecuteChanged** event, and it can be used to extend command support in both Silverlight and WPF.

To create a custom behavior, create a class that inherits from **CommandBehaviorBase<T>** and targets the control you want to monitor. The type parameter for this class specifies the type of the control to which the behavior can be attached. In the constructor of your class, you can subscribe to the events that you want to monitor from the control. The following code example shows the implementation of the**ButtonBaseClickCommandBehavior** class.

**C#**

public class ButtonBaseClickCommandBehavior : CommandBehaviorBase<ButtonBase> { public ButtonBaseClickCommandBehavior(ButtonBase clickableObject) : base(clickableObject) { clickableObject.Click += OnClick; } private void OnClick(object sender, System.Windows.RoutedEventArgs e) { ExecuteCommand(); } }

Using the **CommandBehaviorBase<T>** class, you can define custom behavior classes of your own; this allows you to customize how the behavior interacts with the target control or the command provided by the view model. For example, you could define a behavior that invokes the bound command based on a different control event or that changes the visual state of the control based on the **CanExecute**status of the bound command.

To support the declarative attachment of the command behavior to the target control, an attached property is used. The attached property allows the behavior to be attached to the control in XAML and manages the construction and association of the behavior implementation with the target control. The attached property is defined within a static class. Prism command behaviors are based on the convention that the name of the static class refers to the event that is used to invoke the command. The name of the attached property refers to the type of the object being data bound. Therefore, the Prism command behaviors described earlier use a static class named **Click**, which defines an attached property named **Command**. This allows the use of the **Click.Command** syntax shown earlier.

The command behavior object itself is actually also associated with the target control via an attached property. However, this attached property is private to the static class and is not visible to the developer.

**C#**

public static readonly DependencyProperty CommandProperty = DependencyProperty.RegisterAttached( "Command", typeof(ICommand), typeof(Click), new PropertyMetadata(OnSetCommandCallback)); private static readonly DependencyProperty ClickCommandBehaviorProperty = DependencyProperty.RegisterAttached( "ClickCommandBehavior", typeof(ButtonBaseClickCommandBehavior), typeof(Click), null);

The implementation of the **Command** attached property creates an instance of the **ButtonBaseClickCommandBehavior** class, via the**OnSetCommandCallback** callback method, as shown in the following code example.

**C#**

private static void OnSetCommandCallback(DependencyObject dependencyObject, DependencyPropertyChangedEventArgs e) { ButtonBase buttonBase = dependencyObject as ButtonBase; if (buttonBase != null) { ButtonBaseClickCommandBehavior behavior = GetOrCreateBehavior(buttonBase); behavior.Command = e.NewValue as ICommand; } } private static void OnSetCommandParameterCallback(DependencyObject dependencyObject, DependencyPropertyChangedEventArgs e) { ButtonBase buttonBase = dependencyObject as ButtonBase; if (buttonBase != null) { ButtonBaseClickCommandBehavior behavior = GetOrCreateBehavior(buttonBase); behavior.CommandParameter = e.NewValue; } } private static ButtonBaseClickCommandBehavior GetOrCreateBehavior( ButtonBase buttonBase ) { ButtonBaseClickCommandBehavior behavior = buttonBase.GetValue(ClickCommandBehaviorProperty) as ButtonBaseClickCommandBehavior; if ( behavior == null ) { behavior = new ButtonBaseClickCommandBehavior(buttonBase); buttonBase.SetValue(ClickCommandBehaviorProperty, behavior); } return behavior; }

For 更多 about attached properties, see [Attached Properties Overview](http://msdn.microsoft.com/en-us/library/cc265152(VS.95).aspx) on MSDN.

# Handling Asynchronous Interactions

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

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

## Retrieving Data and Interacting with Web Services

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

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

**C#**

IAsyncResult asyncResult = this.service.BeginGetQuestionnaire(GetQuestionnaireCompleted, null // object state, not used in this example); private void GetQuestionnaireCompleted(IAsyncResult result) { try { questionnaire = this.service.EndGetQuestionnaire(ar); } catch (Exception ex) { // Do something to report the error. } }

It is important to 注意: that in the calls to the **End** method (in this case, **EndGetQuestionnaire**), any exceptions that occurred during the execution of the request will be raised. Your application must handle these and may need to report them in a thread-safe way via the UI. If you do not handle these, the thread will end and you will not be able to process the results.

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

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

**C#**

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

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

**C#**

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

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

**C#**

this.questionnaireRepository.GetQuestionnaireAsync( (result) => { if (result.Error == null) { this.Questionnaire = result.Result; ... } else { // Handle error. } })

# User Interaction Patterns

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

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

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

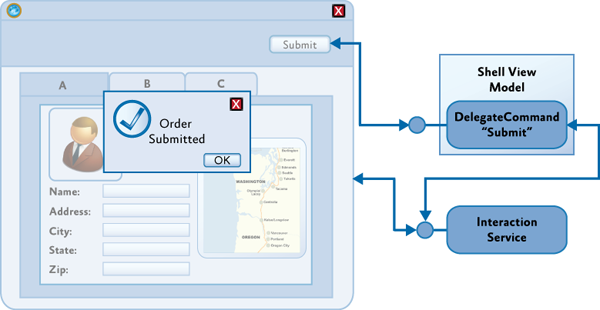
There are two common approaches to implementing these kinds of user interactions in the MVVM pattern. One approach is to implement a service that can be used by the view model to initiate interaction with the user, thereby preserving its independence on the view's implementation. Another approach uses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

## Using an Interaction Service

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

After the view model has a reference to the interaction service, it can programmatically request interaction with the user whenever necessary. The interaction service implements the visual aspects of the interaction, as shown in the following illustration. Using an interface reference in the view model allows for different implementations to be used, according to the implementation requirements of the user interface. For example, implementations of the interaction service for WPF and Silverlight could be provided, allowing for greater re-use of the application's presentation logic.

**Using an interaction service to interact with the user**



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

**C#**

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

However, one of the disadvantages of this approach is that it forces a synchronous programming model that is not shared by other interaction mechanisms in Silverlight resulting in numerous difficulties when implementing the interaction service. An alternative asynchronous implementation allows for the view model to provide a callback to execute on completion of the interaction. The following code illustrates this approach.

**C#**

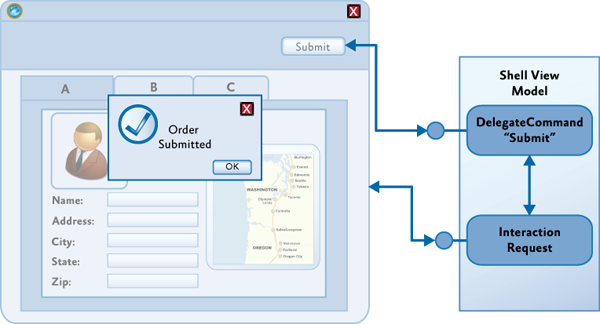
interactionService.ShowMessageBox( "Are you sure you want to cancel this operation?", "Confirm", MessageBoxButton.OK, result => { if (result == MessageBoxResult.Yes) { CancelRequest(); } });

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

## Using Interaction Request Objects

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

**Using an interaction request object to interact with the user**



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

This approach is consistent with the MVVM pattern, enabling the view to reflect state changes it observes on the view model and using two-way data binding for communication of data between the two. The encapsulation of the non-visual elements of the interaction in an interaction request object, and the use of a corresponding behavior to manage the visual elements of the interaction, are very similar to the way command objects and command behaviors are used.

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

### Initiating Interaction Requests from the View Model

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

**C#**

public interface IInteractionRequest { event EventHandler<InteractionRequestedEventArgs> Raised; } public class InteractionRequest<T> : IInteractionRequest { public event EventHandler<InteractionRequestedEventArgs> Raised; public void Raise(T context, Action<T> callback) { var handler = this.Raised; if (handler != null) { handler( this, new InteractionRequestedEventArgs( context, () => callback(context))); } } }

Prism provides pre-defined context classes that support common interaction request scenarios. The **Notification** class is the base class for all context objects. It is used when the interaction request is used to notify the user of an important event in the application. It provides two properties—**Title** and **Content**—which will be displayed to the user. Typically, notifications are one-way, so it is 注意:xpected that the user will change these values during the interaction.

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

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

**C#**

public IInteractionRequest ConfirmCancelInteractionRequest { get { return this.confirmCancelInteractionRequest; } } this.confirmCancelInteractionRequest.Raise( new Confirmation("Are you sure you wish to cancel?"), confirmation => { if (confirmation.Confirmed) { this.NavigateToQuestionnaireList(); } }); }

The MVVM Reference Implementation (MVVM RI) illustrates how the **IInteractionRequest** interface and the **InteractionRequest<T>** class are used to implement user interactions between the view and view model in a survey application (see QuestionnaireViewModel.cs).

### Using Behaviors to Implement the Interaction User Experience

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

The view must be set up to detect an interaction request event, and then to present the appropriate visual display for the request. The Microsoft Expression Blend Behaviors Framework supports the concept of triggers and actions. Triggers are used to initiate actions whenever a specific event is raised.

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

After the event is raised, the **InteractionRequestTrigger** will invoke the specified action. For Silverlight, the Prism Library provides the**PopupChildWindowAction** class, which displays a pop-up window to the user. When the child window is displayed, its data context is set to the context parameter of the interaction request. Using the **ContentTemplate** property of the **PopupChildWindowAction** class, you can specify a data template to define the UI layout to be used for the **Content** property of the context object. The title of the pop-up window is bound to the **Title** property of the context object.

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| By default, the specific type of pop-up window displayed by the **PopupChildWindowAction** class depends on the type of the context object. For a **Notification** context object, a **NotificationChildWindow** is displayed, while for a **Confirmation** context object, a**ConfirmationChildWindow** is displayed. The **NotificationChildWindow** displays a simple popup window to display the notification, while the **ConfirmationChildWindow** also contains **OK** and **Cancel** buttons to capture the user's response. You can override this behavior by specifying a pop-up window using the **ChildWindow** property of the **PopupChildWindowAction** class. |

The following example shows how the **InteractionRequestTrigger** and the **PopupChildWindowAction** are used to display a confirmation pop-up window to the user within the MVVM RI.

**XAML**

<i:Interaction.Triggers> <prism:InteractionRequestTrigger SourceObject="{Binding ConfirmCancelInteractionRequest}"> <prism:PopupChildWindowAction ContentTemplate="{StaticResource ConfirmWindowTemplate}"/> </prism:InteractionRequestTrigger> </i:Interaction.Triggers> <UserControl.Resources> <DataTemplate x:Key="ConfirmWindowTemplate"> <Grid MinWidth="250" MinHeight="100"> <TextBlock TextWrapping="Wrap" Grid.Row="0" Text="{Binding}"/> </Grid> </DataTemplate> </UserControl.Resources>

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| The data template specified using the **ContentTemplate** property defines the UI layout for the **Content** property of the context object. In the preceding code, the **Content** property is a string, so the **TextBlock** is simply bound to the **Content** property itself. |

As the user interacts with the pop-up window, the context object is updated according to the bindings defined in the pop-up window or the data template used to display the **Content** property of the context object. After the user closes the pop-up window, the context object is passed back to the view model, along with any updated values, via the callback method. In the confirmation example used in the MVVM RI, the default confirmation view is responsible for setting the **Confirmed** property on the supplied **Confirmation** object to **true** when the **OK**button is clicked.

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

# Advanced Construction and Wire-Up

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

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

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

### Creating the View and View Model Using MEF

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

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

**C#**

[Import] public QuestionnaireViewModel ViewModel { set { this.DataContext = value; } }

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

**C#**

[Export] public class QuestionnaireViewModel : NotificationObject { ... }

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

**C#**

public QuestionnaireView() { InitializeComponent(); } [ImportingConstructor] public QuestionnaireView(QuestionnaireViewModel viewModel) : this() { this.DataContext = viewModel; }

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

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| You can use property injection or constructor injection in both MEF and Unity; however, you may find property injection to be simpler because you do not have to maintain two constructors. Design-time tools, such as Visual Studio and Expression Blend, require that controls have a default parameter-less constructor in order to display them in the designer. Any additional constructors that you define should ensure that the default constructor is called so that view can be properly initialized via the **InitializeComponent** method. |

### Creating the View and View Model Using Unity

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

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

**C#**

public QuestionnaireView() { InitializeComponent(); } public QuestionnaireView(QuestionnaireViewModel viewModel) : this() { this.DataContext = viewModel; }

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| The default parameter-less constructor is necessary to allow the view to work in design-time tools, such as Visual Studio and Expression Blend. |

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

**C#**

public QuestionnaireView() { InitializeComponent(); } [Dependency] public QuestionnaireViewModel ViewModel { set { this.DataContext = value; } }

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

**C#**

IUnityContainer container; container.RegisterType<QuestionnaireViewModel>();

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

**C#**

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

## Creating the View and View Model Using an External Class

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

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

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

**C#**

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

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

**C#**

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

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| Prism provides extensive support for navigation within regions. Region navigation uses a mechanism very similar to the preceding approach, except that the region manager is responsible for coordinating the instantiation and placement of the view in the specific region. For 更多, see the section, "[View-Based Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#ViewBasedNavigation)" in Chapter 8, "[Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))." |

# Testing MVVM Applications

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

## Testing INotifyPropertyChanged Implementations

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

### Simple Cases

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

**C#**

var changeTracker = new PropertyChangeTracker(viewModel); viewModel.CurrentState = "newState"; CollectionAssert.Contains(changeTracker.ChangedProperties, "CurrentState");

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

### Computed and Non-Settable Properties

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

**C#**

var changeTracker = new PropertyChangeTracker(viewModel); var question = viewModel.Questions.First() as OpenQuestionViewModel; question.Question.Response = "some text"; CollectionAssert.Contains(changeTracker.ChangedProperties, "UnansweredQuestions");

### Whole Object Notifications

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

## Testing INotifyDataErrorInfo Implementations

There are several mechanisms available to enable bindings to perform input validation, such as throwing exceptions when properties are set, implementing the **IDataErrorInfo** interface, and (in Silverlight) implementing the **INotifyDataErrorInfo** interface. Implementing the**INotifyDataErrorInfo** interface allows for greater sophistication because it supports indicating multiple errors per property and performing asynchronous and cross-property validation; as such, it also requires the most testing.

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

### Testing Validation Rules

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

**C#**

// Invalid case var notifyErrorInfo = (INotifyDataErrorInfo)question; question.Response = -15; Assert.IsTrue(notifyErrorInfo.GetErrors("Response").Cast<ValidationResult>().Any()); // Valid case var notifyErrorInfo = (INotifyDataErrorInfo)question; question.Response = 15; Assert.IsFalse(notifyErrorInfo.GetErrors("Response").Cast<ValidationResult>().Any());

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

### Testing the Requirements for INotifyDataErrorInfo Implementations

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

There is no mandatory approach for implementing the **INotifyDataErrorInfo** interface. However, implementations that rely on objects that accumulate validation errors and perform the necessary notifications are typically preferred because they are simpler to test. This is because it is not necessary to verify that the requirements for all the members of the **INotifyDataErrorInfo** interface are met for each validation rule on each validated property (as long, of course, as the error management object is properly tested).

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

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

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

**C#**

var helper = new NotifyDataErrorInfoTestHelper<NumericQuestion, int?>( question, q => q.Response); helper.ValidatePropertyChange( 6, NotifyDataErrorInfoBehavior.Nothing); helper.ValidatePropertyChange( 20, NotifyDataErrorInfoBehavior.FiresErrorsChanged | NotifyDataErrorInfoBehavior.HasErrors | NotifyDataErrorInfoBehavior.HasErrorsForProperty); helper.ValidatePropertyChange( null, NotifyDataErrorInfoBehavior.FiresErrorsChanged | NotifyDataErrorInfoBehavior.HasErrors | NotifyDataErrorInfoBehavior.HasErrorsForProperty); helper.ValidatePropertyChange( 2, NotifyDataErrorInfoBehavior.FiresErrorsChanged);

## Testing Asynchronous Service Calls

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

The standard patterns used to implement asynchronous operations provide different guarantees regarding the thread in which notifications about the status of an operation occur. Although the [Event-based Asynchronous design pattern](http://msdn.microsoft.com/en-us/library/wewwczdw.aspx) guarantees that handlers for the events are invoked on a thread that is appropriate for the application, the [IAsyncResult design pattern](http://msdn.microsoft.com/en-us/library/ms228963.aspx) does not provide any such guarantees forcing the view model code that originates the call to ensure any changes that would affect the view are posted to the UI thread.

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

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

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

**C#**

questionnaireRepositoryMock .Setup( r => r.SubmitQuestionnaireAsync( It.IsAny<Questionnaire>(), It.IsAny<Action<IOperationResult>>())) .Callback<Questionnaire, Action<IOperationResult>>( (q, a) => callback = a); uiServiceMock .Setup(svc => svc.ShowView(ViewNames.QuestionnaireTemplatesList)) .Callback<string>(viewName => requestedViewName = viewName); submitResultMock .Setup(sr => sr.Error) .Returns<Exception>(null); CompleteQuestionnaire(viewModel); viewModel.Submit(); // Simulate callback posted to the UI thread. callback(submitResultMock.Object); // Check expected behavior – request to navigate to the list view. Assert.AreEqual(ViewNames.QuestionnaireTemplatesList, requestedViewName);

|  |
| --- |
| **g405494.note(en-us,PandP.40).gif注意:** |
| Using this testing approach only exercises the functional capabilities of the objects under test; it does not test that the code is thread safe. |

# 更多

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

For 更多 about attached properties, see "Attached Properties Overview" on MSDN:  
<http://msdn.microsoft.com/en-us/library/cc265152(VS.95).aspx>.

For 更多 about MEF, see "Managed Extensibility Framework Overview" on MSDN:  
<http://msdn.microsoft.com/en-us/library/dd460648.aspx>.

For 更多 about Unity, see "Unity Application Block" on MSDN:  
<http://www.msdn.com/unity>.

For 更多 about **DelegateCommand**, see Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

For 更多 about using Microsoft Expression Blend behaviors, see "Working with built-in behaviors" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ff724013(v=Expression.40).aspx>.

For 更多 about creating custom behaviors with Microsoft Expression Blend, see "Creating Custom Behaviors" on MSDN:   
<http://msdn.microsoft.com/en-us/library/ff724708(v=Expression.40).aspx>.

For 更多 about creating custom triggers and actions with Microsoft Expression Blend, see "Creating Custom Triggers and Actions" on MSDN:   
<http://msdn.microsoft.com/en-us/library/ff724707(v=Expression.40).aspx>.

For 更多 about using the dispatcher in WPF and Silverlight, see "Threading Model" and "The Dispatcher Class" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ms741870.aspx>  
<http://msdn.microsoft.com/en-us/library/ms615907(v=VS.95).aspx>.

For 更多 about unit testing in Silverlight, see "Unit Testing with Silverlight 2":  
<http://www.jeff.wilcox.name/2008/03/silverlight2-unit-testing/>.

For 更多 about region navigation, see the section, "[View-Based Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#ViewBasedNavigation)" in Chapter 8, "[Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))."

For 更多 about the Event-based Asynchronous pattern, see "Event-based Asynchronous Pattern Overview" on MSDN:  
<http://msdn.microsoft.com/en-us/library/wewwczdw.aspx>.

For 更多 about the IAsyncResult design pattern, see "Asynchronous Programming Overview" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ms228963.aspx>.

# 7: UI组合

|  |
| --- |
| **索引:** |
| [**UI布局概念**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec1) | [**Shell**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec2) | [**Views**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec3) – [*复合Views*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec4), [*Views和设计模式*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec5), [*命令, UI触发器, Actions, 和行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec6),[*命令*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec7), [*UI触发器, Actions,和行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec8), [*数据绑定*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec9) | [**Regions**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec10) – [*Region管理*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec11), [*实现Region*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec12), [*模块化User Control到Region的映射*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec13), [*默认Region功能*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec14), [*Region适配器*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec15), [*Region行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec16), [*注册行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec17), [*自动发现行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec18),[*Region上下文行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec19), [*激活行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec20), [*Region生命周期行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec21), [*控制特定行为*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec22), [*扩展Region实现*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec23) | [**View组成**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec24) – [*View发现*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec25), [*View注入*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec26), [*导航*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec27), [*何时使用View发现 vs. View注入*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec28) | [**UI布局方案**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec29) | [**实现Shell**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec30) - [*Stock Trader RI Shell*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec31) | [**定义Regions**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec32) - [*Stock Trader RI Shell Regions*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec33), [*IRegion*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec34),[*在XAML中*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec35)*添加Region*, [*通过代码添加Region*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec36) | [**当加载Region时在Region中显示Views**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec37) | [**通过代码在Region中显示Views**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec38) – [*导航*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec39) | [**在Region中排序Views**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec40) | [**多Region间共享数据**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec41) | [**创建一个Region的多个实例**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec42) | [**创建 Views**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec43) – [*用户控件*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec44), [*自定义控件*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec45), [*数据模板*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec46), [*资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec47), [*应用级资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec48), [*模块资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec49), [*控件资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec50) | [**UI设计**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec51)**指南** | [**交互设计指南**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec52) – [*容器构成*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec53),[*在设计时查看组合应用*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec54), [*布局*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec55), [*动画*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec56), [*运行时优化*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec57), [*设计时优化*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec58), [*拥有多XAML资源的大解决方案*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec59), [*XAML资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec60), [*视觉设计师和引用程序集*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec61), [*Silverlight的设计时App.xaml资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec62) | [**创建友好视图设计指南**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec63) – [*设计时的编码*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec64), [*在设计时使用代码缓解问题*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec65),[*理解在设计时运行用户控件的代码*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec66), [*设计时属性*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec67), [*d:DataContext属性*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec68), [*d:DesignInstance 装饰扩展*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec69), [*d:DataContext典型方案*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec70) | [**设计时示例数据设计指南**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec71) – [*使用设计时示例数据*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec72), [*示例数据资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec73), [*Expression Blend XML示例数据*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec74), [*Expression Blend 4和Visual Studio 2010 XAML示例数据*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec75), [*XAML资源*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec76), [*代码*](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec77) | [**UI布局关键决策**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec78) | [**更多**](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#sec79) |

An application user interface (UI) can be built by using one of the following paradigms:

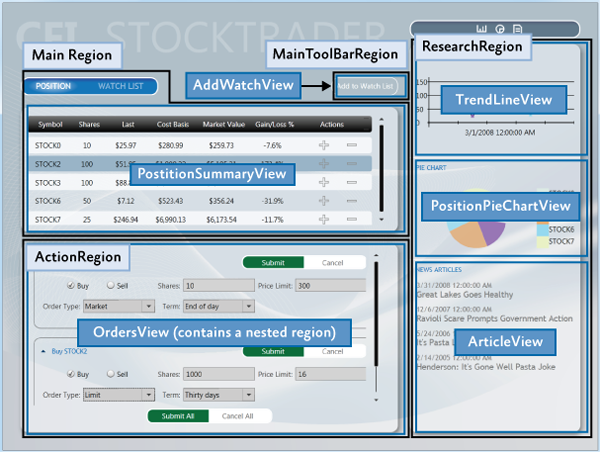
* All required controls for a form are contained in a single Extensible Application Markup Language (**XAML**) file, composing the form at design time.
* Logical areas of the form are separated into distinct parts, typically user controls. The parts are referenced by the form, and the form is composed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

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

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

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

**Stock Trader RI regions and views**



# UI布局概念

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

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

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

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

## Shell

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

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

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

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

## Views

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

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

### Composite Views

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

### Views and Design Patterns

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

The MVVM UI design pattern is recommended because it is a natural fit for the Microsoft XAML platforms, WPF, Silverlight, and Silverlight for Windows Phone 7. The dependency property system and rich data binding stack of these platforms enable the view and view model to communicate in a loosely coupled manner.

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

If you create a view with a user control or custom control and put all the logic in the code-behind file, your view can be difficult to test because you have to create an instance of the view to unit test the logic. This is a problem particularly if the view derives from, or depends on, running WPF or Silverlight components as part of its execution context. To make sure that you can unit test the view logic in isolation without these dependencies, you need to be able to create a mockup of the view to remove the dependencies on the execution context, which requires separate classes for the view and the logic.

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

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| Unit testing and UI automation testing are two different types of testing with different coverage. Unit testing best practices recommend that the object be tested in isolation. To achieve object isolation, you need a mockup or stub for each external dependency. Then granular 单元测试 are run against the object. UI automation testing runs the application, applies gestures to the UI, and then tests for the expected results. This type of test verifies that UI elements are correctly connected to the application logic. |

Separating the logic from the view provides a clean separation of concerns. In addition to testability considerations, this separation enables designers to work on the UI independently of the developer. For 更多 about MVVM, see Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

### Commands, UI Triggers, Actions, and Behaviors

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

#### Commands

Commands separate the semantics and the object that invokes a command from the logic that executes the command. Built into commands is the ability to indicate whether an action is available. Commands in the UI are data bound to **ICommand** properties on the view model. For 更多 about commands, see "[Commands](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#Commands)" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

#### UI Triggers, Actions, and Behaviors

Triggers, actions, and behaviors are part of the **Microsoft.Expression.Interactivity** namespace and are shipped with Expression Blend. They are also part of the Expression SDK. Triggers, actions, and behaviors provide a comprehensive API for handling UI events or commands, and then routing them to the **ICommand** properties methods exposed by the **DataContext**. For 更多 about UI triggers, actions, and behaviors, see sections "Invoking Command Objects from the View" and "Invoking Command Methods from the View" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

### Data Binding

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

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

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

You are strongly encouraged to read the "[Dependency Properties Overview](http://msdn.microsoft.com/en-us/library/ms752914.aspx)" and "[Data Binding Overview](http://msdn.microsoft.com/en-us/library/ms752347.aspx)" topics on MSDN. A full understand of these two topics is critical to successfully developing applications on the Microsoft XAML platforms. For 更多 about data binding, see "[Data Binding](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#DataBinding)" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

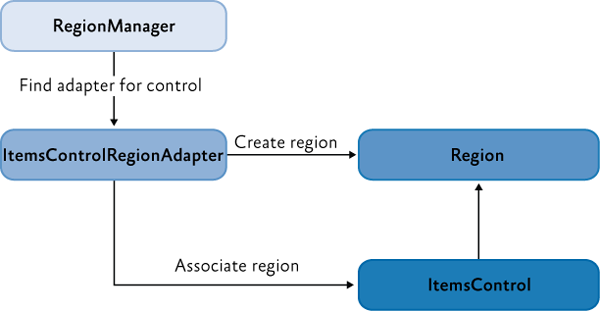
## Regions

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

### Region Manager

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

**Region, control, and adapter relationship**



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

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

The **RegionManager** provides a **RegionContext**attachedpropertythat permits its regions to share data.

### Region Implementation

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

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

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

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

**Stock Trader RI regions**



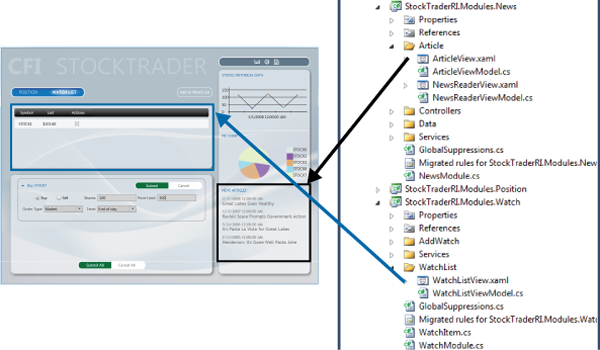
#### Module User Control to Region Mapping

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

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

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

**Module user control to region mapping**



### Default Region Functionality

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

The following sections describe the region adapter and region behaviors.

#### Region Adapter

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

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

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| The Silverlight version of the Prism Library contains a fourth region adapter named **TabControlRegionAdapter**. This is because the**TabControl** control in Silverlight 4 does 注意:xtend the **Selector** class and behaves differently from its WPF counterpart. |

#### Region Behaviors

The Prism Library introduces the concept of region behaviors. These are pluggable components that give a region most of its functionality. Region behaviors were introduced to support view discovery and region context (described later in this topic), and to create an API that is consistent across both WPF and Silverlight. Additionally, behaviors provide an effective way to extend a region's implementation.

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

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

#### Registration Behavior

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

#### Auto-Population Behavior

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

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

#### Region Context Behaviors

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

#### Activation Behavior

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

#### Region Lifetime Behavior

The **RegionMemberLifetimeBehavior** is responsible for determining if an item should be removed from the region when it is deactivated. The **RegionMemberLifetimeBehavior** monitors the region's **ActiveViews** collection to discover items that transition into a deactivated state. The behavior checks the removed items for **IRegionMemberLifetime** or the **RegionMemberLifetimeAttribute** (in that order) to determine if it should be kept alive on removal.

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

The region items are checked in the following order:

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

#### Control-Specific Behaviors

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

The **TabControlRegionSyncBehavior** is used for Silverlight only, and provides behavior that is similar to**SelectorItemsSourceSyncBehavior** for the Silverlight tab control.

### Extending the Region Implementation

The Prism Library provides extension points that allow you to customize or extend the default behavior of the provided APIs. For example, you can write your own region adapters, region behaviors, change the way the Navigation API parses URIs, or extend the Navigation API to work with Silverlight Frame Navigation. For 更多 about extending the Prism Library, see "[Extending Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))."

## View Composition

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

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

### View Discovery

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

### View Injection

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

### Navigation

The Prism Library 4.0 contains Navigation APIs. The Navigation APIs simplify the view injection process by allowing you to navigate a region to an URI. The Navigation API instantiates the view, adds it to the region, and then activates it. Additionally, the Navigation API allows navigating back to a previously created view contained in a region. For 更多 about the Navigation APIs, see Chapter 8, "[Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))."

### When to Use View Discovery vs. View Injection

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

Use view discovery in the following situations:

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

Use view injection in the following situations:

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

# UI Layout Scenarios

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

The decoupling of the view and the location in the UI in which it will be displayed allows the appearance and layout of the application to evolve independently of the views that appear within the region.

The next sections describe the core scenarios you will encounter when you develop a composite application. When appropriate, examples from the Stock Trader RI will be used to demonstrate a solution for the scenario.

## Implementing the Shell

The shell is the application root object in which the primary UI content is contained. In a Windows Presentation Foundation (WPF) application. The shell is the **Window** object. In a Silverlight application, the shell is the **RootVisualUserControl**.

A shell can contain named regions where modules can specify the views that will appear. It can also define certain top-level UI elements, such as the main menu and toolbar. The shell defines the overall structure and appearance for the application, and is similar to an ASP.NET master page control. It could define styles and borders that are present and visible in the shell layout itself, and it could also define styles, templates, and themes that are applied to the views that are plugged into the shell.

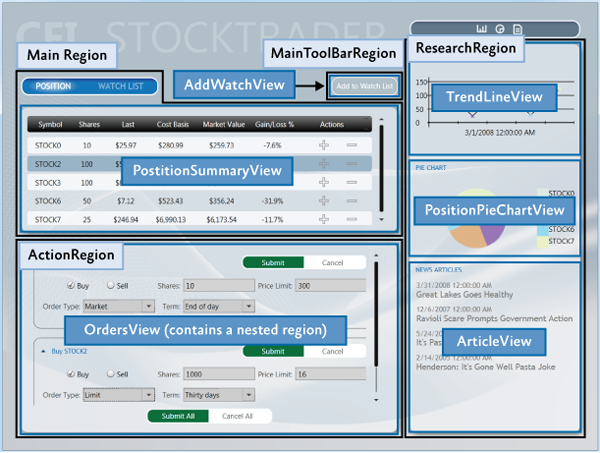
You do not need to have a distinct shell as part of your application architecture to use the Prism Library. If you are building a completely new composite application, implementing a shell provides a well-defined root and initialization pattern for setting up the main UI of your application. However, if you are adding Prism Library features to an existing application, you do not have to change the basic architecture of your application to add a shell. Instead, you can alter your existing window definitions or controls to add regions that can pull in views as needed.

You can also have more than one shell in your application. If your application is designed to open more than one top-level window for the user, each top-level window acts as shell for the content it contains.

### Stock Trader RI Shell

The WPF Stock Trader RI has a shell as its main window. In the following illustration, the shell and views are highlighted. The shell is the main window that appears when the Stock Trader RI starts and which contains all the views. It defines the regions into which modules add their views and a couple of top-level UI items, including the CFI Stock Trader title and the Watch List tear-off banner.

**Stock Trader RI shell window, regions, and views**



The shell implementation in the Stock Trader RI is provided by Shell.xaml, its code-behind file Shell.xaml.cs, and its view model ShellViewModel.cs. Shell.xaml includes the layout and UI elements that are part of the shell, including definitions of regions to which modules add their views.

The following XAML shows the structure and main XAML elements that define the shell. Notice that the **RegionName** attached property is used to define the four regions and that the window background image provides a background for the shell.

**XAML Shell.xaml (WPF)**

<Window x:Class="StockTraderRI.Shell"> <!—shell background --> <Window.Background> <ImageBrush ImageSource="Resources/background.png" Stretch="UniformToFill"/> </Window.Background> <Grid> <!-- logo --> <Canvas x:Name="Logo"> <TextBlock Text="CFI" ... /> <TextBlock Text="STOCKTRADER" .../> </Canvas> <!-- main bar --> <ItemsControl x:Name="MainToolbar" cal:RegionManager.RegionName="{x:Static inf:RegionNames.MainToolBarRegion}"> </ItemsControl> <!-- content --> <Grid> <Controls:AnimatedTabControl x:Name="PositionBuySellTab" cal:RegionManager.RegionName="{x:Static inf:RegionNames.MainRegion}"/> </Grid> <!-- details --> <Grid> <ContentControl x:Name="ActionContent" cal:RegionManager.RegionName="{x:Static inf:RegionNames.ActionRegion}"> </ContentControl> </Grid> <!-- sidebar --> <Grid x:Name="SideGrid"> <Controls:ResearchControl cal:RegionManager.RegionName="{x:Static inf:RegionNames.ResearchRegion}"> </Controls:ResearchControl> </Grid> </Grid> </Window>

The implementation of the **Shell** code-behind file is very simple. The **Shell** is exported so that when the bootstrapper creates it; its dependencies will be resolved by the Managed Extensibility Framework (MEF). The shell has its single dependency—the **ShellViewModel**—injected during construction, as shown in the following example.

**C# Shell.xaml.cs**

[Export] public partial class Shell : Window { public Shell() { InitializeComponent(); } [Import] ShellViewModel ViewModel { set { this.DataContext = value; } } }

**C# ShellViewModel.cs**

[Export] public class ShellViewModel : NotificationObject { // This is where any view model logic for the shell would go. }

The minimal code in the code-behind file illustrates the power and simplicity of the composite application architecture and loose coupling between the shell and its constituent views.

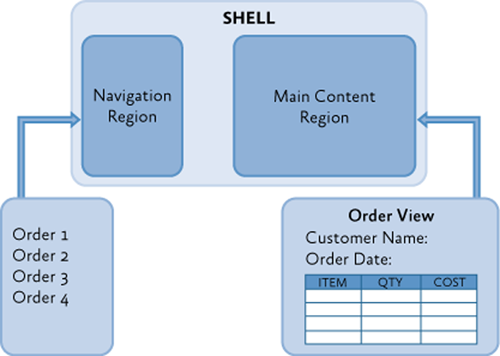
## Defining Regions

You define where views will appear by defining a layout with named locations, known as regions. Regions act as placeholders for one or more views that will be displayed at run time. Modules can locate and add content to regions in the layout without knowing how and where the region is displayed. This allows the layout to change without affecting the modules that add the content to the layout.

Regions are defined by assigning a region name to a WPF or Silverlight control, either in XAML as shown in the previous Shell.xaml file or in code. Regions can be accessed by their region name. At run time, views are added to the named Region control, which then displays the view or views according to the layout strategy that the views implement. For example, a tab control region will lay out its child views in a tabbed arrangement. Regions support the addition or removal of views. Views can be created and displayed in regions either programmatically or automatically. In the Prism Library, the former is achieved through view injection and the latter through view discovery. These two techniques determine how individual views are mapped to the named regions within the application UI.

The shell of the application defines the application layout at the highest level; for example, by specifying the locations for the main content and the navigation content, as shown in the following illustration. Layout within these high-level views is similarly defined, allowing the overall UI to be recursively composed.

**A template shell**



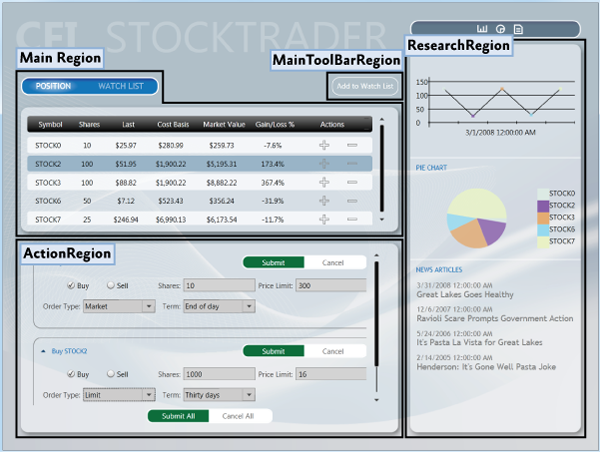
Regions are sometimes used to define locations for multiple views that are logically related. In this scenario, the region control is typically an **ItemsControl**-derived control that will display the views according to the layout strategy that it implements, such as in a stacked or tabbed layout arrangement.

Regions can also be used to define a location for a single view; for example, by using a **ContentControl**. In this scenario, the region control displays only one view at a time, even if more than one view is mapped to that region location.

### Stock Trader RI Shell Regions

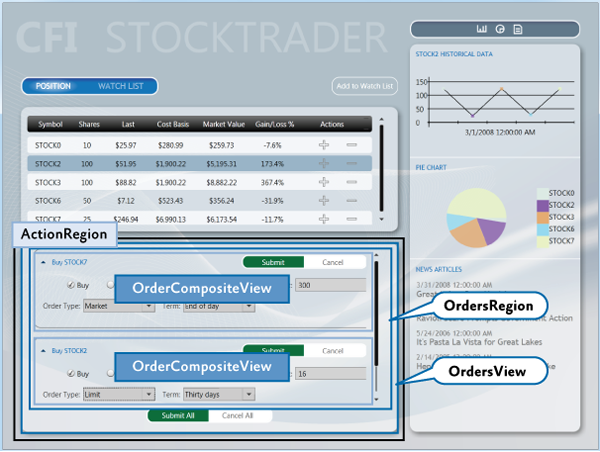
The Stock Trader RI shows the use of both the single view and the multiple view layout approaches. You can see both in the shell for the application, which defines locations for the application's high-level views. The following illustration shows the regions defined by the Stock Trader RI shell.

**Stock Trader RI shell regions**



A multiple-view layout is also demonstrated in the Stock Trader RI when the application is buying or selling a stock. The Buy/Sell area is a list-style region that shows multiple buy/sell views (**OrderCompositeView**) as part of its list, as shown in the following illustration.

**ItemsControl region**



The shell's **ActionRegion** contains the **OrdersView**. The **OrdersView** contains the **Submit All** and **Cancel All** buttons as well as the**OrdersRegion**. The **OrdersRegion** is attached to a **ListBox** control which displays multiple **OrderCompositeViews**.

### IRegion

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

**C#**

public interface IRegion : INavigateAsync, INotifyPropertyChanged { IViewsCollection Views { get; } IViewsCollection ActiveViews { get; } object Context { get; set; } string Name { get; set; } Comparison<object> SortComparison { get; set; } IRegionManager Add(object view); IRegionManager Add(object view, string viewName); IRegionManager Add(object view, string viewName, bool createRegionManagerScope); void Remove(object view); void Deactivate(object view); object GetView(string viewName); IRegionManager RegionManager { get; set; } IRegionBehaviorCollection Behaviors { get; } IRegionNavigationService NavigationService { get; set; } }

### Adding a Region in XAML

The **RegionManager** supplies an attached property that you can use for simple region creation in XAML. To use the attached property, you must load the Prism Library namespace into the XAML and then use the **RegionName** attached property. The following example shows how to use the attached property in a window with an **AnimatedTabControl.**

Notice the use of the **x:Static**markup extension to reference the **MainRegion** string constant. This practice eliminates magic strings in the XAML.

**XAML (WPF)**

<Controls:AnimatedTabControl x:Name="PositionBuySellTab" cal:RegionManager.RegionName="{x:Static inf:RegionNames.MainRegion}"/>

Silverlight 4 does not support **x:Static**. Therefore, you need to use string values for the region names or, optionally, define an application-level string resource for each region name. The **RegionName** attached property could then data binding to the string resource to resolve the region name.

**XAML (Silverlight)**

<Controls:AnimatedTabControl Regions:RegionManager.RegionName="MainRegion" />

### Adding a Region by Using Code

The **RegionManager** can register regions directly without using XAML. The following code example shows how to add a region to a control from the code-behind file. First a reference to the region manager is obtained. Then, using the **RegionManager** static methods**SetRegionManager** and **SetRegionName**, the region is attached to the UI's **ActionContent** control and then that region is named**ActionRegion**.

**C#**

IRegionManager regionManager = ServiceLocator.Current.GetInstance<IRegionManager>(); RegionManager.SetRegionManager(this.ActionContent, regionManager); RegionManager.SetRegionName(this.ActionContent, "ActionRegion");

## Displaying Views in a Region When the Region Loads

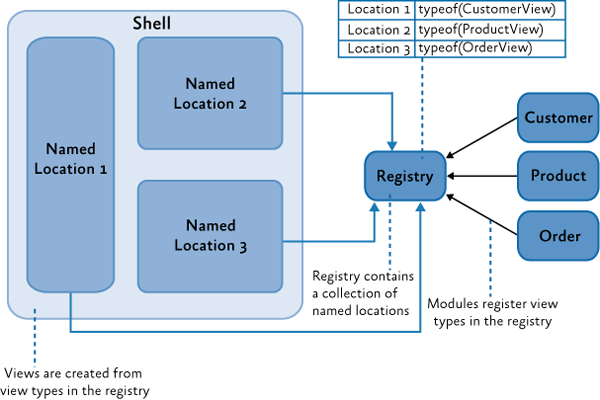
With the view discovery approach, modules can register views (view models or presentation models) for a specific named location. When that location is displayed at run time, any views that have been registered for that location will be created and displayed within it automatically.

Modules register views with a registry. The parent view queries this registry to discover the views that were registered for a named location. After they are discovered, the parent view places those views on the screen by adding them to the placeholder control.

After the application is loaded, the composite view is notified to handle the placement of new views that have been added to the registry.

The following illustration shows the view discovery approach.

**View discovery**



The Prism Library defines a standard registry, **RegionViewRegistry**, to register views for these named locations.

To show a view in a region, register the view with the region manager, as shown in the following code example. You can directly register a view type with the region, in which case the view will be constructed by the dependency injection container and added to the region when the control hosting the region is loaded.

**C#**

// View discovery this.regionManager.RegisterViewWithRegion("MainRegion", typeof(EmployeeView));

Optionally, you can provide a delegate that returns the view to be shown, as shown in the next example. The region manager will display the view when the region is created.

**C#**

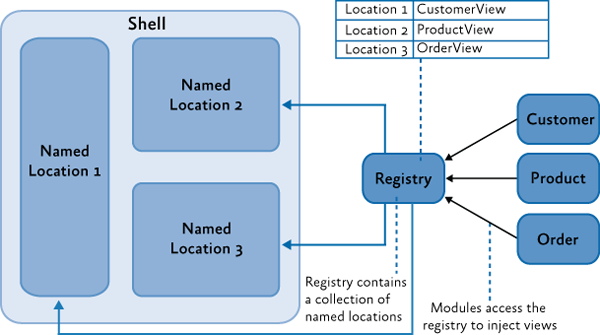
// View discovery this.regionManager.RegisterViewWithRegion("MainRegion", () => this.container.Resolve<EmployeeView>());

The UI Composition QuickStart has a walkthrough in the EmployeeModule ModuleInit.cs file that demonstrates how to use the**RegisterViewWithRegion** method.

## Displaying Views in a Region Programmatically

In the view injection approach, views are programmatically added or removed from a named location by the modules that manage them. To enable this, the application contains a registry of named locations in the UI. A module can use the registry to look up one of the locations and then programmatically inject views into it. To make sure that locations in the registry can be accessed similarly, each of the named locations adheres to a common interface used to inject the view. The following illustration shows the view injection approach.

**View injection**



The Prism Library defines a standard registry, **RegionManager**, and a standard interface, **IRegion**, for access these locations.

To use view injection to add a view to a region, get the region from the region manager, and then call the **Add** method, as shown in the following code. With view injection, the view is displayed only after the view is added to a region, which can happen when the module is loaded or when a user action completes a predefined action.

**C#**

// View injection IRegion region = regionManager.Regions["MainRegion"]; var ordersView = container.Resolve<OrdersView>(); region.Add(ordersView, "OrdersView"); region.Activate(ordersView);

In addition to the Stock Trader RI, the UI Composition QuickStart has a walkthrough for implementing view injection.

### Navigation

The Prism Library 4.0 includes Navigation APIs that provide a rich and consistent API for implementing navigation in a WPF or Silverlight application.

Region navigation is a form of view injection. When a navigation request is processed, it will attempt to locate a view in the region that can fulfill the request. If it cannot find a matching view, it calls the application container to create the object, and then injects the object into the target region and activates it.

The following code example from the Stock Trader RI **ArticleViewModel** illustrates how to initiate a navigation request.

**C#**

this.regionManager.RequestNavigate(RegionNames.SecondaryRegion, new Uri("/NewsReaderView", UriKind.Relative));

For 更多 about region navigation, see Chapter 8, "[Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))." The View-Switching Navigation QuickStart and State-Based Navigation QuickStart are also examples of implementing application navigation.

## Ordering Views in a Region

Whether it uses view discovery or view Injection, an application might need to order how views appear in a **TabControl**, **ItemsControl**, or any other control that displays multiple active views. By default, views appear in the order that they were registered and added into the region.

When a composite application is built, views are often registered from different modules. Declaring dependencies between modules can help alleviate the problem, but when modules and views do not have any real interdependencies, declaring an artificial dependency couples modules unnecessarily.

To allow views to participate in ordering themselves, the Prism Library provides the **ViewSortHint**attribute. This attribute contains a string**Hint** property that allows a view to declare a hint of how it should be ordered in the region.

When displaying views, the **Region** class uses a default view sorting routine that uses the hint to order the views. This is a simple case-sensitive ordinal sort. Views that have the sort hint attribute are ordered ahead of those without. Also, those without the attribute appear in the order they were added to the region.

If you want to change how views are ordered, the **Region** class provides a **SortComparison** property that you can set with your own**Comparison<object>** delegate method. It is important to 注意: that the ordering of the region's **Views** and **ActiveViews** properties are reflected in the UI because adapters such as the **ItemsControlRegionAdapter** bind directly to these properties. A custom region adapter could implement its own sorting and filter that will override how the region orders views.

The View Switching QuickStart demonstrates a simple numbering scheme to order the views in the left-hand-side navigation region. The following code examples show **ViewSortHint** applied to each of the navigation item views.

**C#**

[Export] [ViewSortHint("01")] public partial class EmailNavigationItemView [Export] [ViewSortHint("02")] public partial class CalendarNavigationItemView [Export] [ViewSortHint("03")] public partial class ContactsDetailNavigationItemView [Export] [ViewSortHint("04")] public partial class ContactsAvatarNavigationItemView

## Sharing Data Between Multiple Regions

The Prism Library provides multiple approaches to communicating between views, depending on your scenario. The region manager provides the **RegionContext** property as one of these approaches.

**RegionContext** is useful when you want to share context between a parent view and child views that are hosted in a region.**RegionContext** is an attached property. You set the value of the context on the region control so that it can be made available to all child views that are displayed in that region control. The region context can be any simple or complex object and can be a data-bound value. The**RegionContext** can be used with either view discovery or view injection.

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| The **DataContext** property in Silverlight and WPF is used to set the local data context for the view. It allows the view to use data binding to communicate with a view model, local presenter, or model. **RegionContext** is used to share context between multiple views and is not local to a single view. It provides a simple mechanism for sharing context between multiple views. |

The following code shows how the **RegionContext** attached property is used in XAML.

**XAML**

<TabControl AutomationProperties.AutomationId="DetailsTabControl" cal:RegionManager.RegionName="{x:Static local:RegionNames.TabRegion}" cal:RegionManager.RegionContext="{Binding Path=SelectedEmployee.EmployeeId}" ...>

You can also set the **RegionContext** in code, as shown in the following example.

**C#**

RegionManager.Regions["Region1"].Context = employeeId;

To retrieve the **RegionContext** in a view, the **GetObservableContext** static method of the **RegionContext** class is used. It passes the view as a parameter and then accesses its **Value** property, as shown in the following code example.

**C#**

private void GetRegionContext() { this.Model.EmployeeId = (int)RegionContext.GetObservableContext(this).Value; }

The value of the **RegionContext** can be changed from within a view by simply assigning a new value to its **Value** property. Views can opt to be notified of changes to the **RegionContext** by subscribing to the **PropertyChanged** event on the **ObservableObject** that is returned by the **GetObservableContext** method. This allows multiple views to be kept in synchronization when their **RegionContext** is changed. The following code example demonstrates subscribing to the **PropertyChanged** event.

**C#**

ObservableObject<object> viewRegionContext = RegionContext.GetObservableContext(this); viewRegionContext.PropertyChanged += this.ViewRegionContext\_OnPropertyChangedEvent; private void ViewRegionContext\_OnPropertyChangedEvent(object sender, PropertyChangedEventArgs args) { if (args.PropertyName == "Value") { var context = (ObservableObject<object>) sender; int newValue = (int)context.Value; } }

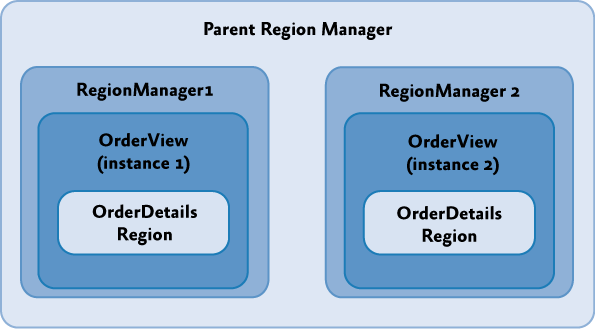
|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| The **RegionContext** is set as an attached property on the content object hosted in the region. This means that the content object has to derive from **DependencyObject**. In the preceding example, the view is a visual control, which ultimately derives from**DependencyObject**.  If you choose to use WPF or Silverlight data templates to define your view, the content object will represent the **ViewModel** or**PresentationModel**. If your view model or presentation model needs to retrieve the **RegionContext**, it will need to derive from the**DependencyObject** base class. |

## Creating Multiple Instances of a Region

Scoped regions are available only with view injection. You should use them if you need a view to have its own instance of a region. Views that define regions with attached properties automatically inherit their parent's **RegionManager**. Usually, this is the global **RegionManager**that is registered in the shell window. If the application creates more than one instance of that view, each instance would attempt to register its region with the parent **RegionManager**. **RegionManager** allows only uniquely named regions; therefore, the second registration would produce an error.

Instead, use scoped regions so that each view will have its own **RegionManager** and its regions will be registered with that**RegionManager** rather than the parent **RegionManager**, as shown in the following illustration.

**Parent and scoped RegionManagers**



To create a local **RegionManager** for a view, specify that a new **RegionManager** should be created when you add your view to a region, as illustrated in the following code example.

**C#**

IRegion detailsRegion = this.regionManager.Regions["DetailsRegion"]; View view = new View(); bool createRegionManagerScope = true; IRegionManager detailsRegionManager = detailsRegion.Add(view, null, createRegionManagerScope);

The **Add** method will return the new **RegionManager** that the view can retain for further access to the local scope.

## Creating Views

The visual representation of your application can take many forms, including user controls, custom controls, and data templates, to name a few. In the case of the Stock Trader RI, user controls are typically used to represent distinct sections on the main window, but this is not a standard. In your application, you should use an approach that you are most familiar with and that fits into how you work as a designer. Regardless of the predominating visual representation in your application, you will inevitably use a combination of user controls, custom controls, and data templates in your overall design. The following figure shows where the Stock Trader RI uses these various items. This illustration also serves as a reference for the following sections, which describe each of the items.

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



### User Controls

Both Expression Blend and Visual Studio 2010 provide rich support for creating user controls. User controls created with these tools are therefore recommended for creating UI content with the Prism Library. As mentioned earlier in this topic, the Stock Trader RI uses them extensively to create content that will be inserted into regions. The **WatchListView.xaml** user control is a good example of a simple UI representation that is contained inside the **WatchModule**. This control is a very simple control that is straightforward to create using this model.

### Custom Controls

In some situations, a user control is too limiting. In these cases, custom layout or extensibility is more important than ease of creation. This is where custom controls are useful. In the Stock Trader RI, the pie chart control is a good example of this. This control is composed from data derived from the positions and shows a chart of the overall portfolio. This type of control is a little more challenging than a user control to create, and it has limited visual design support in Expression Blend and Visual Studio 2010, compared to a user control.

### Data Templates

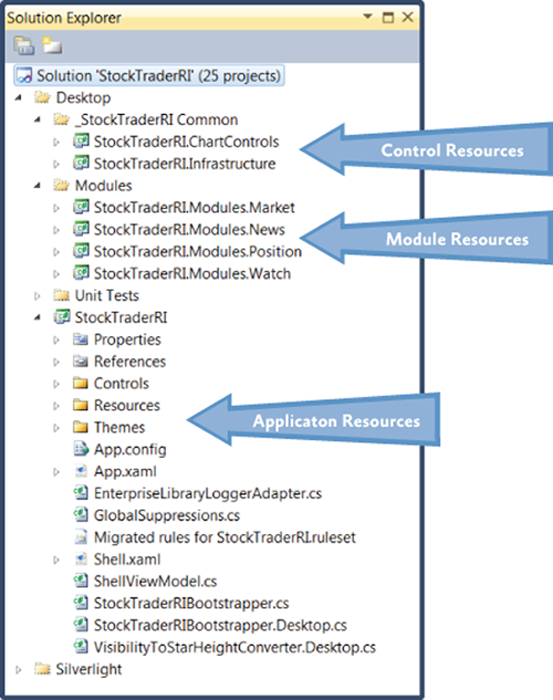
Data templates are an important part of most types of data-driven applications. The use of data templates for list-based controls is prevalent throughout the Stock Trader RI. In many cases, you can use a data template to create complete visual representations without needing to create any type of control. The **ResearchRegion** uses a data template to show articles and, in conjunction with an **Items** style, provides an indication of which item was selected.

Expression Blend has full visual design support for data templates. Visual Studio 2010 only provides support for using the XAML editor to edit data templates.

### Resources

Resources such as styles, resource dictionaries, and control templates can be scattered throughout an application. This is especially true with a composite application. When you consider where to place resources, pay special attention to dependencies between UI elements and the resources they need. The Stock Trader RI solution, shown in the following figure, contains labels that indicate the various areas where resources can live.

**Resource distribution across a solution**



#### Application Resources

Typically, application resources are resources that are available to an application as a whole. These resources tend to be focused on the root application, but they can also provide default styling on a type basis for modules or controls. An example of this is a text box style that is applied to the text box type in the root application. This style will be available to all text boxes in the application unless the style is overridden at the module or control level.

#### Module Resources

Module resources play the same role as root application resources in that they can apply to all items in a module. Using resources at this level can provide a consistent appearance across the entire module and can also allow for reuse in more specific instances that span one or more visual components. The use of resources at the module level should be contained within the individual module. Creating dependencies between modules can lead to issues that are difficult to locate when UI elements appear incorrectly.

#### Control Resources

Control resources are usually contained in control libraries and can be used by all the controls in the control library. These resources tend to have the most limited scope because control libraries typically contain very specific controls and do not contain user controls. (In an application created with the Prism Library, user controls are typically placed in the modules in which they are used.)

# UI Design Guidance

The goal of this topic is to provide some high-level guidance to the XAML designer and developer who are building an application with the Prism Library and WPF or Silverlight. This topic describes UI layout, visual representation, data binding, resources, and the presentation model. After reading this topic, you should have a high-level understanding of how to approach designing the UI of an application based on the Prism Library and some of the techniques that can help you create a maintainable UI in composite applications.

## Guidelines for Designing User Interfaces

The layout of composite applications created with the Prism Library builds on the standard principals of WPF and Silverlight—the layout uses the concepts of panels that contain related items. However, with composite applications, the content inside the various panels is dynamic and is not known during design time. This forces designers and developers to create page structures that can contain layout content and then design each of the elements that fit into the layout separately. As a designer or developer, this means that you have to think about two main layout concepts in the Prism Library: container composition and regions.

### Container Composition

Container composition is really just an extension of the containment model that WPF and Silverlight inherently provide. The term containercan mean any element, including a window, page, user control, panel, custom control, control template, or data template, that can contain other elements.

How you visualize your UI can vary from implementation to implementation, but you will find recurring themes that stand out. You will create a window, page, or user control that contains both fixed content and dynamic content. The fixed content will consist of the overall structure of the containing UI element, and the dynamic content will be what is placed inside a region.

For example, the WPF Stock Trader RI has a startup window named Shell.xaml that contains the overall structure for the application. The next illustration shows the shell loaded in Expression Blend. Notice that only the fixed portion of the UI is visible. The remaining sections of the shell are dynamically inserted into the various regions by the modules as the application loads.

The design-time experience is a little limited in this type of application, but the fact that you know content will be placed in the various regions at run time is something that you need to design for. To see an example of this, compare the designer view of the main page in the next illustration to the run-time view in the illustration that follows it. In the designer view, the page is mostly empty. Contrast that with the run-time view, where there is a position area that contains a tab control with position data, and a trend line, pie chart, and news area pertaining to the selected stocks. The differences between the designer view and run-time view demonstrate the challenges designers and developers face when they create applications built with the Prism Library.

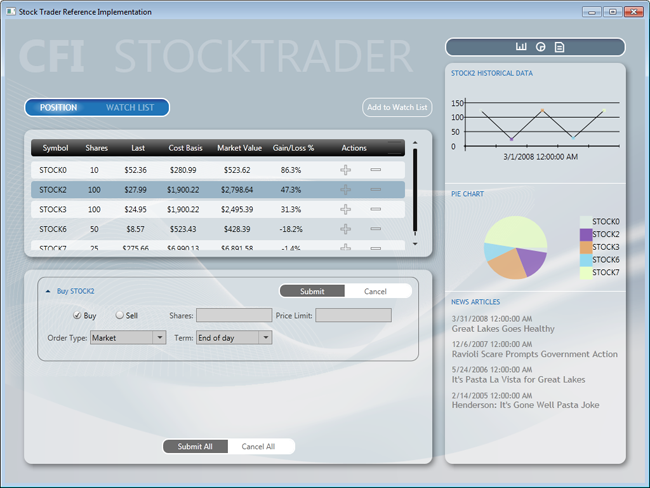
The items cannot be seen during design time; therefore, determining how big they are and how they fit into the overall appearance of the application is a little difficult. Consider the following as you create the layout for your containers:

* Are there any size constraints that will limit how large content can be? If there are, consider using containers that support scrolling.
* Consider using an expander and **ScrollViewer** combination for situations in which a large amount of dynamic content needs to fit into a confined area.
* Pay close attention to how content enlarges as the screen content grows to ensure that the appearance of your application is appealing in any resolution.

**Stock Trader RI main window in Expression Blend**



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



### Viewing Composite Application at Design Time

The two previous figures illustrate one of the challenges of working with high-level views that are composed at run time. Each UI element in a composite application must be designed separately. This makes it hard to visualize how the composite page or window will look at run time. To visualize the composite view in its composed state, you can create a test project with a page or window that contains all the UI elements for the view you want to test.

* Additionally, consider using the design-time sample data features in Expression Blend and Visual Studio 2010 to populate UI elements with data. Design-time data is very helpful when you work with data templates, list controls, charts, or graphs. For 更多, see the section "[Guidelines for Design-Time Sample Data](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#GuidelinesforDesignTimeSampleData)."

### Layout

Consider the following when you design the layout of a composite application:

* The shell defines the main layout of the application. Each area of the layout is a region and should be kept as an empty container. Do not place content inside regions at design time because content will be loaded there at run time.
* The shell should contain the background, titles, and the footer. Think of the shell as an ASP.NET master page**.**
* Control containers that act as regions are decoupled from the views that they contain. Therefore, you should be able to change the size of the views without modifying the controls, and you should be able to change the size of the controls without modifying the views. You should consider the following when defining the size of a view:
  + If a view will be used in several regions or if it is uncertain where it will be used, design it with dynamic width and height.
  + If the views have fixed sizes, the regions of the shell should use dynamic sizes.
  + If the shell regions have fixed sizes, the views should use dynamic sizes.
  + Views might require a fixed height and dynamic width. An example of this is the **PositionPieChart** view located in the sidebar of the Stock Trader RI.
  + Other views might require a dynamic height and width**.** For example, the **NewsReader** views in the sidebar of the Stock Trader RI. The height itself depends on the title's length, and the width should always adapt to the region's size (sidebar width). The same applies to the **PositionSummaryView** view, where the grid's width should adapt to the screen size and the height should adapt to the number of rows in the grid.
* Views should generally have transparent backgrounds, allowing the shell background to provide the application visual background.
* Always use named resources for assigning colors, brushes, fonts and font sizes, rather than directly assigning the property value in XAML. This makes application maintenance much easier over time. It also allows an application to respond to changes in resource dictionaries at run time.

### Animation

Consider the following when you use animation in the shell or views:

* You can animate the layout of the shell, but you will have to animate its contents and views separately.
* Design and animate each view separately.
* Use soft or gentle animations to provide a visual clue that a UI element is being brought into view or being removed from view. This gives an application a polished look and feel.

Expression Blend offers a rich set of behaviors, easing functions, and an outstanding editing experience for animating and transitioning UI elements based on visual state changes or events. For 更多, see the "[VisualStateManager Class](http://msdn.microsoft.com/en-us/library/cc626338(v=VS.95).aspx)" topic on MSDN.

### Run-Time Optimization

Consider the following tips for performance optimization:

* Place any common resources in the App.xaml file or a merged dictionary to avoid duplicating the styles.
* In Silverlight, avoid using non-system fonts for specific static text that should be styled differently from the rest of the application (for example, the application title). In this case, determine if it is better to convert the text to a path or to embed the font.

Embedding a font can affect the size of the downloaded .xap file because some font files are rather large. To minimize the size of the downloaded .xap file, Expression Blend allows you to download a subset of the font characters instead of the entire font.

For 更多 about this, see "[Using Custom Fonts in Silverlight](http://silverlight.net/learn/learnvideo.aspx?video=69800)."

### Design-Time Optimizations

The following sections describe design-time scenarios and provide solutions for making the most of the design-time experience.

#### Large Solutions with Many XAML Resources

In large applications with many XAML resources that are part of the solution, visual designer load time can be affected, sometimes significantly. This performance slowdown exists because the visual designer must parse all merged XAML resources. The solution to this problem is to move all XAML resources to another solution, compile that solution, and then reference the new XAML resource DLL from the large solution. Because the XAML resources are in a binary referenced assembly, the visual designer does not parse the XAML resources, thus improving design-time performance. When moving XAML resources to an external assembly, you might want to consider exposing**ComponentResourceKeys** for your resources**.** For 更多, see the "[ComponentResourceKey Markup Extension](http://msdn.microsoft.com/en-us/library/ms753186.aspx)" topic on MSDN.

#### XAML Assets

XAML is a powerful and expressive language for creating assets such as images, diagrams, drawings, and 3-D scenes. Some developers and designers prefer creating XAML assets instead of using .ico, .jpg, or .png image files. One reason that they prefer the XAML approach is to take advantage of the resolution independence of XAML rendering. Another is that they can use one tool set, the Expression Suite, to create all the required assets and design their applications.

If the solution has many of these assets, design-time visual designer loading can be affected. Moving assets to a separate DLL solves the performance problem. Moving the assets also enables reuse across multiple solutions.

#### Visual Designers and Referenced Assemblies

An unfortunate side-effect of moving XAML resources and assets to a binary referenced assembly is that the Expression Blend and Visual Studio 2010 property editors do not list resources located in binary referenced assemblies. This means that you will not be able to pick a named resource from one of the resource pickers provided by the tools. Instead, you will need to type the name of the resource.

### Silverlight Design-Time App.xaml Resources

Silverlight composite applications can be structured in several ways to allow run-time delayed loading of assemblies or to reduce initial .xap download size. One strategy that you can use is to create your main Silverlight application, and then create a satellite assembly for each module. When you add the satellite project assembly, you can choose one of the Silverlight application project templates or the Silverlight class library project template.

Choosing a Silverlight application project template for a satellite assembly provides a deployment benefit: the assembly will be packaged as a .xap file when built. However, there is a side effect that affects the visual designers when more than one Silverlight application is present in a single solution. The main Silverlight application will have its application resources merged in App.xaml. The problem is that the visual designer consumes resources exposed in App.xaml. Because there is more than one Silverlight application in the solution, the visual designer does not attempt to use resources from another Silverlight application; instead, it uses only those resources from the active Silverlight application.

Expression Blend 4 provides a solution to this problem. When Expression Blend detects this condition, it displays a dialog box that allows you to select a design time resource dictionary that will be used across all projects in the solution.

Visual Studio 2010 does not have this feature; therefore, satellite Silverlight application assemblies will not have the normal rich visual design-time experience unless you merge application-level resources in the local assembly. If you choose to merge application-level resources for a better visual design experience, you must remember to remove them before deploying your application.

## Guidelines for Creating Designer Friendly Views

The following are some of the characteristics of a designer friendly (also known as a blendable or tool-able) application:

* It provides a productive editing experience by using the Visual Studio and Expression Blend designers.
* It is tooling-enabled. For example, it allows you to use the binding builder.
* It provides design-time sample data when required.
* It allows code to be executed at design time without causing unhandled exceptions.

The following actions are performed many times during an editing session. User code that is not designer friendly will cause one or more of these actions to fail, thus reducing the productivity and creativity of a developer or designer.

* Design surface actions:
  + Constructing objects
  + Loading objects
  + Setting property values
  + Performing design surface gestures
  + Using a control as the root element
  + Hosting a control inside another control
  + Opening, closing, and reopening a XAML file repeatedly
  + Rebuilding the project
  + Reloading the designer
* Binding builder actions:
  + Discovering the **DataContext**
  + Listing the available data sources
  + Listing data source type properties
* Design-time sample data actions:
  + Using controls on the design surface to correctly display sample data

### Coding for Design Time

To give you a rich design-time experience, the Visual Studio and Expression Blend designers instantiate objects and run code at design time. However, null reference exceptions caused by code that attempts to access a reference type before it has been instantiated cause a high percentage of loading failures and unnecessary design time exceptions.

The following table lists the main causes of poor design-time experiences. By avoiding the following issues and using the techniques to mitigate these problems, your design-time experience and productivity will be greatly enhanced, and the developer-to-designer workflow will be much smoother.

|  |  |  |
| --- | --- | --- |
| **Avoid This in User Code** | **Visual Studio 2010** | **Blend 4** |
| Spinning multiple threads at design time. For example, instantiating and starting a **Timer**in a constructor or **Loaded** event at design time. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Using controls that cause stack overflows at design time.  Using controls that attempt to recursively load themselves. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Throwing null reference exceptions in converters or data template selectors. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Throwing null reference or other exceptions in constructors. These are caused by:   * Using code that calls into the business or data layers to return data from a database or over the network at design time. * Attempting to resolve dependencies by using MEF, inversion of control (IoC), or a Service Locator before bootstrapping or container initialization code has run. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Throwing null reference or other exceptions inside the **Loaded** events of controls or user controls. This happens when you make assumptions about the state of the control that might be true at run time but are not true at design time. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Attempting to access the **Application** or **Application.Current** object at design time. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Consuming a **StaticResource** in WPF **UserControls**. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |
| Creating very large projects. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. | Incorrectly formatted figure.  pandpDocTools-topicTitle: 7: Composing the User Interface  errorText: Incorrectly formatted figure. |

### Mitigating Problems in Design-Time User Code

A few defensive coding practices will eliminate most of the issues described in the preceding table. However, before you can mitigate problems in design-time user code, you must understand that your application controls and code are being executed by the designer in isolation, inside an uninitialized application domain. Uninitialized in this case means that the usual startup, bootstrapping, or initialization code has not run.

When your application executes at run time, the startup code in App.xaml.cs or App.xaml.vb is run. If you have code in there that the rest of your application depends on, this code will not have been executed at design time. If you have not anticipated this in your code, unwanted exceptions will occur. (This is why attempting to access the **Application** or **Application.Current** object in user code at design time will result in exceptions.) To mitigate these issues:

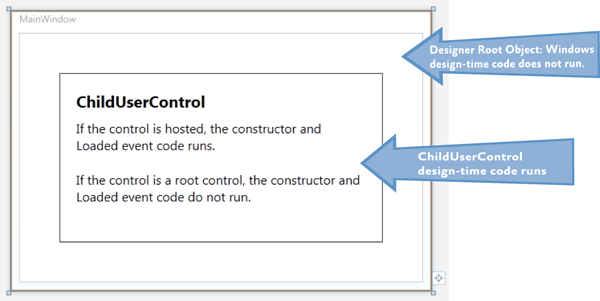
* Never assume that referenced objects will be instantiated in design-time code. In code that can be executed at design time, always perform a null check before accessing any reference object.
* If your code accesses the **Application** or **Application.Current** objects, perform a null reference check before accessing the object.
* If your constructors or **Loaded** event handlers need to run complex code or code that accesses a database or calls out to the network, consider one of the following solutions:
  + Wrap the code inside a check that determines if the code is running at design time by calling one of the below**System.ComponentModel DesignerProperties** methods:
    - **WPF**: DesignerProperties.GetIsInDesignMode
    - **Silverlight**: DesignerProperties.IsInDesignTool
  + Instead of running the code directly in the constructor or **Loaded** event handler, abstract the calls to a class behind an interface, and then use one of many techniques to resolve that dependency differently at design time, run time, and test time.

For example, instead of calling out to a data service directly to retrieve data, wrap the data service calls in a class that exposes the methods through an interface. Then, at design time, resolve the interface with a mock or design-time object.

### Understanding when User Control Code Executes at Design-Time

Both Expression Blend and Visual Studio use mockups of the root object displayed in a designer pane. This is necessary to provide the required design experience. Because the root object is mocked, its constructor and **Loaded** event code are 注意:xecuted at design time. However, the remaining controls in the scene are constructed normally, and their **Loaded**event is raised just like at run time.

In the following illustration, the root **Windows** constructor and **Loaded** event code will not be executed. The child user controls constructor and **Loaded** event code will be executed.



These concepts are important, especially if you are building composite applications or applications that are built dynamically at run time.

Most application views are coded and designed independently. Because they are designed independently, they are typically the root object in the designer. Because of this, their constructor and **Loaded** event code never executes.

However, if you take that same user control and place it on a design surface as a child of another control, the once isolated user control code is now executing at design time. If you have not followed the above practices for mitigating design-time code problems, the now hosted user control could become unfriendly and cause designer load issues.

### Design-Time Properties

The built-in "d:" design-time properties provide a smooth road to a successful design-time tooling experience.

The problem we need to solve is how to provide a shape to the Binding Builder tools at design-time. In this case, the shape is an instantiated **Type** that the Binding Builder can reflect on, and then list those properties for selection when building a binding.

Shape is also provided by design-time sample data. Sample data is covered in the section, "[Guidelines for Design-Time Sample Data](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40)#GuidelinesforDesignTimeSampleData)."

The following sections describe how to use the **d:DataContext** property and the **d:DesignInstance** markup extension.

The "d:" in the property and markup extension is the alias for the design namespace that the design properties are members of. The following are links to MSDN topics that cover the "d:" properties and markup extensions:

* "[Design-Time Attributes in the WPF Designer](http://msdn.microsoft.com/en-us/library/ee839627.aspx)"
* "[Design-Time Attributes in the Silverlight Designer](http://msdn.microsoft.com/en-us/library/ff602277(VS.95).aspx)"

The "d:" properties and markup extensions cannot be created or extended in user code; they can only be used in XAML. The "d:" properties and markup extensions are not compiled into your application; they are used only by the Visual Studio and Expression Blend tooling.

#### d:DataContext Property

**d:DataContext**, specifies a design-time data context for a control and its children. When specifying **d:DataContext**, you should always provide the same shape to the design-time **DataContext** as the run-time **DataContext**.

If both a **DataContext** and a **d:DataContext** are specified for a control, the tooling will use the **d:DataContext**.

#### d:DesignInstance Markup Extension

If markup extensions are new to you, read the "[Markup Extensions and WPF XAML](http://msdn.microsoft.com/en-us/library/ms747254.aspx)" topic on MSDN.

**d:DesignInstance** returns an instantiated Type ("shape") that you will want to assign as the data source for binding to controls in the designer. The type does not need to be creatable to be used for establishing shape. The following table explains the **d:DesignInstance**markup extension properties.

|  |  |
| --- | --- |
| **Markup Extension Property** | **Definition** |
| **Type** | Name of the Type that will be created. Type is the default parameter in the constructor. |
| **IsDesignTimeCreatable** | Can the specified Type be created? If false, a faux Type will be created rather than the real Type. The default is false. |
| **CreateList** | If true, returns a generic list of the specified Type. The default is false. |

#### Typical d:DataContext Scenario

The following three code examples demonstrate a repeatable pattern for wiring up views and view models and enabling the designer's tooling.

The **PersonViewModel** is a dependency that the **PersonView** has at run time. While the view model in the example is incredibly simple, real-world view models typically have one or more external dependencies that must be resolved, and those dependencies are typically injected into their constructor.

When the **PersonView** is constructed, its dependency **PersonViewModel** will be built and its dependencies resolved by MEF or a dependency injection container.

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| If the view model has no external dependencies that need to be resolved, the view model can be instantiated in the view's XAML, and its**DataContext** and the **d:DataContext** are not required. |

**C# PersonViewModel.cs**

[Export] public class PersonViewModel { public String FirstName { get; set; } public String LasName { get; set; } }

**C# PersonView.xaml.cs**

[Export] public partial class PersonView : UserControl { public PersonView() { InitializeComponent(); } [Import] public PersonViewModel ViewModel { get { return this.DataContext as PersonViewModel; } set { this.DataContext = value; } } }

This is a good pattern for wiring up a view and view model; however, it leaves the view unaware of its **DataContext**'s shape (view model) at design time.

In the following XAML example, you can see the **d:DesignInstance** markup extension used on the **Grid** to return a faux instance of**PersonViewModel** that is then exposed by the **d:DataContext**. As a result, all child controls of the **Grid** will inherit the **d:DataContext**, enabling the designer tooling to discover and use its types and properties, resulting in a more productive design experience for developers and designers.

**XAML PersonView.xaml**

<UserControl xmlns:local="clr-namespace:WpfApplication1" x:Class="WpfApplication1.PersonView" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:d="http://schemas.microsoft.com/expression/blend/2008" mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300"> <Border BorderBrush="LightGray" BorderThickness="1" CornerRadius="10" Padding="10"> <Grid d:DataContext="{d:DesignInstance local:PersonViewModel}"> <Grid.RowDefinitions> <RowDefinition Height="Auto" /> <RowDefinition Height="Auto" /> </Grid.RowDefinitions> <Grid.ColumnDefinitions> <ColumnDefinition Width="100" /> <ColumnDefinition Width="Auto" /> </Grid.ColumnDefinitions> <Label Grid.Column="0" Grid.Row="0" Content="First Name" /> <Label Grid.Column="0" Grid.Row="1" Content="Las Name" /> <TextBox Grid.Column="1" Grid.Row="0" Width="150" MaxLength="50" HorizontalAlignment="Left" VerticalAlignment="Top" Text="{Binding Path=FirstName, Mode=TwoWay}" /> <TextBox Grid.Column="1" Grid.Row="1" Width="150" MaxLength="50" HorizontalAlignment="Left" VerticalAlignment="Top" Text="{Binding Path=LasName, Mode=TwoWay}" /> </Grid> </Border> </UserControl>

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| **Attached Property and ViewModel Locator Solution** There are several alternative techniques for associating a view and view model available from the developer community. One of the challenges is that solutions that work great at run time do not always work at design time. One such solution is the use of attached properties and view model locators to assign a view's **DataContext**. The view model locator is required so that the view model can be constructed and have its dependencies resolved. The problem with this solution is that you must also include the **d:DataContext** – **d:DesignInstance** combination because the visual designer tooling cannot be reflected in the results of the attached property the way that it can with the **d:DesignInstance**. Regardless of which technique you implement in your applications for resolving shape at design time, the most important goal is to be consistent throughout your application. Consistency will make application maintenance much easier and will lead to a successful designer-developer workflow. |

## Guidelines for Design-Time Sample Data

The WPF and Silverlight Designer team published an in-depth, scenario-based training article that discusses the use of sample data in WPF and Silverlight projects. The article, "[Sample Data in the WPF and Silverlight Designer](http://blogs.msdn.com/b/wpfsldesigner/archive/2010/06/30/sample-data-in-the-wpf-and-silverlight-designer.aspx)," is available on MSDN.

### Using Design-Time Sample Data

If you use a visual design tool, such as Expression Blend or Visual Studio 2010, design-time sample data becomes very important. The views can be populated with data and images, making the design task easier and quicker to accomplish. This results in improved productivity and creativity.

Empty list controls that contain data templates will not be visible unless they are populated with data, making the task of editing the empty controls more time consuming because you need to run the application to see how the last edit will look at run time.

### Sample Data Sources

You can use sample data from any of the following sources:

* Expression Blend XML sample data
* Expression Blend 4 and Visual Studio 2010 XAML sample data
* XAML resources
* Code

The data from each of these sources is described in the following subsections.

#### Expression Blend XML Sample Data

Expression Blend gives you the capability to quickly create an XML schema and populate a corresponding XML file. This is accomplished without any dependency on any project classes.

The purpose of this type of sample data is to let designers start their projects quickly, without waiting for a developer or before application classes are available for consumption.

While most sample data is supported in both the Expression Blend and Visual Studio designers, XML sample data is an Expression Blend feature and does not render in the Visual Studio designer.

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| XML sample data file is not compiled or added to the assembly when built; however, the XML schema is compiled into the built assembly. |

#### Expression Blend 4 and Visual Studio 2010 XAML Sample Data

Beginning in Expression Blend 4 and Visual Studio 2010, the **d:DesignData** markup extension was added to enable the design-time loading of XAML sample data.

Sample data XAML files contain XAML that instantiates one or more types and assigns values to properties.

**d:DesignData** has a **Source** property that takes a uniform resource identifier (URI) to the sample data XAML file located in the project. The**d:DesignData** markup extension loads the XAML file, parses it, and then returns an object graph. The object graph can be consumed by the**d:DataContext** property, **CollectionViewSource d:DesignSource** property, or **DomainDataSource d:DesignData** property.

One of the challenges that the **d:DesignData** markup extension overcomes is that it can create sample data for non-creatable user types. For example, WCF Rich Internet Application (RIA) Services entity–derived objects cannot be created in code. In addition, developers might have their own types that are not creatable, but would still like to have sample data for these types.

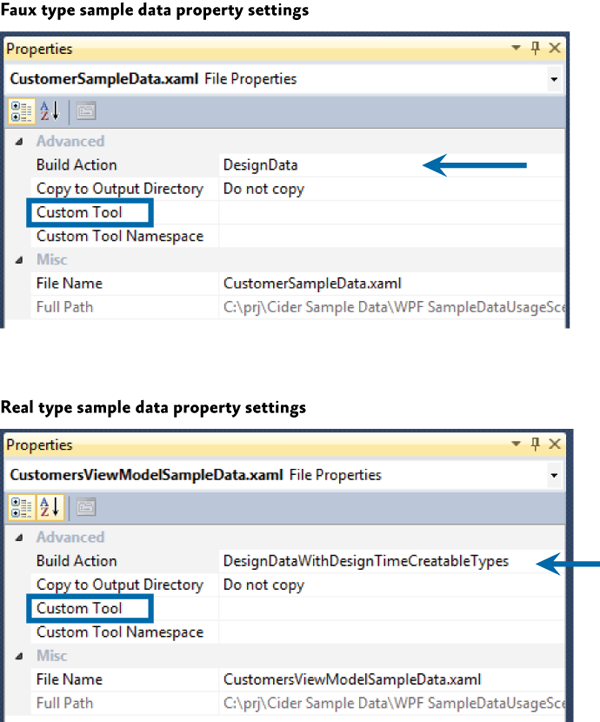
You can change how **d:DesignData** processes your sample data file by setting the **Build Action** property on the sample data file in the**Solution Explorer** as follows:

* **Build Action = DesignData** – faux types will be created
* **Build Action = DesignDataWithDesignTimeCreatableTypes** – real types will be created

When Expression Blend is used to create sample data for a class, it creates a XAML sample data file with the **Build Action** set to**DesignData**. If you require real types, open the solution in Visual Studio and change the **Build Action** for the sample data file to**DesignDataWithDesignTimeCreatableTypes**.

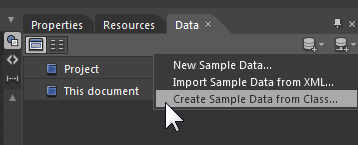
|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| In the next illustration, the **Custom Tool** property is empty. This is required for sample data to work correctly. By default, Expression Blend correctly sets this property to empty. When you use Visual Studio 2010 to add a sample data file, you typically add a new resource dictionary item and edit from there. In this case, you must set the **Build Action** and clear the **Custom Tool** property. |

**Sample data file properties**



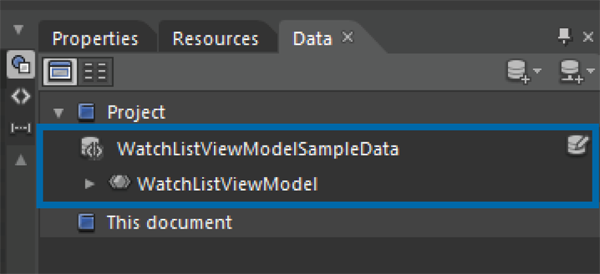
Expression Blend provides tooling for quickly creating and binding XAML sample data. The XAML sample data can be used and viewed in the Visual Studio 2010 designer, as shown in the following illustration.

**Defining sample data in Expression Blend 4**



After it generates the sample data, the data will appear in the Data pane, as shown in the following illustration.

**Data pane**



You can then drag it onto the root element of the view, such as the **UserControl**, and have it set the **d:DataContext** property. You can also drop sample data collections onto items controls, and Blend will wire up the sample data to the control.

|  |
| --- |
| **f921098.note(en-us,PandP.40).gif注意:** |
| XAML sample data files are not compiled into or included in built assemblies. |

#### XAML Resource

You can create a resource in XAML that instantiates the desired types, and then bind that resource to a **DataContext** or list control.

This technique can be used to quickly create throw-away sample data that is used for editing a data template that would take longer to edit without the sample data.

#### Code

If you prefer creating sample data in code, you can write a class that exposes properties or methods that return sample data to their consumer. For example, you could write a **Customers** class that in its default empty constructor populated itself with multiple instances of the **Customer** class. Each of the **Customer** instances would have the appropriate property values set also.

One technique that you can use to consume the sample data class described previously is to use the **d:DataContext**, **d:DesignInstance**combination, ensuring that you set the **d:DesignInstanceIsDesignTimeCreatable** property to **True**. The reason **IsDesignTimeCreatable**must be **True**is that you want the customers constructor to be executed so that the code to populate the class will run. If customers is treated as a faux type, the customers code will never be run and only the "shape" will be discoverable by the tooling.

The following XAML example instantiate the **Customers** class, and then sets it as the **d:DataContext**. Child controls of this **Grid** can consume data exposed by the **Customers** class.

**XAML**

<Grid d:DataContext="{d:DesignInstance local:Customers, IsDesignTimeCreatable=True}">

# UI Layout Key Decisions

When you begin a composite application project, there are some UI design decisions that you need to make that will be difficult to change later. Generally, these decisions are application-wide and their consistency helps developers and designer productivity.

The following are the important UI layout decisions:

* Decide on application flow and define regions accordingly.
* Decide which type of view loading each region will use.
* Decide if you want to use the Region Navigation APIs.
* Decide which UI Design pattern you will use (MVVM, presentation model, and so on).
* Decide on a sample data strategy.

# 更多

For 更多 about extending the Prism Library, see "[Extending Prism](http://msdn.microsoft.com/en-us/library/gg430866(v=pandp.40))."

For 更多 about commands, see "[Commands](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#Commands)" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

For 更多 about data binding, see "[Data Binding](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40)#DataBinding)" in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

For 更多 about region navigation, see Chapter 8, "[Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40))."

For 更多 about the guidelines discussed in this chapter, see the following:

* "Dependency Properties Overview" on MSDN:
* <http://msdn.microsoft.com/en-us/library/ms752914.aspx>.
* Data binding; see:
  + "Data Binding Overview" on MSDN:  
    <http://msdn.microsoft.com/en-us/library/ms742521.aspx>.
  + "Data Binding in WPF" in MSDN Magazine:  
    <http://msdn.microsoft.com/en-us/magazine/cc163299.aspx>.
* "Data Templating Overview" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/ms742521.aspx>.
* "Resources Overview" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/ms750613.aspx>.
* "UserControl Class" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/system.windows.forms.usercontrol.aspx>.
* "VisualStateManager Class" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/cc626338(v=VS.95).aspx>.
* "Customizing Controls For Windows Presentation Foundation" in MSDN Magazine:  
  <http://msdn.microsoft.com/en-us/magazine/cc163421.aspx>.
* "ComponentResourceKey Markup Extension" MSDN topic:  
  <http://msdn.microsoft.com/en-us/library/ms753186.aspx>.
* "Design-Time Attributes in the WPF Designer" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/ee839627.aspx>.
* "Design-Time Attributes in the Silverlight Designer" on MSDN:   
  <http://msdn.microsoft.com/en-us/library/ff602277(VS.95).aspx>.
* "Markup Extensions and WPF XAML" on MSDN:  
  <http://msdn.microsoft.com/en-us/library/ms747254.aspx>.
* "Using Custom Fonts in Silverlight":  
  <http://silverlight.net/learn/learnvideo.aspx?video=69800>.
* "Sample Data in the WPF and Silverlight Designer" on MSDN:  
  <http://blogs.msdn.com/b/wpfsldesigner/archive/2010/06/30/sample-data-in-the-wpf-and-silverlight-designer.aspx>.
* Learning the Visual Studio WPF and Silverlight Designer

This contains tutorials and articles on layout, resources, data binding, sample data, debugging data bindings, object data sources, and master-detail forms.

* <http://blogs.msdn.com/b/wpfsldesigner/archive/2010/01/15/learn.aspx>.

# 8: 导航

|  |
| --- |
| **索引:** |
| [**Prism中的导航**](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec1) | [**基于状态的导航**](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec2) – [*在不同格式或风格下显示数据*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec3), [*反映应用的状态*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec4),[*与用户交互*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec5) | [**基于View的导航**](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec6) – [*Prism的Region*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec7)*概述*, [*Region导航基础*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec8), [*在导航过程中View和View Model的角色*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec9), [*在导航过程中传递参数*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec10), [*迁移到既存View*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec11)*的导航*, [*在导航过程中确认或中止*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec12), [*使用导航日志*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec13), [*使用WPF和Silverlight的导航框架*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec14), [*Region导航队列*](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec15) | [**更多**](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#sec16) |

As the user interacts with a rich client application, its user interface (UI) will be continuously updated to reflect the current task and data that the user is working on. The UI may undergo considerable changes over time as the user interacts with and completes various tasks within the application. The process by which the application coordinates these UI changes is often referred to as navigation.

Frequently, navigation means that certain controls in the UI are removed, while other controls are added. In other cases, navigation may mean that the visual state of one or more existing controls is updated—for example, some controls may be simply hidden or collapsed, while other controls are shown or expanded. Similarly, navigation may mean that the data being displayed by a control is updated to reflect the current state of the application—for example, in a master-detail scenario, the data displayed in the detail view will be updated based on the currently selected item in the master view. All of these scenarios can be considered navigation because the user interface is updated to reflect the user's current task and the application's current state.

Navigation within an application can result from the user's interaction with the UI (via mouse events or other UI gestures) or from the application itself as a result of internal logic-driven state changes. In some cases, navigation may involve very simple UI updates that require no custom application logic. In other cases, the application may implement complex logic to programmatically control navigation to ensure that certain business rules are enforced—for example, the application may not allow the user to navigate away from a certain form without first ensuring that the data entered is correct.

Implementing the required navigation behavior in a Windows Presentation Foundation (WPF) or Silverlight application can often be relatively straightforward because they both provide direct support for navigation. However, navigation can be more complex to implement in applications that use the Model-View-ViewModel (MVVM) pattern or in composite applications that use multiple loosely-coupled modules. Prism provides guidance on implementing navigation in these situations.

# Navigation in Prism

Navigation is defined as the process by which the application coordinates changes to its UI as a result of the user's interaction with the application or internal application state changes.

UI updates can be accomplished by adding or removing elements from the application's visual tree, or by applying state changes to existing elements within the visual tree. WPF and Silverlight are very flexible platforms, and it is often possible to implement a particular navigation scenario using either of these two approaches. However, the approach that will be most appropriate for your application depends on multiple factors.

Prism differentiates between the two styles of navigation described earlier. Navigation accomplished via state changes to existing controls in the visual tree is referred to as state-based navigation. Navigation accomplished via the addition or removal of elements from the visual tree is referred to as view-based navigation. Prism provides guidance on implementing both styles of navigation, focusing on the case where the application is using the Model-View-ViewModel (MVVM) pattern to separate the UI (encapsulated in the view) from the presentation logic and data (encapsulated in the view model).

## State-Based Navigation

In state-based navigation, the view that represents the UI is updated either through state changes in the view model or through the user's interaction within the view itself. In this style of navigation, instead of replacing the view with another view, the view's state is changed. Depending on how the view's state is changed, the updated UI may feel to the user like navigation.

This style of navigation is suitable in the following situations:

* The view needs to display the same data or functionality in different styles or formats.
* The view needs to change its layout or style based on the underlying state of the view model.
* The view needs to initiate limited modal or non-modal interaction with the user within the context of the view.

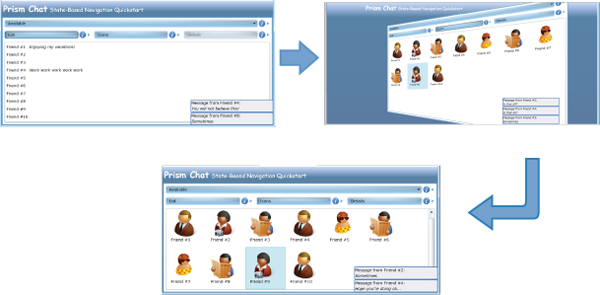
This style of navigation is not suitable for situations in which the UI has to present different data to the user or when the user has to perform a different task. In these situations, it is better to implement separate views (and view models) to represent the data or task, and then to navigate between them using view-based navigation, as described later on in this chapter. Similarly, this style of navigation is not suitable if the number of UI state changes required to implement the navigation are overly complex because the view's definition can become large and difficult to maintain. In this case, it is better to implement the navigation across separate views by using view-based navigation.

The following sections describe the typical situations in which state-based navigation can be used. Each of these sections refers to the State-Based Navigation QuickStart, which implements an instant messaging–style application that allows users to manage and chat with their contacts.

### Displaying Data in Different Formats or Styles

Your application may often need to present the same data to the user, but in different formats or styles. In this case, you can use a state-based navigation within the view to switch between the different styles, potentially using an animated transition between them. For example, the State-Based Navigation QuickStart allows users to choose how their contacts are displayed—either as a simple text list or as avatars (icons). Users can switch between these visual representations by clicking the **List** button or the **Avatars** button. The view provides an animated transition between the two representations, as shown in the following illustration.

**Contact view navigation in the State-Based Navigation QuickStart**



Because the view is presenting the same data, but in a different visual representation, the view model is not required to be involved in the navigation between representations. In this case, navigation is entirely handled within the view itself. This approach provides the UI designer with a lot of flexibility to design a compelling user experience without requiring changes to the application's code.

Microsoft Expression Blend behaviors provide a good way to implement this style of navigation within a view. The State-Based Navigation QuickStart application uses Expression Blend's **DataStateBehavior** data-bound to a radio button to switch between two visual states that are defined using the visual state manager, one button to show the contacts as a list and one button to show the contacts as icons.

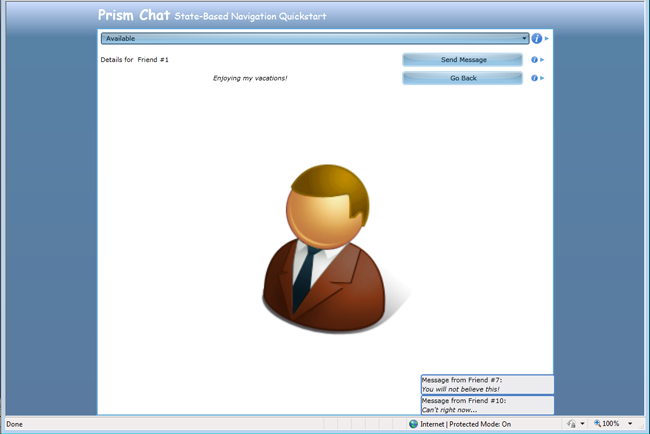
**XAML**

<DataStateBehavior Binding="{Binding IsChecked, ElementName= ShowAsListButton}" TrueState="ShowAsList" FalseState="ShowAsIcons"/>

As the user clicks the **Contacts** or **Avatar** radio buttons, the visual state is toggled between the **ShowAsList** visual state and the**ShowAsIcons** visual state. The flip transition animation between these states is also defined using the visual state manager.

Another example of this style of navigation is shown by the State-Based Navigation QuickStart application when the user switches to the details views for the currently selected contact. The following illustration shows an example of this.

**The Contact Details view in the State-Based Navigation QuickStart**

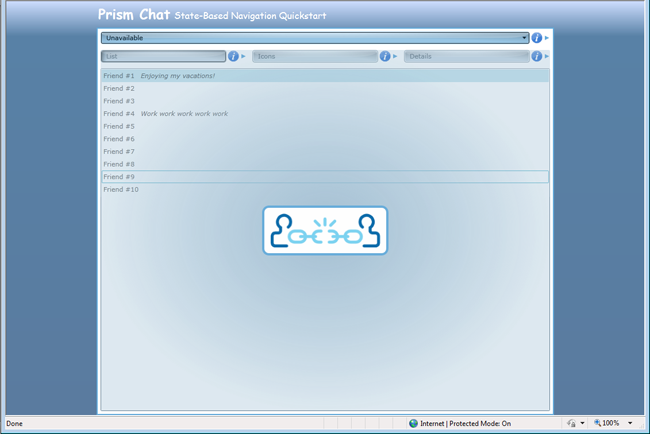


Again, this can be easily implemented using the Expression Blend **DataStateBehavior;** however, this time it is bound to the **ShowDetails**property on the view model, which toggles between the **ShowDetails** and **ShowContacts** visual states using a flip transition animation.

### Reflecting Application State

Similarly, the view within an application may sometimes need to change its layout or style based on changes to an internal application state, which in turn is represented by a property on a view model. An example of this scenario is shown in the State-Based Navigation QuickStart where the user's connection status is represented on the Chat view model class using a **ConnectionStatus** property. As the user's connection status changes, the view is informed (via a property change notification event) allowing the view to visually represent the current connection state appropriately, as shown in the following illustration.

**Connection state representation in the State-Based Navigation QuickStart**



To implement this, the view defines a **DataStateBehavior** data bound to the view model's **ConnectionStatus** property to toggle between the appropriate visual states.

**XAML**

<DataStateBehavior Binding="{Binding ConnectionStatus}" TrueState="Available" FalseState="Unavailable"/>

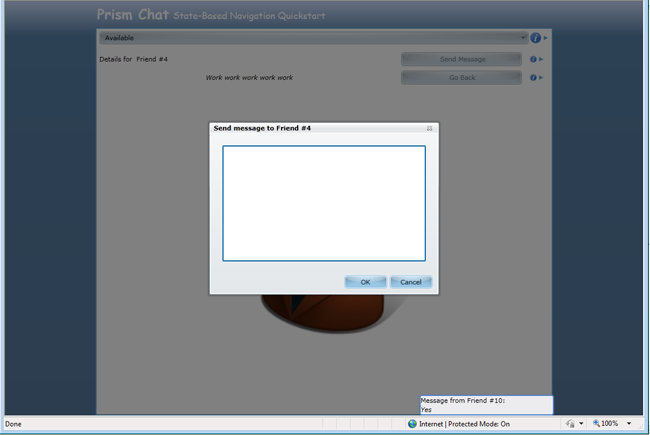
注意: that the connection state can be changed by the user via the UI or by the application according to some internal logic or event. For example, the application may move to an "unavailable" state if the user does not interact with the view within a certain time period or when the user's calendar indicates that he or she is in a meeting. The State-Based Navigation QuickStart simulates this scenario by switching the connection status randomly using a timer. When the connection status is changed, the property on the view model is updated, and the view is informed via a property changed event. The UI is then updated to reflect the current connection status.

All the preceding examples involve defining visual states in the view and switching between them as a result of the user's interaction with the view or via changes in properties defined by the view model. This approach allows the UI designer to implemenent navigation-like visual behavior in the view without requiring the view to be replaced or requiring any code changes to the application's code. This approach is suitable when the view is required to render the same data in different styles or layouts. It is not suitable for situations in which the user is to be presented with different data or application functionality or when navigating to a different part of the application.

### Interacting With the User

Frequently, an application will need to interact with the user in a limited way. In these situations, it is often more appropriate to interact with the user within the context of the current view, instead of navigating to a new view. For example, in the State-Based Navigation QuickStart, the user is able to send a message to a contact by clicking the **Send Message** button. The view then displays a pop-up window that allows the user to type the message, as shown in the following illustration. Because this interaction with the user is limited and logically takes place within the context of the parent view, it can be easily implemented as state-based navigation.

**Interacting with the user using a pop-up window in the State-Based Navigation QuickStart**



To implement this behavior, the State-Based Navigation QuickStart implements a **SendMessage** command, which is bound to the **Send Message** button. When this command is invoked, the view model interacts with the view to display the pop-up window. This is achieved using the Interaction Request pattern described in Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))."

The following code example shows how the view in the State-Based Navigation QuickStart application responds to the**SendMessageRequest** interaction request object provided by the view model. When the request event is received, the**SendMessageChildWindow** is displayed as a popup window.

**XAML**

<prism:InteractionRequestTrigger SourceObject="{Binding SendMessageRequest}"> <prism:PopupChildWindowAction> <prism:PopupChildWindowAction.ChildWindow> <vs:SendMessageChildWindow /> </prism:PopupChildWindowAction.ChildWindow> </prism:PopupChildWindowAction> </prism:InteractionRequestTrigger>

## View-Based Navigation

Although state-based navigation can be useful for the scenarios outlined earlier, navigation within an application will most often be accomplished by replacing one view within the application's UI with another. In Prism, this style of navigation is referred to as view-based navigation.

Depending on the requirements of the application, this process can be fairly complex and require careful coordination. The following are common challenges that often have to be addressed when implementing view-based navigation:

* The target of the navigation—the container or host control of the views to be added or removed—may handle navigation differently as views are added or removed from it, or they may visually represent navigation in different ways. In many cases, the navigation target will be a simple **Frame** or **ContentControl**, and navigated views will simply be displayed within these controls. However, there are many scenarios where the target for the navigation operation is a different type of container control, such as a **TabControl** or a**ListBox** control. In these cases, navigation may require the activation or selection of an existing view or the addition of new view is a specific way.
* The application will also often have to define how the view to be navigated to is identified. For example, in a web application, the page to be navigated to is often directly identified by a Uniform Resource Identifier (URI). In a client application, the view can be identified by type name, resource location, or in a variety of different ways. Furthermore, in a composite application, which is composed from loosely coupled modules, the views will often be defined in separate modules. Individual views will need to be identified in a way that does not introduce tight coupling and dependencies between modules.
* After the view is identified, the process by which the new view is instantiated and initialized has to be carefully coordinated. This can be particularly important when using the MVVM pattern. In this case, the view and view model may need to be instantiated and associated with each other via the view's data context during the navigation operation. In the case when the application is leveraging a dependency injection container, such as the Unity Application Block (Unity) or the Managed Extensibility Framework (MEF), the instantiation of the views and/or view models (and other dependent classes) may have to be achieved using a specific construction mechanism.
* The MVVM pattern provides a separation between the application's UI and its presentation and business logic. However, the navigational behavior of the application will often span UI and presentation logic parts of the application. The user will often initiate navigation from within the view, and the view will be updated as a result of that navigation, but navigation will often also need to be initiated or coordinated from within the view model. The ability to cleanly separate the navigational behavior of the application across the view and view model is an important aspect to consider.
* An application will also often need to pass parameters or context to the view so that it can be initialized properly. For example, if the user navigates to a view to update the details of a specific customer, the customer's ID or data will have to be passed to the view so that it can display the correct information.
* Many applications will also have to carefully coordinate navigation to ensure that certain business rules are obeyed. For example, users may be prompted before navigating away from a view so that they can correct any invalid data or be prompted to submit or discard any data changes that they have made within that view. This process requires careful coordination between the previous view and the new view.
* Lastly, most modern applications allow the user to easily navigate backward (or forward) to previously displayed views. Similarly, some applications implement their workflows using a sequence of views or forms and allow users to navigate forward or backward through them, adding or updating data as they go, before completing the task and submitting all their changes at one time. These scenarios require some kind of journaling (or history) mechanism so that the sequence of navigation can be stored, replayed, or pre-defined.

Prism provides support and guidance for these challenges by extending Prism's region mechanism to support navigation. The following sections provide a brief summary of Prism regions and describe how they have been extended to support view-based navigation.

### Prism Region Overview

Prism regions are designed to support the development of composite applications (that is, applications that consist of multiple modules) by allowing the application's overall UI to be constructed in a loosely-coupled way. Regions allow views defined in a module to be displayed within the application's UI without requiring the module to have explicit knowledge of the application's overall UI structure. They allow the layout of the application's UI to be changed easily, thereby allowing the UI designer to choose the most appropriate UI design and layout for the application without requiring changes in the modules themselves.

Prism regions are essentially named placeholders within which views can be displayed. Any control in the application's UI can be a declared a region by simply adding a **RegionName** attached property to it, as shown here.

**XAML**

<ContentControl prism:RegionManager.RegionName="MainRegion" ... />

For each control specified as a region, Prism creates a **Region** object to represent the region and a **RegionAdapter** object, which manages the placement and activation of views into the specified control. The Prism Library provides **RegionAdapter** implementations for most of the common Silverlight and WPF controls. You can create a custom **RegionAdapter** to support additional controls or when you need to define a custom behavior. The **RegionManager** class provides access to the **Region** objects within the application.

In many cases, the region control will be a simple control, such as a **ContentControl**, that can display one view at a time. In other cases, the **Region** control will be a control that is able to display multiple views at the same time, such as a **TabControl** or a **ListBox** control.

The region adapter manages a list of views within the associated region. One or more of these views will be displayed in the region control according to its defined layout strategy. Views can be assigned a name that can be used to retrieve that view later on. The region adaptermanages the active state of the views within the region. The active view is the view that is the selected or top-most view—for example, in a**TabControl**, the active view is the one displayed in the selected tab; in a **ContentControl**, the active view is the view that is currently displayed as the control's content.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| The active state of a view is important to consider during navigation. Frequently, you will want the active view to participate in navigation so that it can save data before the user navigates away from it, or so that it can confirm or cancel the navigation operation. |

Previous versions of Prism allowed views to be displayed in a region in two ways. The first, called view injection, allows views to be programmatically displayed in a region. This approach is useful for dynamic content, where the view to be displayed in the region changes frequently, according to the application's presentation logic.

View injection is supported through the **Add** method on the **Region** class. The follow code example shows how you can obtain a reference to a **Region** object via the **RegionManager** class and programmatically add a view to it. In this example, the view is created using a dependency injection container.

**C#**

IRegionManager regionManager = ...; IRegion mainRegion = regionManager.Regions["MainRegion"]; InboxView view = this.container.Resolve<InboxView>(); mainRegion.Add(view);

The second method, called view discovery, allows a module to register a view type against a region name. Whenever a region with the specified name is displayed, an instance of the specified view will be automatically created and displayed in the region. This approach is useful for relatively static content, where the view to be displayed in a region does not change.

View discovery is supported through the **RegisterViewWithRegion** method on the **RegionManager** class. This method allows you to specify a callback method that will be called when the named region is shown. The following code example shows how you can create a view (via the dependency injection container) when the main region is first shown.

**C#**

IRegionManager regionManager = ...; regionManager.RegisterViewWithRegion("MainRegion", () => container.Resolve<InboxView>());

For a detailed overview of Prisms region support and information about how to leverage regions to compose the application's UI using view injection and discovery, see Chapter 7, "[Composing the User Interface](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))." The rest of this chapter describes how regions have been extended to support view-based navigation, and how this addresses the various challenges described earlier.

### Basic Region Navigation

Both view injection and view discovery can be considered to be limited forms of navigation–view injection is a form of explicit, programmatic navigation, and view discovery is a form of implicit or deferred navigation. However, in Prism 4.0, regions have been extended to support a more general notion of navigation, based on URIs and an extensible navigation mechanism.

Navigation within a region means that a new view is to be displayed within that region. The view to be displayed is identified via a URI, which, by default, refers to the name of the view to be created. You can programmatically initiate navigation using the **RequestNavigate**method defined by the **INavigateAsync**interface.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| Despite its name, the **INavigateAsync** interface does not represent asynchronous navigation that's carried out on a separate background thread. Instead, the **INavigateAsync** interface represents the ability to perform pseudo-asynchronous navigation. The**RequestNavigate** method may return synchronously following the completion of navigation operation, or it may return while a navigation operation is still pending, as in the case where the user needs to confirm the navigation. By allowing you to specify callbacks and continuations during navigation, Prism provides a mechanism to enable these scenarios without requiring the complexity of navigating on a background thread. |

The **INavigateAsync** interface is implemented by the **Region** class, allowing you to initiate navigation within that region.

**C#**

IRegion mainRegion = ...; mainRegion.RequestNavigate(new Uri("InboxView", UriKind.Relative));

You can also call the **RequestNavigate** method on the **RegionManager**, which allows you to specify the name of the region to be navigated. This convenient method obtains a reference to the specified region and then calls the **RequestNavigate** method, as shown in the preceding code example.

**C#**

IRegionManager regionManager = ...; regionManager.RequestNavigate("MainRegion", new Uri("InboxView", UriKind.Relative));

By default, the navigation URI specifies the name of a view that is registered in the container. The code below illustrates the relationship of the view registration name in the container and the use of the name when navigating the view when using the Unity container.

**C#**

container.RegisterType<object, InboxView>("InboxView"); regionManager.Regions[Constants.MainRegion].RequestNavigate(new Uri("InboxView", UriKind.Relative));

|  |
| --- |
| **g430861.note(en-us,PandP.40).gifImportant:** |
| When the region navigation service creates a view, it requests a type of **Object** from the container with a name that matches the one supplied in the navigation URI. Various containers provide different registration mechanisms to support this. For instance, in Unity you will need to register your views against the **Object** type, map it to your view type, and provide a name that matches the one used in the navigation URI. In MEF, it is implemented differently and only the contract name is used. Therefore, as long as the view is exported with a contract name that matches the name in the URI request, the view can be successfully constructed. |

Example: When using Unity to register your view for navigation:

**Do not use:**

container.RegisterType<InboxView>("InboxView");

**Instead you must use:**

container.RegisterType<object,InboxView>("InboxView");

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| The name used to register the view in the container and to navigate does not have to be associated with the type name; any string will suffice. For example, you can use the full type name instead of a quoted string:  typeof(InboxView).FullName |

Using MEF, you can simply export the view type with the specified name.

**C#**

[Export("InboxView")] public partial class InboxView : UserControl

During navigation, the specified view is instantiated, via the container or MEF, along with its corresponding view model and other dependent services and components. After the view is instantiated, it is then added to the specified region and activated (activation is described in more detail later in this chapter).

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| The preceding description illustrates view-first navigation, where the URI refers to the name of the view type, as it is exported or registered with the container. With view-first navigation, the dependent view model is created as a dependency of the view. An alternative approach is to use view model–first navigation, where the navigation URI refers to the name of the view model type, as it is exported or registered with the container. View model–first navigation is useful when the view is defined as a data template, or when you want your navigation scheme to be defined independently of the views. |

The **RequestNavigate** method also allows you to specify a callback method, or a delegate, which will be called when navigation is complete.

**C#**

private void SelectedEmployeeChanged(object sender, EventArgs e) { ... regionManager.RequestNavigate(RegionNames.TabRegion, "EmployeeDetails", NavigationCompleted); } private void NavigationCompleted(NavigationResult result) { ... }

The **NavigationResult** class defines properties that provide information about the navigation operation. The **Result** property indicates whether or not navigation succeeded. If navigation failed, the **Error** property provides a reference to any exception that was thrown during navigation. The **Context** property provides access to the navigation URI and any parameters it contains, and a reference to the navigation service that coordinated the navigation operation.

### View and View Model Participation in Navigation

Frequently, the views and view models in your application will want to participate in navigation. The **INavigationAware** interface enables this. You can implement this interface on the view or (more commonly) the view model. By implementing this interface, your view or view model can opt-in to participate in the navigation process.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| In the description that follows, although a reference is made to calls to this interface during navigation between views, it should be 注意:d that the **INavigationAware** interface will be called during navigation whether it is implemented by the view or by the view model. During navigation, Prism checks to see whether the view implements the **INavigationAware** interface; if it does, it calls the required methods during navigation. Prism also checks to see whether the object set as the view's **DataContext** implements this interface; if it does, it calls the required methods during navigation. |

This interface allows the view or view model to participate in a navigation operation. The **INavigationAware** interface defines three methods.

**C#**

public interface INavigationAware { bool IsNavigationTarget(NavigationContext navigationContext); void OnNavigatedTo(NavigationContext navigationContext); void OnNavigatedFrom(NavigationContext navigationContext); }

The **IsNavigationTarget** method allows an existing (displayed) view or view model to indicate whether it is able to handle the navigation request. This is useful in cases where you can re-use an existing view to handle the navigation operation or when navigating to a view that already exists. For example, a view displaying customer information can be updated to display a different customer's information. For 更多 about using this method, see the section, "[Navigating to Existing Views](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#NavigatingtoExistingViews)," later in this chapter.

The **OnNavigatedFrom** and **OnNavigatedTo** methods are called during a navigation operation. If the currently active view in the region implements this interface (or its view model), its **OnNavigatedFrom** method is called before navigation takes place. The**OnNavigatedFrom** method allows the previous view to save any state or to prepare for its deactivation or removal from the UI, for example, to save any changes that the user has made to a web service or database.

If the newly created view implements this interface (or its view model), its **OnNavigatedTo** method is called after navigation is complete. The **OnNavigatedTo** method allows the newly displayed view to initialize itself, potentially using any parameters passed to it on the navigation URI. For 更多, see the next section, "[Passing Parameters During Navigation](http://msdn.microsoft.com/en-us/library/gg430861(v=pandp.40)#PassingParametersDuringNavigation)."

After the new view is instantiated, initialized, and added to the target region, it then becomes the active view, and the previous view is deactivated. Sometimes you will want the deactivated view to be removed from the region. Prism provides the **IRegionMemberLifetime**interface, which allows you to control the lifetime of views within regions by allowing you to specify whether deactivated views are to be removed from the region or simply marked as deactivated.

**C#**

public class EmployeeDetailsViewModel : IRegionMemberLifetime { public bool KeepAlive { get { return true; } } }

The **IRegionMemberLifetime** interface defines a single read-only property, **KeepAlive**. If this property returns **false**, the view is removed from the region when it is deactivated. Because the region no longer has a reference to the view, it then becomes eligible for garbage collection (unless some other component in your application maintains a reference to it). You can implement this interface on your view or your view model classes. Although the **IRegionMemberLifetime** interface is primarily intended to allow you to manage the lifetime of views within regions during activation and deactivation, the **KeepAlive** property is also considered during navigation after the new view is activated in the target region.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| Regions that can display multiple views, such as those that use an **ItemsControl** or a **TabControl**, will display both non-active and active views. Removal of a non-active view from these types of regions will result in the view being removed from the UI. |

### Passing Parameters During Navigation

To implement the required navigational behavior in your application, you will often need to specify additional data during navigation request than just the target view name. The **NavigationContext** object provides access to the navigation URI, and to any parameters that were specified within it. You can access the **NavigationContext** from within the **IsNavigationTarget**, **OnNavigatedFrom**, and **OnNavigatedTo**methods.

Prism provides the **UriQuery** class to help specify and retrieve navigation parameters. You can use this class to add navigation parameters to the navigation URI before you initiate navigation and to access individual parameters during navigation. The **UriQuery** class maintains a list of name-value pairs, one for each parameter.

The following code example shows how to add individual parameters to the **UriQuery** instance so that it can be appended to the navigation URI.

**C#**

Employee employee = Employees.CurrentItem as Employee; if (employee != null) { UriQuery query = new UriQuery(); query.Add("ID", employee.Id); \_regionManager.RequestNavigate(RegionNames.TabRegion, new Uri("EmployeeDetailsView" + query.ToString(), UriKind.Relative)); }

You can retrieve the navigation parameters using the **Parameters** property on the **NavigationContext** object. This property returns an instance of the **UriQuery** class, which provides an indexer property to allow easy access to individual parameters.

**C#**

public void OnNavigatedTo(NavigationContext navigationContext) { string id = navigationContext.Parameters["ID"]; }

### Navigating to Existing Views

Frequently, it is more appropriate for the views in your application to be re-used, updated, or activated during navigation, instead of replaced by a new view. This is often the case where you are navigating to the same type of view but need to display different information or state to the user, or when the appropriate view is already available in the UI but needs to be activated (that is, selected or made top-most).

For an example of the first scenario, imagine that your application allows the user to edit customer records, using the **EditCustomer** view, and the user is currently using that view to edit customer ID 123. If the customer decides to edit the customer record for customer ID 456, the user can simply navigate to the **EditCustomer** view and enter the new customer ID. The **EditCustomer** view can then retrieve the data for the new customer and update its UI accordingly.

An example of the second scenario is where the application allows the user to edit more than one customer record at a time. In this case, the application displays multiple **EditCustomer** view instances in a tab control—for example, one for customer ID 123 and another for customer ID 456. When the user navigates to the **EditCustomer** view and enters customer ID 456, the corresponding view will be activated (that is, its corresponding tab will be selected). If the user navigates to the **EditCustomer** view and enters customer ID 789, a new instance will be created and displayed in the tab control.

The ability to navigate to an existing view is useful for a variety of reasons. It is often more efficient to update an existing view instead of replace it with a new instance of the same type. Similarly, activating an existing view, instead of creating a duplicate view, provides a more consistent user experience. In addition, the ability to handle these situations seamlessly without requiring much custom code means that the application is easier to develop and maintain.

Prism supports the two scenarios described earlier via the **IsNavigationTarget** method on the **INavigationAware** interface. This method is called during navigation on all views in a region that are of the same type as the target view. In the preceding examples, the target type of the view is the **EditCustomer** view, so the **IsNavigationTarget** method will be called on all existing **EditCustomer** view instances currently in the region. Prism determines the target type from the view URI, which it assumes is the short type name of the target type.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| For Prism to determine the type of the target view, the view's name in the navigation URI should be the same as the actual target type's short type name. For example, if your view is implemented by the **MyApp.Views.EmployeeDetailsView** class, the view name specified in the navigation URI should be **EmployeeDetailsView**. This is the default behavior provided by Prism. You can customize this behavior by implementing a custom content loader class; you can do this by implementing the **IRegionNavigationContentLoader**interface or by deriving from the **RegionNavigationContentLoader** class. |

The implementation of the **IsNavigationTarget** method can use the **NavigationContext** parameter to determine whether it can handle the navigation request. The **NavigationContext** object provides access to the navigation URI and the navigation parameters. In the preceding examples, the implementation of this method in the **EditCustomer** view model compares the current customer ID to the ID specified in the navigation request, and it returns **true** if they match.

**C#**

public bool IsNavigationTarget(NavigationContext navigationContext) { string id = navigationContext.Parameters["ID"]; return \_currentCustomer.Id.Equals(id); }

If the **IsNavigationTarget** method always returns **true**, regardless of the navigation parameters, that view instance will always be re-used. This allows you to ensure that only one view of a particular type will be displayed in a particular region.

### Confirming or Cancelling Navigation

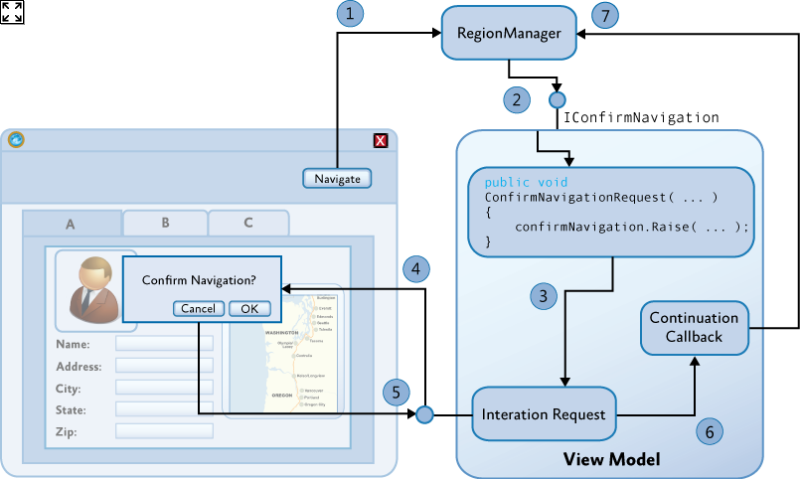
You will often find that you will need to interact with the user during a navigation operation, so that the user can confirm or cancel it. In many applications, for example, the user may try to navigate while in the middle of entering or editing data. In these situations, you may want to ask the user whether he or she wants to save or discard the data that has been entered before continuing to navigate away from the page, or whether the user wants to cancel the navigation operation altogether. Prism supports these scenarios via the**IConfirmNavigationRequest** interface.

The **IConfirmNavigationRequest** interface derives from the **INavigationAware** interface and adds the **ConfirmNavigationRequest**method. By implementing this interface on your view or view model class, you allow them to participate in the navigation sequence in a way that allows them to interact with the user so that the user can confirm or cancel the navigation. You will often use an **Interaction Request** object, as described in "Using Interaction Request Objects" in Chapter 6, "[Advanced MVVM Scenarios](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))," to display a confirmation pop-up window.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| The **ConfirmNavigationRequest** method is called on the active view or view model, similar to the **OnNavigatedFrom** method described earlier. |

The **ConfirmNavigationRequest** method provides two parameters, a reference to the current navigation context as described earlier, and a callback method that you can call when you want navigation to continue. For this reason, the callback is known as a continuation callback. You can store a reference to the continuation callback so the application can call it after it finishes interacting with the user. If your application interacts with the user through an **Interaction Request** object, you can chain the call to the continuation callback to the callback from the interaction request. The following diagram illustrates the overall process.

**Confirming Navigation Using an InteractionRequest Object**

[](http://msdn.microsoft.com/en-us/library/Gg430861.430462631ab962dccbce65c298a2eed8(l=en-us,v=pandp.40).png)

The following steps summarize the process of confirming navigation using an **InteractionRequest** object:

1. Navigation operation is initiated via a **RequestNavigate** call.
2. If the view or view model implements **IConfirmNavigation**, call **ConfirmNavigationRequest**.
3. The view model raises the interaction request event.
4. The view displays the confirmation pop-up window and awaits the user's response.
5. The interaction request callback is invoked when the user closes the pop-up window.
6. Continuation callback is invoked to continue or cancel the pending navigation operation.
7. The navigation operation is completed or canceled.

To illustrate this, look at the View-Switching Navigation Quick Start. This application provides the ability for the user to compose a new email using the **ComposeEmailView** and **ComposeEmailViewModel** classes. The view model class implements the **IConfirmNavigation**interface. If the user navigates, such as by clicking the **Calendar** button, when they are composing an email, the**ConfirmNavigationRequest** method will be called so that the view model can confirm the navigation with the user. To support this, the view model class defines an interaction request, as shown in the following code example.

**C#**

public class ComposeEmailViewModel : NotificationObject, IConfirmNavigationRequest { private readonly InteractionRequest<Confirmation> confirmExitInteractionRequest; public ComposeEmailViewModel(IEmailService emailService) { this.confirmExitInteractionRequest = new InteractionRequest<Confirmation>(); } public IInteractionRequest ConfirmExitInteractionRequest { get { return this.confirmExitInteractionRequest; } } }

In the **ComposeEmailVew** class, an interaction request trigger is defined, and data is bound to the **ConfirmExitInteractionRequest**property on the view model. When the interaction request is made, a simple pop-up window will be displayed to the user.

**XAML**

<UserControl.Resources> <DataTemplate x:Name="ConfirmExitDialogTemplate"> <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center" Text="{Binding}"/> </DataTemplate> </UserControl.Resources> <Grid x:Name="LayoutRoot" Background="White"> <ei:Interaction.Triggers> <prism:InteractionRequestTrigger SourceObject="{Binding ConfirmExitInteractionRequest}"> <prism:PopupChildWindowAction ContentTemplate="{StaticResource ConfirmExitDialogTemplate}"/> </prism:InteractionRequestTrigger> </ei:Interaction.Triggers> ...

The ConfirmNavigationRequest method on the ComposeEmailVewModel class is called if the user attempts to navigate while an email is being composed. The implementation of this method invokes the interaction request defined earlier so that the user can confirm or cancel the navigation operation.

pandpDocTools-topicTitle: 8: Navigation

errorText: The ConfirmNavigationRequest method on the ComposeEmailVewModel class is called if the user attempts to navigate while an email is being composed. The implementation of this method invokes the interaction request defined earlier so that the user can confirm or cancel the navigation operation.

**C#**

void IConfirmNavigationRequest.ConfirmNavigationRequest( NavigationContext navigationContext, Action<bool> continuationCallback) { this.confirmExitInteractionRequest.Raise( new Confirmation {Content = "...", Title = "..."}, c => {continuationCallback(c.Confirmed);}); }

The callback for the interaction request is called when the user clicks the buttons in the confirmation pop-up window to confirm or cancel the operation. This callback simply calls the continuation callback, passing in the value of the **Confirmed** flag, and causing the navigation to continue or be canceled.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| It should be 注意:d that after the interaction request event is raised, the **ConfirmNavigationRequest** method immediately returns so that the user can continue to interact with the UI of the application. When the user clicks the **OK** or **Cancel** buttons on the pop-up window, the callback method of the interaction request is made, which in turn calls the continuation callback to complete the navigation operation. All the methods are called on the UI thread. Using this technique, no background threads are required. |

Using this mechanism, you can control if the navigation request is carried out immediately or is deferred, pending an interaction with the user or some other asynchronous interaction (for example, as a result of a web service request). To enable navigation to proceed, you can simply call the continuation callback method, passing **true** to indicate that it can continue. Similarly, you can pass **false** to indicate that the navigation should be canceled.

**C#**

void IConfirmNavigationRequest.ConfirmNavigationRequest( NavigationContext navigationContext, Action<bool> continuationCallback) { continuationCallback(true); }

If you want to defer navigation, you can store a reference to the continuation callback you can then call when the interaction with the user (or web service) completes. The navigation operation will be pending until you call the continuation callback.

If the user initiates another navigation operation in the meantime, the navigation request then becomes canceled. In this case, calling the continuation callback has no effect because the navigation operation to which it relates is no longer current. Similarly, if you decide not to call the continuation callback, the navigation operation will be pending until it is replaced with a new navigation operation.

### Using the Navigation Journal

The **NavigationContext** class provides access to the region navigation service, which is responsible for coordinating the sequence of operations during navigation within a region. It provides access to the region in which navigation is taking place, and to the navigation journal associated with that region. The region navigation service implements the **IRegionNavigationService**, which is defined as follows.

**C#**

public interface IRegionNavigationService : INavigateAsync { IRegion Region {get; set;} IRegionNavigationJournal Journal {get;} event EventHandler<RegionNavigationEventArgs> Navigating; event EventHandler<RegionNavigationEventArgs> Navigated; event EventHandler<RegionNavigationFailedEventArgs> NavigationFailed; }

Because the region navigation service implements the **INavigateAsync** interface, you can initiate navigation within the parent region by calling its **RequestNavigate** method. The **Navigating** event is raised when a navigation operation is initiated. The **Navigated** event is raised when navigation within a region is completed. The **NavigationFailed** is raised if an error was encountered during navigation.

The **Journal** property provides access to the navigation journal associated with the region. The navigation journal implements the**IRegionNavigationJournal** interface, which is defined as follows.

**C#**

public interface IRegionNavigationJournal { bool CanGoBack { get; } bool CanGoForward { get; } IRegionNavigationJournalEntry CurrentEntry { get; } INavigateAsync NavigationTarget { get; set; } void Clear(); void GoBack(); void GoForward(); void RecordNavigation(IRegionNavigationJournalEntry entry); }

You can obtain and store a reference to the region navigation service within a view during navigation via the **OnNavigatedTo** method call. By default, Prism provides a simple stack-based journal that allows you to navigate forward or backward within a region.

You can use the navigation journal to allow the user to navigate from within the view itself. In the following example, the view model implements a **GoBack** command, which uses the navigation journal within the host region. Therefore, the view can display a **Back** button that allows the user to easily navigate back to the previous view within the region. Similarly, you can implement a **GoForward** command to implement a wizard style workflow.

**C#**

public class EmployeeDetailsViewModel : INavigationAware { ... private IRegionNavigationService navigationService; public void OnNavigatedTo(NavigationContext navigationContext) { navigationService = navigationContext.NavigationService; } public DelegateCommand<object> GoBackCommand { get; private set; } private void GoBack(object commandArg) { if (navigationService.Journal.CanGoBack) { navigationService.Journal.GoBack(); } } private bool CanGoBack(object commandArg) { return navigationService.Journal.CanGoBack; } }

You can implement a custom journal for a region if you need to implement a specific workflow pattern within that region.

|  |
| --- |
| **g430861.note(en-us,PandP.40).gif注意:** |
| The navigation journal can only be used for region-based navigation operations that are coordinated by the region navigation service. If you use view discovery or view injection to implement navigation within a region, the navigation journal will not be updated during navigation and cannot be used to navigate forward or backward within that region. |

### Using the WPF and Silverlight Navigation Frameworks

Prism region navigation was designed to address a wide range of common scenarios and challenges that you may face when implementing navigation in a loosely-coupled, modular application that uses the MVVM pattern and a dependency injection container, such as Unity, or the Managed Extensibility Framework (MEF). It also was designed to support navigation confirmation and cancelation, navigation to existing views, navigation parameters and navigation journaling.

By supporting navigation within Prism regions, it also supports navigation within a wide range of layout controls and supports the ability to change the layout of the application's UI without affecting its navigation structure. It also supports pseudo-synchronous navigation, which allows for rich user interaction during navigation.

However, the Prism region navigation was not designed to replace the Silverlight navigation framework (introduced in Silverlight 3.0) or WPF's navigation framework. Instead, Prism region navigation was designed to be used side-by-side with the Silverlight and WPF navigation frameworks.

The Silverlight navigation framework provides support for deep linking, browser integration, and navigation URI mapping. Navigation takes place with a **Frame** control. The **Frame** can optionally display a navigation bar that allows the user to navigate forward or backward through the views displayed in the **Frame**. It is common to use the Silverlight navigation framework to implement top-level navigation within the application's shell and then to use Prism regions to implement navigation within the rest of the application. In this way, your application can support deep linking and be integrated with the browser's journal and address bar, but it still takes advantage of Prism region navigation.

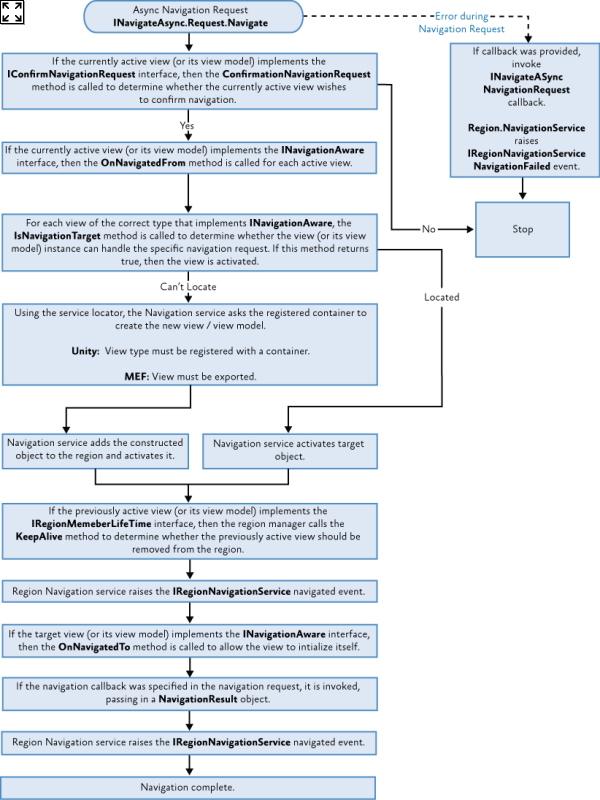
By default, the Silverlight navigation framework does not directly support the use of the MVVM pattern or the user of a dependency injection container or MEF. However, you can implement a custom content loader—the component used by the frame to load the content associated with a specific URI—which can instantiate and initialize the view and its associated view model as appropriate and display it in the **Frame**. You can also implement a custom content loader to more fully integrate Silverlight's navigation framework with the Prism region navigation mechanisms.

The WPF navigation framework is not as extensible as the one in Silverlight; therefore, it is difficult for it to support the MVVM pattern and dependency injection. It is also based on a **Frame** control that provides similar functionality in terms of journaling and navigation UI. You can use the WPF navigation framework alongside Prism region navigation, though it may be easier and more flexible to implement navigation using only Prism regions.

### The Region Navigation Sequence

The following illustration provides an overview of the sequence of operations during a navigation operation. It is provided for reference so that you can see how the various elements of the Prism region navigation work together during a navigation request.

**Prism region navigation sequence**

[](http://msdn.microsoft.com/en-us/library/Gg430861.e8eaa02d7a2d9a1370c3bf6d79636697(l=en-us,v=pandp.40).png)

# 更多

更多关于Prism中Region的详细信息，请查看Chapter 7，"[Composing the User Interface](http://msdn.microsoft.com/en-us/library/ff921098(v=pandp.40))."

For 更多 about the MVVM pattern and Interaction Request pattern, see Chapter 5, "[Implementing the MVVM Pattern](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40))" and Chapter 6, "[Advanced MVVM Scenarios](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))."

For 更多 about the **Interaction Request** object, see "Using Interaction Request Objects" in Chapter 6, "[Advanced MVVM Scenarios](http://msdn.microsoft.com/en-us/library/gg405494(v=pandp.40))."

For 更多 about the Visual State Manager, see "VisualStateManager Class" on MSDN:  
<http://msdn.microsoft.com/en-us/library/cc626338(v=VS.95).aspx>.

For 更多 about using Microsoft Expression Blend behaviors, see "Working with built-in behaviors" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ff724013(v=Expression.40).aspx>.

For 更多 about creating custom behaviors with Microsoft Expression Blend, see "Creating Custom Behaviors" on MSDN:   
<http://msdn.microsoft.com/en-us/library/ff724708(v=Expression.40).aspx>.

For 更多 about the Silverlight Navigation Framework, see "Navigation Overview" on MSDN:  
<http://msdn.microsoft.com/en-us/library/cc838245(VS.95).aspx>.

For 更多 about integrating Silverlight's Navigation Framework with Prism, see "Integrating Prism v4 Region Navigation with Silverlight Frame Navigation" on Karl Schifflett's blog:  
<http://blogs.msdn.com/b/kashiffl/archive/2010/10/05/integrating-prism-v4-region-navigation-with-silverlight-frame-navigation.aspx>.

# 9: 松耦合组件之间的通讯

|  |
| --- |
| **索引:** |
| [**命令解决方案**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec1) | [**创建Delegate 命令**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec2) | [**创建复合命令**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec3) | [**创建全局命令**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec4) | [**绑定全局命令**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec5) | [**Region上下文**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec6) | [**共享服务**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec7) | [**事件聚合**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec8) | [**IEventAggregator**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec9)| [**CompositePresentationEvent**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec10) | [**创建并发布Event**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec11) – [*创建Event*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec12), [*发布Event*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec13) | [**订阅Events**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec14) – [*在UI线程上订阅*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec15), [*订阅过滤*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec16), [*使用强引用订阅*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec17), [*默认订阅*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec18), [*取消Event*](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec19)*订阅* | [**更多**](http://msdn.microsoft.com/en-us/library/ff921122(v=pandp.40)#sec20) |

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, it is important that you know the differences between the approaches so that you can best determine which approach to use in your particular scenario. The Prism Library provides the following communication approaches:

* **Solution commanding**. Use when there is an expectation of immediate action from the user interaction.
* **Region context**. Use this to provide contextual information between the host and views in the host's region. This approach is somewhat similar to the **DataContext**, but it does not rely on it.
* **Shared services**. Callers can call a method on the service which raises an event to the receiver of the message. Use this if none of the preceding is applicable.
* **Event aggregation**. For communication across view models, presenters, or controllers when there is not a direct action-reaction expectation.

# Solution Commanding

If you need to respond to a user gesture, such as clicking on a command invoker (for example, a button or menu item), and if you want the invoker to be enabled based on business logic, use commanding**.**

Windows Presentation Foundation (WPF) provides **RoutedCommand**, which is good at connecting command invokers, such as menu items and buttons, with command handlers that are associated with the current item in the visual tree that has keyboard focus.

However, in a composite scenario, the command handler is often a view model that does not have any associated elements in the visual tree or is not the focused element. To support this scenario, the Prism Library provides **DelegateCommand**, which allows you to call a delegate method when the command is executed, and **CompositeCommand**, which allows you to combine multiple commands**.**These commands are different from the built-in **RoutedCommand**, which will route command execution and handling up and down the visual tree. This allows you to trigger a command at a point in the visual tree and handle it at a higher level.

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

**CompositeCommands** support enablement. **CompositeCommands** listen to the **CanExecuteChanged** event of each one of its connected commands. It then raises this event notifying its invoker(s). The invoker(s) reacts to this event by calling **CanExecute** on the**CompositeCommand**. The **CompositeCommand** then again polls all its child commands by calling **CanExecute** on each child command. If any call to **CanExecute** returns **false**, the **CompositeCommand** will return **false**, thus disabling the invoker(s).

How does this help you with cross module communication? Applications based on the Prism Library may have global **CompositeCommands**that are defined in the shell that have meaning across modules, such as **Save**, **Save All**, and **Cancel**. Modules can then register their local commands with these global commands and participate in their execution.

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| **About WPF Routed Events and Routed Commands**  A routed event is a type of event that can invoke handlers on multiple listeners in an element tree, instead of notifying only the object that directly subscribed to the event. WPF-routed commands deliver command messages through UI elements in the visual tree, but the elements outside the tree will not receive these messages because they only bubble up or down from the focused element or an explicitly stated target element. Routed events can be used to communicate through the element tree, because the event data for the event is perpetuated to each element in the route. One element could change something in the event data, and that change would be available to the next element in the route. Therefore, you should use WPF routed events in the following scenarios: defining common handlers at a common root or defining your own custom control class. |

## Creating a Delegate Command

To create a delegate command, instantiate a **DelegateCommand** field in the constructor of your view model, and then expose it as an**ICommand** property.

**C# ArticleViewModel.cs**

public class ArticleViewModel : NotificationObject { private readonly ICommand showArticleListCommand; public ArticleViewModel(INewsFeedService newsFeedService, IRegionManager regionManager, IEventAggregator eventAggregator) { this.showArticleListCommand = new DelegateCommand(this.ShowArticleList); } public ICommand ShowArticleListCommand { get { return this.showArticleListCommand; } } }

## Creating a Composite Command

To create a composite command, instantiate a **CompositeCommand** field in the constructor, add commands to it, and then expose it as an**ICommand** property.

**C#**

public class MyViewModel : NotificationObject { private readonly CompositeCommand saveAllCommand; public ArticleViewModel(INewsFeedService newsFeedService, IRegionManager regionManager, IEventAggregator eventAggregator) { this.saveAllCommand = new CompositeCommand(); this.saveAllCommand.RegisterCommand(new SaveProductsCommand()); this.saveAllCommand.RegisterCommand(new SaveOrdersCommand()); } public ICommand SaveAllCommand { get { return this.saveAllCommand; } } }

## Making a Command Globally Available

Typically, to create a globally available command, create an instance of the **DelegateCommand**or the **CompositeCommand** and expose it through a static class.

**C#**

public static class GlobalCommands { public static CompositeCommand MyCompositeCommand = new CompositeCommand(); }

In your module, associate child commands to the globally available command.

**C#**

GlobalCommands.MyCompositeCommand.RegisterCommand(command1); GlobalCommands.MyCompositeCommand.RegisterCommand(command2);

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| To increase the testability of your code, you can use a proxy class to access the globally available commands and mock that proxy class in your tests. |

## Binding to a Globally Available Command

The following code example shows how to bind a button to the command in WPF.

**XAML**

<Button Name="MyCompositeCommandButton" Command="{x:Static local:GlobalCommands.MyCompositeCommand}">Execute My Composite Command </Button>

Silverlight does not provide support for **x:static**, so perform the following steps to bind a button to a command in Silverlight:

1. On the view's model, create a public property to obtain the command from the static class. This is shown in the following code example.

**C#**

public ICommand MyCompositeCommand { get { return GlobalCommands.MyCompositeCommand; } }

1. Typically, models are passed to the view through its **DataContext**(this is done at the view's code behind file). The following code example shows how to set the model as the **DataContext** of the view. By doing this, you can declaratively bind the command to a control in your view in the XAML code.

**C#**

view.DataContext = model;

1. Ensure that the following XML namespace is added to root element of the view's XAML file.

**XAML**

xmlns:prism="clr-namespace:Microsoft.Practices.Prism.Commands;assembly=Microsoft.Practices.Prism"

1. Bind a button to the command in Silverlight using the **Click.Command** attached property, as shown in the following code example.

**XAML**

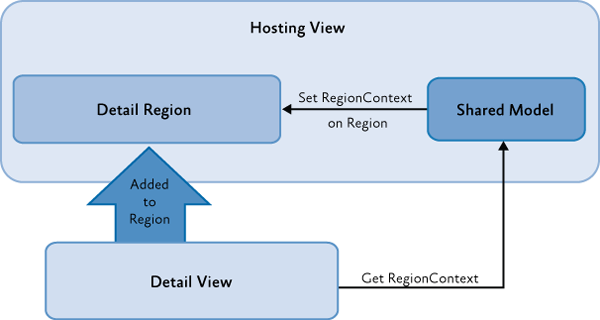
<Button Name="MyCommandButton" prism:Click.Command="{Binding MyCompositeCommand}"/>Execute MyCommand</Button>

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| Another approach is to store the command as a resource inside the App.xaml file in the **Application.Resources** section. Then, in the view—which must be created after setting that resource—you can set **prism: Click.Command="{Binding MyCompositeCommand, Source={StaticResource GlobalCommands}}"** to add an invoker to the command. |

# Region Context

There are a lot of scenarios where you might want to share contextual information between the view that is hosting a region and a view that is inside a region. For example, a master detail–like view shows a business entity and exposes a region to show additional detail information for that business entity. The Prism Library uses a concept named **RegionContext** to share an object between the host of the region and any views that are loaded inside the region, as shown in the following illustration.

**Using RegionContext**



Depending on the scenario, you can choose to share a single piece of information (such as an identifier) or a shared model. The view can retrieve the **RegionContext**, and then sign up for change notifications. The view can also change the **RegionContext**'s value. There are several ways of exposing and consuming the **RegionContext**:

* You can expose **RegionContext** to a region in Extensible Application Markup Language (XAML).
* You can expose **RegionContext** to a region in code.
* You can consume **RegionContext** from a view inside a region.

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| The Prism Library currently only supports consuming the **RegionContext** from a view inside a region if that view is a**DependencyObject**. If your view is not a **DependencyObject** (for example, you are using WPF automatic data templates and adding your view model directly in the region), consider creating a custom **RegionBehavior** to forward the **RegionContext** to your view objects. |

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| **About the Data Context Property** Data context is a concept that allows elements to inherit information from their parent elements about the data source that is used for binding. Child elements automatically inherit the **DataContext** of their parent element. The data flows down the visual tree. The best method for binding a view model to a view in Silverlight is by using the **DataContext** property; that is why, in most cases, the**DataContext** is used to store the view's model.  Because of this, unless you have very simple views, it is not recommended that you use the **DataContext** property as a communication mechanism between different loosely coupled views. |

# Shared Services

Another method of cross-module communication is through shared services. When the modules are loaded, modules add their services to the service locator. Typically, services are registered and retrieved from a service locator by common interface types. This allows modules to use services provided by other modules without requiring a static reference to the module. Service instances are shared across modules, so you can share data and pass messages between modules.

In the Stock Trader Reference Implementation (Stock Trader RI), the Market module provides an implementation of **IMarketFeedService**. The Position module consumes these services by using the shell application's dependency injection container, which provides service location and resolution. The **IMarketFeedService** is meant to be consumed by other modules, so it can be found in the**StockTraderRI.Infrastructure** common assembly, but the concrete implementation of this interface does not need to be shared, so it is defined directly in the Market module and can be updated independently of other modules.

To see how these services are exported into the MEF container, see the **MarketFeedService.cs** and **MarketHistoryService.cs** files, as shown in the following code example. The Position module's **ObservablePosition**receives the **IMarketFeedService** service through constructor dependency injection.

**C# MarketFeedService.cs**

[Export(typeof(IMarketFeedService))] [PartCreationPolicy(CreationPolicy.Shared)] public class MarketFeedService : IMarketFeedService, IDisposable { ... }

This helps with cross-module communication because service consumers do not need a static reference to modules providing the service. This service can be used to send or receive data between modules.

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| Some dependency injection containers allow the registration of dependencies using attributes, as shown in this example. Other containers may use explicit registration. In these cases, the registration typically occurs during module loading when Prism invokes the**IModule.Initialize** method. See Chapter 4, "[Modular Application Development](http://msdn.microsoft.com/en-us/library/gg405479(v=PandP.40).aspx)" for 更多. |

# Event Aggregation

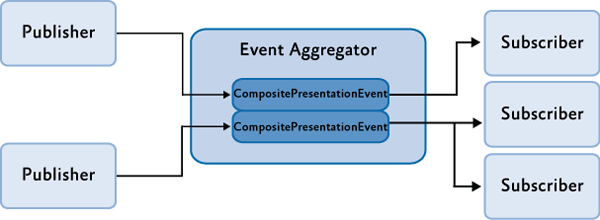
The Prism Library provides an event mechanism that enables communications between loosely coupled components in the application. This mechanism, based on the event aggregator service, allows publishers and subscribers to communicate through events and still do not have a direct reference to each other.

The**EventAggregator** provides multicast publish/subscribe functionality. This means there can be multiple publishers that raise the same event and there can be multiple subscribers listening to the same event. Consider using the **EventAggregator** to publish an event across modules and when sending a message between business logic code, such as controllers and presenters.

One example of this, from the Stock Trader RI, is when the **Process Order** button is clicked and the order successfully processes; in this case, other modules need to know the order is successfully processed so they can update their views.

Events created with the Prism Library are typed events. This means you can take advantage of compile-time type checking to detect errors before you run the application. In the Prism Library, the **EventAggregator** allows subscribers or publishers to locate a specific **EventBase**. The event aggregator also allows for multiple publishers and multiple subscribers, as shown in the following illustration.

**Event aggregator**



|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| **About .NET Framework Events** Using .NET Framework events is the most simple and straightforward approach for communication between components if loose coupling is not a requirement. Events in the .NET Framework implement the Publish-Subscribe pattern, but to subscribe to an object, you need a direct reference to that object, which, in composite applications, typically resides in another module. This results in a tightly coupled design. Therefore, .NET Framework events are used for communication within modules instead of between modules. If you use .NET Framework events, you have to be very careful of memory leaks, especially if you have a non-static or short-lived component that subscribes to an event on a static or longer-lived one. If you do not unsubscribe the subscriber, it will be kept alive by the publisher, and this will prevent the first one from being garbage-collected. |

## IEventAggregator

The **EventAggregator**class is offered as a service in the container and can be retrieved through the **IEventAggregator** interface. The event aggregator is responsible for locating or building events and for keeping a collection of the events in the system.

**C#**

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

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

## CompositePresentationEvent

The real work of connecting publishers and subscribers is done by the **CompositePresentationEvent** class. This is the only implementation of the **EventBase** class that is included in the Prism Library. This class maintains the list of subscribers and handles event dispatching to the subscribers.

The **CompositePresentationEvent** 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 **CompositePresentationEvent** class.

**C# CompositePresentationEvent.cs**

public class CompositePresentationEvent<TPayload> : EventBase { ... public SubscriptionToken Subscribe(Action<TPayload> action); public SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption threadOption); public SubscriptionToken Subscribe(Action<TPayload> action, bool keepSubscriberReferenceAlive) public virtual SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption threadOption, bool keepSubscriberReferenceAlive); public virtual SubscriptionToken Subscribe(Action<TPayload> action, ThreadOption threadOption, bool keepSubscriberReferenceAlive, Predicate<TPayload> filter); public virtual void Publish(TPayload payload); public virtual void Unsubscribe(Action<TPayload> subscriber); public virtual bool Contains(Action<TPayload> subscriber) ... }

## Creating and Publishing Events

The following sections describe how to create, publish, and subscribe to **CompositePresentationEvent** using the **IEventAggregator**interface.

### Creating an Event

The **CompositePresentationEvent<TPayload>** is intended to be the base class for an application's or module's specific events. **TPayLoad**is the type of the event's payload. The payload is the argument that will be passed to subscribers when the event is published.

For example, the following code shows the **TickerSymbolSelectedEvent** in the Stock Trader Reference Implementation (Stock Trader RI). The payload is a string containing the company symbol. Notice how the implementation for this class is empty.

**C#**

public class TickerSymbolSelectedEvent : CompositePresentationEvent<string>{}

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| In a composite application, the events are frequently shared between multiple modules, so they are defined in a common place. In the Stock Trader RI, this is done in the **StockTraderRI.Infrastructure**project. |

### Publishing an Event

Publishers raise an event by retrieving the event from the **EventAggregator** andcalling the **Publish** method. To access the**EventAggregator**, you can use dependency injection by adding a parameter of type **IEventAggregator** to the class constructor.

For example, the following code demonstrates publishing the **TickerSymbolSelectedEvent**.

**C#**

this.eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Publish("STOCK0");

## Subscribing to Events

Subscribers can enlist with an event using one of the **Subscribe** method overloads available on the **CompositePresentationEvent** class. There are several ways to subscribe to **CompositePresentationEvents**. Use the following criteria to help determine which option best suits your needs:

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

The following sections describe these options.

pandpDocTools-topicTitle: 9: Communicating Between Loosely Coupled Components

errorText: The following sections describe these options.

### Subscribing on the UI Thread

Frequently, subscribers will need to update UI elements in response to events. In WPF and Silverlight, only a UI thread can update UI elements.

By default, the subscriber receives the event on the publisher's thread. If the publisher sends the event from the UI thread, the subscriber can update the UI. However, if the publisher's thread is a background thread, the subscriber may be unable to directly update UI elements. In this case, the subscriber would need to schedule the updates on the UI thread using the **Dispatcher** class.

The **CompositePresentationEvent** provided with the Prism Library can assist by allowing the subscriber to automatically receive the event on the UI thread. The subscriber indicates this during subscription, as shown in the following code example.

**C#**

public void Run() { ... this.eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews, ThreadOption.UIThread); ); } public void ShowNews(string companySymbol) { this.articlePresentationModel.SetTickerSymbol(companySymbol); }

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

* **PublisherThread**. Use this setting to receive the event on the publishers' thread. This is the default setting.
* **BackgroundThread**. Use this setting to asynchronously receive the event on a .NET Framework thread-pool thread.
* **UIThread**. Use this setting to receive the event on the UI thread.

### Subscription Filtering

Subscribers may not need to handle every instance of a published event. In these cases, the subscriber can use the **filter** parameter. The**filter** parameter is of type **System.Predicate<TPayLoad>** and is a delegate that gets executed when the event is published to determine if the payload of the published event matches a set of criteria required to have the subscriber callback invoked. If the payload does not meet the specified criteria, the subscriber callback is 注意:xecuted.

Frequently, this filter is supplied as a lambda expression, as shown in the following code example.

**C#**

FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>(); fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread, false, fundOrder => fundOrder.CustomerId == this.customerId);

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| Due to Silverlight's security model, the methods supplied to **Subscribe** must be publicly accessible methods if you want a weakly-held subscription (which is the default **CompositePresentationEvent** behavior). Because lambda expressions and anonymous delegates are generated as private classes and methods, these cannot be used in conjunction with weak subscriptions. Instead, you will either need to use public methods or subscribe to the event with a **keepSubscriberReferenceAlive** option to force a strongly-held subscription (see example below). |

For Silverlight, due to its security model, you need to call a separate public method, as shown in the following code example, or subscribe using strong references (shown in a later example).

**C#**

public bool FundOrderFilter(FundOrder fundOrder) { return fundOrder.CustomerId == this.customerId; } ... FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>(); subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread, false, FundOrderFilter);

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| The **Subscribe** method returns a subscription token of type **Microsoft.Practices.Prism.Events.SubscriptionToken** that can be used to remove a subscription to the event later. This token is particularly useful when you are using anonymous delegates or lambda expressions as the callback delegate or when you are subscribing the same event handler with different filters. |

|  |
| --- |
| **f921122.note(en-us,PandP.40).gif注意:** |
| It is not recommended to modify the payload object from within a callback delegate because several threads could be accessing the payload object simultaneously. You could have the payload be immutable to avoid concurrency errors. |

### Subscribing Using Strong References

If you are raising multiple events in a short period of time and have noticed performance concerns with them, you may need to subscribe with strong delegate references. If you do that, you will then need to manually unsubscribe from the event when disposing the subscriber.

By default,**CompositePresentationEvent** maintains a weak delegate reference to the subscriber's handler and filter on subscription. This means the reference that **CompositePresentationEvent** holds on to will not prevent garbage collection of the subscriber. Using a weak delegate reference relieves the subscriber from the need to unsubscribe and allows for proper garbage collection.

However, maintaining this weak delegate reference is slower than a corresponding strong reference. For most applications, this performance will not be noticeable, but if your application publishes a large numberof events in a short period of time, you may need to use strong references with **CompositePresentationEvent**. If you do use strong delegate references, your subscriber should unsubscribe to enable proper garbage collection of your subscribing object when it is no longer used.

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

**C#**

FundAddedEvent fundAddedEvent = eventAggregator.GetEvent<FundAddedEvent>(); bool keepSubscriberReferenceAlive = true; fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread, keepSubscriberReferenceAlive, fundOrder => fundOrder.CustomerId == \_customerId);

The **keepSubscriberReferenceAlive** parameter is of type **bool**:

* When set to **true**, the event instance keeps a strong reference to the subscriber instance, thereby not allowing it to get garbage collected. For information about how to unsubscribe, see the section "Unsubscribing from an Event" later in this topic.
* When set to **false** (the default value when this parameter omitted), the event maintains a weak reference to the subscriber instance, thereby allowing the garbage collector to dispose the subscriber instance when there are no other references to it. When the subscriber instance gets collected, the event is automatically unsubscribed.

### Default Subscriptions

For a minimal or default subscription, the subscriber must provide a callback method with the appropriate signature that receives the event notification. For example, the handler for the **TickerSymbolSelectedEvent** requires the method to take a string parameter, as shown in the following code example.

**C#**

public void Run() { ... this.eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews); } public void ShowNews(string companySymbol) { articlePresentationModel.SetTickerSymbol(companySymbol); }

### Unsubscribing from an Event

If your subscriber no longer wants to receive events, you can unsubscribe by using your subscriber's handler or you can unsubscribe by using a subscription token.

The following code example shows how to directly unsubscribe to the handler.

**C#**

FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>(); fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.PublisherThread); fundAddedEvent.Unsubscribe(FundAddedEventHandler);

The following code example shows how to unsubscribe with a subscription token. The token is supplied as a return value from the**Subscribe** method.

**C#**

FundAddedEvent fundAddedEvent = this.eventAggregator.GetEvent<FundAddedEvent>(); subscriptionToken = fundAddedEvent.Subscribe(FundAddedEventHandler, ThreadOption.UIThread, false, fundOrder => fundOrder.CustomerId == this.customerId); fundAddedEvent.Unsubscribe(subscriptionToken);

# 更多

更多关于弱引用的详细信息，请查看MSDN上的"Weak References"： <http://msdn.microsoft.com/en-us/library/ms404247.aspx>.

# 10: Silverlight和WPF之间共享代码

|  |
| --- |
| **索引:** |
| [**目标和好处**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec1) | [**讨论之外**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec2) | [**多目标方案**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec3) | [**多目标注意事项**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec4) | [**多目标元素**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec5) | [**一个多目标解决方案: 多连接项目**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec6) | [**核心应用**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec7) | [**创建多目标应用**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec8) | [**设计和编码设计指南**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec9) – [*使用Separated Presentation模式*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec10)*，最大限度的共享代码*, [*编写可以在所有平台上运行的代码*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec11), [*使用 #if语句，如果你使用简单或者单行的构造*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec12) , [*使用Partial类，如果大多数类都一样，但是在特定平台上需要特别实现*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec13), [*使用Partial方法，如果你只需要在特定平台上调用一个扩展方法*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec14), [*为特定平台编写机能单一的类*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec15), [*分别为Silverlight和WPF创建解决方案文件夹*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec16), [*在编码时检查 Silverlight和WPF引用*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec17) | [**流程设计指南**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec18) – [*在Silverlight*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec19)*中开发核心应用*,[*链接在源工程和目标工程之间共享代码*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec20), [*为Silverlight和WPF工程使用同样的命名空间*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec21) |[**Team Build指南**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec22) – [*正确配置Team Build*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec23) | [**对比Silverlight和WPF**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec24) | [**Silverlight和WPF体系结构概述**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec25) | [**Silverlight和WPF**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec26)**之间的区别** – [*资源*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec27), [*触发器*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec28), [*数据绑定*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec29), [*命令*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec30), [*其他*](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec31)| [**更多**](http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40)#sec32) |

This topic helps you understand multi-targeting from a Prism perspective and its advantages and disadvantages. Multi-targeted code targets two different platforms with largely the same code-base. This allows binaries targeting two different technologies to be produced while keeping the code as much the same as possible. In this case, the technologies this topic describes are Windows Presentation Foundation (WPF) and Silverlight. This topic includes some considerations you should think about when developing multi-targeting applications for these technologies.

# Goal and Benefits

When writing applications for WPF and Silverlight that have similar features and capabilities, it makes sense to strive for a single code-base. Although the WPF and Silverlight platforms are very similar, they have limited binary compatibility. Only Silverlight 4 assemblies that reference a certain set of core, portable framework assemblies can be loaded into the .NET Framework 4.0 runtime.

Because Prism offers largely the same capabilities for both WPF and Silverlight, much of your code can be built that targets both of these technologies. Supporting multi-targeted applications is primarily about implementing the patterns and infrastructure that maximize the possibility for sharing code and components between the two environments, and for allowing an application to integrate environment-specific functionality so that it can take full advantage of desktop or browser-specific features. By creating your multi-targeting composite application using Prism, you can reuse source code across WPF and Silverlight applications.

# Out of Scope

If you can structure your assemblies to take advantage of the binary compatibility support between Silverlight and WPF, you should do so. For 更多 about this, see the CLR Team Blog post, "[Sharing Silverlight Assemblies with .NET Apps](http://blogs.msdn.com/b/clrteam/archive/2009/12/01/sharing-silverlight-assemblies-with-net-apps.aspx)."

This chapter is not intended to describe this scenario; instead, it describes the challenges and solutions for building multi-targeted applications by sharing source code.

# Multi-Targeting Scenarios

The primary scenario is for applications that deliver both a feature-rich desktop experience and a wide-reach Internet application experience. In this scenario, you may want to develop an application that has the same features and workflow on WPF and Silverlight or one that offers different features and workflows. The following are some applications of multi-targeting:

* You can provide users with a full-featured application while they are in the office and a scaled, browser-based version for when they are traveling.
* You can provide internal users with a desktop-based application and external customers or partners with a browser-hosted application.

For example, a business may have both a call-center application for customers who want to place their orders over the phone and an online order application for customers who want to place their orders online. However, the forms are not the exactly the same. The call-center desktop order form offers 更多 and expanded functionality than the online order form. However, because they accomplish similar things there will be certain parts of the order form and business logic that can be reused across both scenarios.

Service-oriented applications are easier to multi-target because Silverlight is inherently service-oriented. Silverlight does not have support for local storage or database access because of its targeted feature set and security restrictions. Additionally, connected applications are also easier to multi-target because of Silverlight's connected nature.

# Multi-Targeted Considerations

By making your solution multi-targetable, you should also consider the following:

* Silverlight offers limited storage on the local client computer in isolated storage.
* You may lose simplicity and readability of code in your multi-targetable solution. Because some features of WPF are not available in Silverlight, you will need to work around these issues and your code may not be as elegant or readable.
* By default, Silverlight applications execute in a secure sandbox, so there are a several things that you cannot do outside of the sandbox context. These applications have restricted access to the local computer and are constrained to help prevent malicious behavior. These restrictions prevent access to devices and interacting with other running programs.

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| If you need access to some of these items, you can create an out-of-browser Silverlight application with elevated trust. For 更多, see "[Trusted Applications](http://msdn.microsoft.com/en-us/library/ee721083(VS.95).aspx)" on MSDN. |

* Silverlight supports only asynchronous communications, so you cannot multi-target applications that use synchronous communications.

# Multi-Targeted Elements

Typically there is a significant amount of code that is unrelated to the actual presentation technology. Because of the very close nature of the Silverlight and .NET Framework runtimes, the bulk of this code can be shared between both technologies. This also encourages heavy use of Separated Presentation patterns to isolate the logic of the presentation from the actual visual presentation, to help maximize the separation between user interface (UI) and non-UI code. Usually, you can multi-target the following source code elements:

* **Presentation patterns**. Patterns such as Model-View-ViewModel (MVVM), Model-View-Presenter (MVP), and Model-View-Controller (MVC) can be used if the logic is largely the same across the platforms.
* **Services**. Services that assist in presentation can often be multi-targeted.
* **单元测试**. Many 单元测试 can be multi-targeted with tools, such as the Silverlight Unit Test, Framework that use the same attribute structure as MSTest.
* **Simple views**. If the XAML used is supported by both Silverlight and WPF. If a view consists of only basic controls and simple data binding, it is possible that it can also be shared between WPF and Silverlight.

Silverlight's API is largely a subset of the .NET Framework's API, so it often makes sense to develop your application against this smaller API to reduce the chance of using a feature not available for both platforms. Because of the differences between the XAML in Silverlight and WPF, the following elements are harder to reuse:

* Complex views (XAML)
* Controls
* Styling
* Animation
* Expression Blend behaviors and triggers

# A Solution to Multi-Targeting: Multiple Linked Projects

Silverlight and WPF have only limited binary compatible, so some code and components have to be recompiled for each target environment. The approach Prism takes is to provide guidance on structuring application and module code into multiple linked projects. Each project manages all the references, resources, and code specific to the WPF or Silverlight target environment. Code that is shared is linked between two projects so that it is automatically compiled into each target. 单元测试 can similarly be shared between the two environments, so that they can be written once and run against the target code in either of the two target environments. The Prism team created a tool named Project Linker to help create and maintain these links. The Project Linker was used to link projects in the Prism Library, QuickStarts, and reference implementations.

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| You can download [Project Linker](http://visualstudiogallery.msdn.microsoft.com/en-us/5e730577-d11c-4f2e-8e2b-cbb87f76c044) from Visual Studio Gallery or open Visual Studio, click **Extension Manager** on the **Tools** menu, click**Online Gallery** in the **Extension Manager** dialog box, and then search for "Project Linker." |

Non-UI code and components are probably going to be the easiest to share, so adhering to separated UI patterns is essential in making sure that the UI and non-UI pieces of the application or modules are cleanly separated.

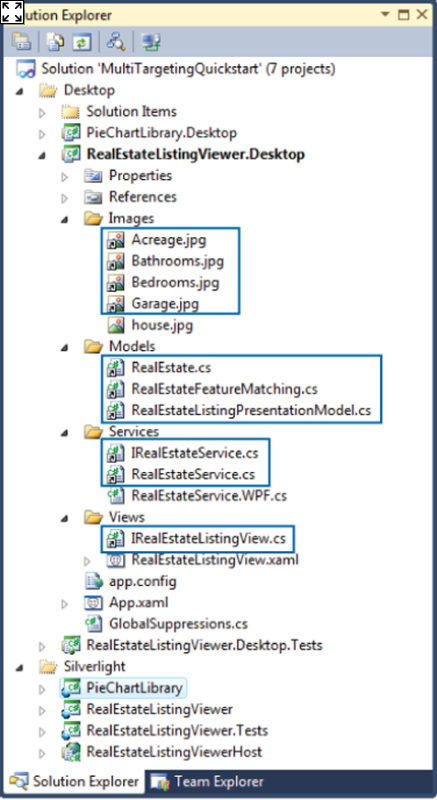
## Core Application

The overall pattern is based on defining the core application in shared code and then augmenting that with extensions that implement WPF (desktop) or Silverlight (browser) specific functionality. The core application defines the overall structure of the application and contains the application code and components that are common to the two environments. Silverlight is largely a subset of WPF, so developing the core application in Silverlight reduces the risk of relying on an API or feature that is available in WPF but not in Silverlight.

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| Many solutions in the Prism source tree actually have the core source files in the WPF projects and the Silverlight projects link to the files in the WPF projects. This is mostly because of historical reasons since the first version of Prism was built prior to the first Silverlight release. |

The following illustration shows the Solution Explorer view for the Multi-Targeted QuickStart. Most files in the WPF version of the QuickStart are linked files because the core application was developed in Silverlight. The shared files found in the QuickStart are images, models, services, interfaces, and resources. The shared (linked) files are highlighted.

**Shared files in the Multi-Targeting QuickStart**

[](http://msdn.microsoft.com/en-us/library/Ff921109.b797dc251af61200b5c7992b5aa6297c(l=en-us,v=pandp.40).png)

# Creating Multi-Targeted Applications

This section describes three areas to consider when developing a multi-targeted application:

* Design and Code Guidelines. This section describes considerations for sharing code between Silverlight and WPF.
* Process Guidelines. This section describes approaches for sharing code between Silverlight and WPF.
* Team Build Guidelines. This section describes specific Microsoft Team Build issues when building multi-targeted applications.
* Contrasting Silverlight and WPF. This section describes differences between Silverlight 4 and WPF.

## Design and Code Guidelines

Design and code guidelines include the following:

* Use Separated Presentation patterns to maximize the amount of shared code.
* Where possible, write code so it compiles on both platforms. When this is not possible, do the following:
  + Use #if statements if you have simple or single line constructs.
  + Use partial classes when most of the class is similar but some methods have platform-specific implementations.
  + Use partial methods only if you need to call an extra method on one platform but not the other.
  + Build platform-specific classes with a single responsibility.
* Create a solution folder for Silverlight and another for WPF.
* Check Silverlight and WPF references when refactoring code.

The next sections describe each of these guidelines in more detail.

### Use Separated Presentation Patterns to Maximize the Amount of Shared Code

Sharing view-code across platforms can be difficult. Sharing presentation and business logic is easier if you separate this logic from the UI logic. Additionally, this makes your code easier to understand and maintain.

### Write Code So It Compiles on Both Platforms

Where possible, write your application code so that it compiles on both platforms to enable reuse. When this approach becomes too complicated, you have to make the tradeoff to see if the cost of having two codebases is less than having a less elegant solution. For example, in Silverlight you can execute the following LINQ expression against the **Items** property of an **ItemsControl**.

**C#**

ItemsControl someItemsControl = new ItemsControl(); someItemsControl.Items.Add(new TextBox()); bool hasDependencyObjects = someItemsControl.Items.Any(item => item is DependencyObject);

However, because the **Items** property is not **IEnumerable<T>** in WPF, this preferred approach does not work. Instead of creating a different version for WPF and Silverlight, opt for a less-preferred but single source solution, as shown in the following code.

**C#**

ItemsControl someItemsControl = new ItemsControl(); someItemsControl.Items.Add(new TextBox()); bool hasDependencyObjects = false; foreach (var item in someItemsControl.Items) { if (item is DependencyObject) { hasDependencyObjects = true; break; } }

The easiest approach is to begin writing your code in Silverlight, because it is a more constrained version of the .NET Framework.

#### Use #if Statements If You Have Simple or Single Line Constructs

Sometimes it is not possible to create a single code base because of incompatibility between WPF and Silverlight. In this case, you can use #if SILVERLIGHT constructs to create conditional compiled sections. The following code example shows a #if SILVERLIGHT statement.

**C#**

#if SILVERLIGHT System.Windows.Application.Current.RootVisual = shell; #else Application.Current.MainWindow = shell; shell.Show(); #endif

However, #if statements have several drawbacks:

* The code is less readable and maintainable. If code is scattered with #if constructs, it becomes hard to read and a lot harder to maintain.
* Debugging becomes harder. If there is a compiler error within a construct, when you try to open the file, Visual Studio selects the solution that has the physical file instead of the solution that has the error. This means you either have to manually close the file and open the correct solution or edit the code without IntelliSense.

#### Use Partial Classes When Most of the Class Is Similar but Some Methods Have Platform-Specific Implementations

When the changes between Silverlight and WPF become more complex, you can create partial classes. Mark the shared class files as **partial**and then create a partial class for the specific platforms. This also applies to 单元测试. Below are some additional recommendations:

* Try to keep the platform-specific methods private. This way, the 单元测试 will not need to contain specific logic for specific platforms.
* Make sure your class has a single, clear responsibility. Any partial methods for platform-specific code should only change the implementation details.

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| If the differences between the two platforms become very extensive, and the classes for both platforms become very different, consider creating platform-specific classes instead of partial class. |

The following code example shows a shared partial class, **RealEstateService**, that is shared between the Silverlight and WPF projects in the Multi-Targeting QuickStart.

**C# RealEstateService.cs**

namespace RealEstateListingViewer.Services { public partial class RealEstateService: IRealEstateService { public RealEstate GetRealEstate () { ... return property; } }

The following code example shows the Silverlight-specific partial class for retrieving images for the **RealEstateService**class in the Multi-Targeting QuickStart.

**C# RealEstateService.silverlight.cs**

namespace RealEstateListingViewer.Services { public partial class RealEstateService { /// <summary> /// Return the images. In Silverlight, you want to download the image /// from a web server. /// You can either store the images on the server or build an /// HTTP handler to retrieve the images. /// </summary> private static BitmapImage GetImage() { Uri imageUri; Uri source = App.Current.Host.Source; if (source.ToString().StartsWith("file://", StringComparison.OrdinalIgnoreCase)) { imageUri = new Uri("../Images/House.jpg", UriKind.Relative); } else { source = new Uri(string.Format(CultureInfo.InvariantCulture, "{0}://{1}:{2}/", source.Scheme, source.Host, source.Port)); imageUri = new Uri(source, "Images/house.jpg"); } return new BitmapImage(imageUri); } } }

The following code example shows the WPF-specific partial class for retrieving images for the **RealEstateService**class in the Multi-Targeting QuickStart.

**C# RealEstateService.WPF.cs**

namespace RealEstateListingViewer.Services { public partial class RealEstateService { /// <summary> /// Return the images. In a windows application, normally you /// retrieve the image from a database. /// But for simplicity, it is just being retrieved from the file system. /// </summary> private static BitmapImage GetImage() { return new BitmapImage(new Uri("../Images/house.jpg", UriKind.Relative)); } } }

#### Use Partial Methods Only if You Need to Call an Extra Method on One Platform but Not the Other

If some work needs to be performed only in either Silverlight or WPF, you could also use partial methods. This means you can put an interface for the method in the parent class and put an implementation of that interface in only one of the platform-specific classes. For the other platform, the compiler will remove the method call. There are several limitations to partial methods:

* Partial method declarations must begin with the contextual keyword **partial** and the method must return **void**.
* Partial methods can have **ref** parameters but not **out** parameters.
* Partial methods are implicitly **private**; therefore, they cannot be **virtual**.
* Partial methods cannot be **extern**, because the presence of the body determines whether they are defining or implementing.
* Partial methods can have **static** and **unsafe** modifiers.
* Partial methods can be generic. Constraints are put on the defining partial method declaration and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
* You cannot make a **delegate** to a partial method.

#### Build Platform-Specific Classes with a Single Responsibility

Frequently, it makes more sense to factor all platform-specific code into a separate class (for example, services or service agents). This can happen if most of the logic differs between the platforms. This way, you can create platform-specific implementations for services, such as caching, data access, or authentication. This approach also works for providing functionality that is only present in one platform. The following are some additional recommendations:

* Use a common interface to share code between the different platforms. For example, in the Prism Library, there are several platform specific classes for loading module types, such as the **XapModuleTypeLoader** for Silverlight and the **FileModuleTypeLoader** for desktop applications. They both implement the **IModuleTypeLoader** interface.
* When there is some shared functionality between the different platforms, favor composition over inheritance (for example, using the strategy pattern). In other words, see if it makes sense to factor out the shared code in a shared class with a specific responsibility. In some scenarios, inheritance makes sense.

### Create a Solution Folder for Silverlight and Another for WPF

Use solution folders to keep your solution organized. Typically, you do this by using two solution folders, one for the Silverlight code and the other for WPF code. For an example of how to structure your solution, see the Multi-Targeting QuickStart.

### Check Silverlight and WPF References When Refactoring Code

Sometimes a Silverlight reference might slip into a WPF project or vice versa. This is caused by using refactoring tools. If you get unexpected compiler errors about Silverlight assemblies not being referenced in your WPF project, check the references.

## Process Guidelines

Process guidelines include the following:

* Develop the core application in Silverlight.
* Link the shared code between the source project and the target project.
* Use the same namespace for Silverlight and WPF projects.

The next sections describe each of these guidelines in more detail.

### Develop the Core Application in Silverlight

Because Silverlight is a subset of WPF, the same code will work on WPF without major modifications, so you should develop your core application in Silverlight.

### Link the Shared Code Between the Source Project and the Target Project

For files that are common to both Silverlight and WPF but need different references, you should link the files. One of the ways to do this is to use the Project Linker tool to link the shared code between the source project and the target project.

### Use the Same Namespace for Silverlight and WPF Projects

Keep the namespaces the same between projects because the shared code requires the same namespace.

## Team Build Guidelines

This section describes Team Build guidelines.

### Configure Team Build to Build in Place

If you use Team Build to build a solution that holds both WPF and Silverlight projects, you might run into file collision problems. By default, Team Build will copy the output from the projects to a single output folder. Because the output of the WPF and Silverlight projects should have the same name, there is a file name collision problem that will prevent you from compiling the projects or running 单元测试.

By setting the property **CustomizableOutDir** to **true**, you are telling Team Build to build in place instead of copying the output to a single location. This prevents the collision name problem.

Additionally, if you set this property and want to run 单元测试 in your build, you also need to specify where the **TestContainers** are located.

**XML**

<PropertyGroup> <!--Build in place--> <CustomizableOutDir>true</CustomizableOutDir> </PropertyGroup> <!—Override the BeforeTestConfiguration target to specify where the testcontainers live. --> <Target Name="BeforeTestConfiguration"> <!-- Change the outdir, because the testtoolstask needs this to execute all the tests --> <PropertyGroup> <OutDir>$(SolutionRoot)\Source\</OutDir> </PropertyGroup> <!--Create a list of all tests dll's to run (test only the desktop versions) --> <CreateItem Include="$(SolutionRoot)\\*\*\Desktop\\*\*\Bin\Debug\%2a.Tests.dll"> <Output ItemName="TestContainer" TaskParameter="Include" /> </CreateItem> </Target>

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| This example assumes that your desktop projects are located in a folder named Desktop. MSTest.exe is only able to run 单元测试 that are targeting the desktop version of the .NET Framework, so Silverlight test assemblies are excluded. |

# Contrasting Silverlight and WPF

Silverlight and WPF both allow you to develop rich user experiences based on XAML and the .NET Framework. However, there are some differences between these platforms, and these differences have to be carefully considered when transitioning an application between Silverlight and WPF or when building an application that targets both WPF and Silverlight.

|  |
| --- |
| **f921109.note(en-us,PandP.40).gif注意:** |
| This topic describes differences between Silverlight 4 and WPF that is part of the .NET Framework 4.0. These differences are expected to be reduced in future versions of Silverlight and WPF. |

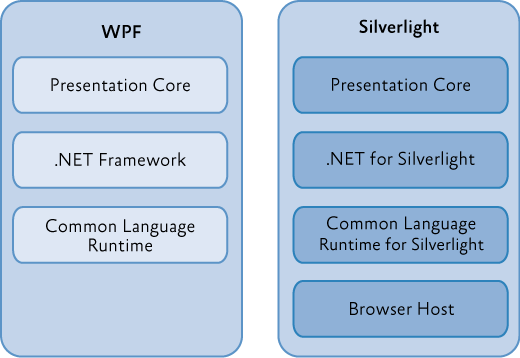
## Silverlight and WPF Architectural Overview

WPF provides developers with a unified programming model for building rich Windows applications that incorporate UI, media, and documents. WPF enables software developers to deliver a new level of "user experience" (UX) by providing a declarative-based language (XAML) for specifying vector-based graphics that can scale and take advantage of hardware acceleration.

Silverlight is a cross-browser, cross-platform implementation of the .NET Framework for delivering next-generation rich interactive media and content over the web, desktop business applications, and browser-hosted Rich Internet Applications (RIAs) that can integrate data and services from many sources. Silverlight enables developers to build applications that significantly enhance the typical end user experience, compared with traditional web applications. Like WPF, Silverlight provides a XAML-based language to specify user interfaces.

Silverlight and WPF share many of the same features and capabilities, but they are built on top of different run-time stacks, as illustrated in the following image. WPF leverages the full .NET Framework and executes on the common language runtime (CLR). Silverlight is based on a subset of XAML and the full .NET Framework, and it executes on a different version of the CLR.

**WPF and Silverlight**



## Differences Between Silverlight and WPF

To keep Silverlight small and lightweight, some WPF and .NET Framework features are not available in Silverlight. Because of this, there can be subtle—and not so subtle—differences that have to be carefully considered when moving an application between Silverlight and WPF or when building an application that targets both WPF and Silverlight. This section describes some of the major differences the patterns & practices team encountered during Prism development. These differences relate to Silverlight 4 and WPF 4.0, the current versions at the time of this writing.

pandpDocTools-topicTitle: 10: Sharing Code Between Silverlight and WPF

errorText: To keep Silverlight small and lightweight, some WPF and .NET Framework features are not available in Silverlight. Because of this, there can be subtle—and not so subtle—differences that have to be carefully considered when moving an application between Silverlight and WPF or when building an application that targets both WPF and Silverlight. This section describes some of the major differences the patterns & practices team encountered during Prism development. These differences relate to Silverlight 4 and WPF 4.0, the current versions at the time of this writing.

### 资源

Resources are simple collections of key-value pairs that can store almost any element (strings, brushes, styles, data sources, and many others). Resources can be associated with almost any element class that exposes a **Resources** property of type **ResourceDictionary**. The following are the main differences between Silverlight and WPF concerning resources:

* Resources can contain static or dynamic content. Dynamic content can be changed at any time and consumers of the resource will be automatically updated. Dynamic resource references are not supported in Silverlight; therefore, only static resource references are available.

### 触发器

Triggers allow designers to define the visual behavior of a control by declaratively specifying how its properties change in response to events or property changes, such as highlighting a button when it is clicked. Typically, triggers are fired when a property of a control changes and results in one or more other properties of that control also changing. Triggers are defined inside a style and can be applied to any object of the specified target type.

Silverlight does not support triggers in **Styles**, **ControlTemplates**, or **DataTemplates**(the **Triggers** collection does 注意:xist on these elements). However, similar behavior can be achieved by using the Silverlight Visual State Manager (VSM) to define interactive control templates. Using VSM, the visual behavior of a control, including any animations and transitions, are defined in the control template. This can be easily done by using Expression Blend 4. However, be aware that the XAML file will get more complex and that control templates built for Silverlight are not yet compatible with WPF.

The Expression Blend SDK provides Expression Blend behaviors for animation and visual state management that can be used to apply a consist look and feel between WPF and Silverlight UIs.

### 数据绑定

Both WPF and Silverlight provide data binding support. The following are the main differences between Silverlight and WPF data binding:

* In Silverlight, there is no **OneWayToSource** data flow mode.
* In Silverlight, you cannot bind directly to XML data. A possible workaround for this is to convert the XML to CLR objects, and then bind to the CLR object.
* In Silverlight, there is no **XMLDataProvider** class.
* In Silverlight, there is no **DataTemplateSelector** class. In WPF, this class provides a way to choose a **DataTemplate** based on the data object and the data-bound element. However, to help support certain MVVM scenarios, Prism offers similar functionality for Silverlight in its **DataTemplateSelector** class.

### 命令

The following are the differences between Silverlight and WPF regarding commanding:

* Routed commands are not available in Silverlight. However, the **ICommand** interface is available in Silverlight, allowing developers to create their own custom commands. The Prism Library provides the **DelegateCommand** and **CompositeCommand** classes to simplify command implementation.
* In WPF, controls can be hooked up to commands through their **Command** property. By doing this, developers can declaratively associate controls and commands. This also means a control can interact with a command so that it can invoke the command and have its enabled status automatically updated. In Silverlight, several controls support the **Command** property, but not as many as in WPF. For Silverlight controls that do not offer **Command** properties, consider using Expression Blend behaviors to invoke the command based on triggers.
* There is no input gesture or input binding support in Silverlight.

### 其他

The following are some miscellaneous differences between Silverlight and WPF:

* In Silverlight, the **UIElement** class has an internal constructor; therefore, you cannot create a control inheriting from it.
* In Silverlight, there is no **x:Type** markup extension support or support for custom markup extensions.
* In Silverlight, there is no **x:Static** markup extension support or support for custom markup extensions. In Silverlight, items added to a **TabControl** control are not automatically wrapped inside a **TabItem** type, as is the case with WPF.
* All Silverlight network calls must be asynchronous.
* Silverlight networking, except for trusted out-of-browser applications, respect server-client access policies for servers outside the site of origin.

# 更多

You can download [Project Linker](http://visualstudiogallery.msdn.microsoft.com/en-us/5e730577-d11c-4f2e-8e2b-cbb87f76c044) from Visual Studio Gallery or open Visual Studio, click on **Tools**, point to **Extension Manager**, click on**Online Gallery**, and search for "Project Linker."

For 更多 about structuring your assemblies to take advantage of the binary compatibility support between Silverlight and WPF, see the CLR Team Blog post, "Sharing Silverlight Assemblies with .NET Apps":  
<http://blogs.msdn.com/b/clrteam/archive/2009/12/01/sharing-silverlight-assemblies-with-net-apps.aspx>.

For 更多 about WPF architecture, see "WPF Architecture" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ms750441.aspx>.

For 更多 about Silverlight architecture, see "Silverlight Architecture" on MSDN:  
<http://msdn.microsoft.com/en-us/library/bb404713(VS.95).aspx>.

For a summary of the differences between WPF and Silverlight, see "WPF Compatibility" on MSDN:  
<http://msdn.microsoft.com/en-us/library/cc903925(VS.95).aspx>.

For 更多 about the Visual State Manager and how it works in creating controls, see the following articles about creating customizable controls on MSDN:

* <http://msdn.microsoft.com/en-us/library/cc278064(vs.95).aspx> (for Silverlight).
* <http://msdn.microsoft.com/en-us/library/ee330302.aspx> (for WPF in .NET Framework 4).

For 更多 about creating an application with elevated trust, see "Trusted Applications" on MSDN:  
<http://msdn.microsoft.com/en-us/library/ee721083(VS.95).aspx>.

# 11: 部署Prism应用

|  |
| --- |
| **索引:** |
| [**部署 Silverlight Prism 应用**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec1) | [**以XAP格式打包Prism模块**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec2) | [**精简你的XAP格式模块的下载大小**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec3) | [**准备你的Web服务器来承载Silverlight Prism应用**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec4) | [**部署应用**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec5) | [**部署WPF Prism 应用**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec6) | [**通过ClickOnce部署WPF Prism应用**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec7) – [*ClickOnce发布流程*](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec8), [*ClickOnce部署和更新流程*](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec9) | [**更多**](http://msdn.microsoft.com/en-us/library/gg430856(v=pandp.40)#sec10) |

To successfully move a Prism application into production, you need to plan for deployment as part of the design process of your application. This chapter covers the considerations and actions you need to perform to prepare your application for deployment and the actions you need to take to get the application in the user's hands.

Silverlight and Windows Presentation Foundation (WPF) have two very different hosting environments, so the deployment considerations are different, depending on whether you are building a Silverlight Prism application or a WPF Prism application.

# 部署Silverlight Prism应用

Silverlight applications are delivered as XAP files via an HTTP request from a browser. A XAP file is really just a .zip file with a different file name extension and certain expectations about its content. A XAP file contains a set of application assemblies, an application manifest XAML file that describes the package, and possibly additional resource files that the application uses. The XAP file is downloaded by the Silverlight plug-in hosted in a web page. After the Silverlight plug-in is downloaded, it activates the application and runs it within the context of a web page.

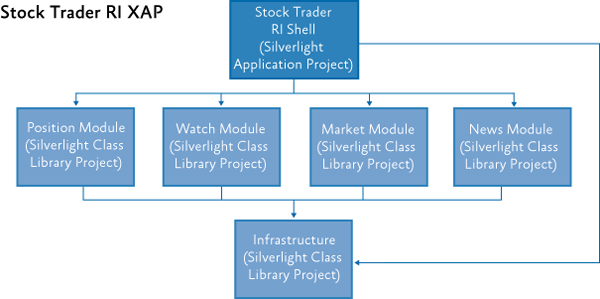
Even if you plan to run your Silverlight application Out-Of-Browser (OOB), the user will first need to access the application with the browser via its hosting page; after that, the user can then optionally install the application for OOB use. As a result, the deployment of a Silverlight Prism application is mostly a matter of getting your XAP files in the right place on a web server so the hosting page can download and run them.

## 以XAP格式打包Prism模块

As discussed in Chapter 4, "Modular Application Development," you can package your modules in multiple ways, including having multiple modules in a single XAP file or having a single XAP file act as a container for a single module. The decision should be determined by whether multiple modules need to be downloaded at the same time because of dependencies between them or the use cases of the application, or whether the modules are logically independent and decoupled in lifetime. Packaging each module in its own XAP file can be a little cleaner from a maintenance and deployment perspective, because each module XAP file becomes a separate unit of deployment that can be versioned independently and added or removed from the application through a simple change to the module catalog.

As an example, consider the architecture of the Stock Trader Reference Implementation (Stock Trader RI). It consists of the shell application and four modules: position, watch, market, and news. Because the application was designed for all those features to "light up" at application start, the modules of the Stock Trader RI are all added statically. The Stock Trader RI shell application project has references to all the module assemblies as class libraries, and the modules are loaded through the **ModuleCatalog.AddModule** method during application startup in the bootstrapper. In this case, there is only a single XAP that results from the building of the StockTraderRI.Silverlight shell application project, and it contains the shell, the four modules, and the shared infrastructure class library, as shown in the following illustration. If you were deploying the StockTraderRI application, you would simply place that XAP file on your web server and set up the source parameter tag of your Silverlight plug-in object tag in the hosting page to point to that XAP file (typically in a \ClientBin subfolder of your site, using an ASP.NET Web Application hosting project template).

**Stock Trader RI XAP structure**

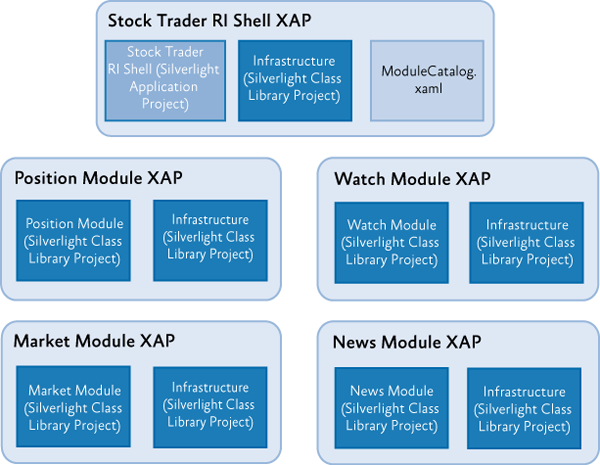


Alternatively, if the modules of the Stock Trader RI were more loosely coupled in terms of when they needed to be loaded, they could have been developed as Silverlight Application projects so they would be built into individual XAP files. If you did that, the deployment architecture would look more like the following illustration. After you break the modules into their own XAP files, you can choose to delay the loading of some of the modules, if appropriate, based on the application functionality. You would typically use a ModuleCatalog.xaml file packaged as part of the shell XAP file to determine what modules the application is composed of, what their dependencies are, and what their loading characteristics are (on-demand or not).

Notice in the following illustration that because the shell and each of the modules has a reference to the infrastructure assembly as a shared class library, if you leave the default settings on those references, you will end up with a separate copy of the library in each of the individual XAP files, which adds unnecessary download size and bandwidth utilization to the application.

|  |
| --- |
| **g430856.note(en-us,PandP.40).gif注意:** |
| If you plan to have your users install the application OOB, and you expect to deploy updates to your application after it is deployed, you will need to stick to putting all the modules in a single XAP file. The **Application.CheckAndDownloadUpdateAsync** method will only go out and update the main XAP file that the application was launched from. |

**Modified Stock Trader RI XAP structure**



## Reducing the Download Size of Your Module XAP Files

To address the duplication of shared class libraries in multiple module XAP files in a Prism application, you have a couple of options. The first is to include the shared library in the shell application XAP file, as it will be by default when you add the reference. Then you go to the reference in each of the modules that also uses the reference and set the **Copy Local** property on the reference to **false**. This causes the referenced assembly to not be included in the compiled XAP file for that module. However, at run time, the one provided by the shell application XAP file will also be available to the modules.

The second approach is to leverage application library caching in Silverlight. To use this feature, you go to the Silverlight project properties of a project and select the check box labeled **Reduce XAP size by using application library caching**. When you do this, any referenced assemblies that have the right metadata files collocated with them will not be included in the XAP file; instead, they will be placed in a separate .zip file, and the .zip file will be referenced by the XAP file's ApplicationManifest.xaml file as an external part. The signed Silverlight Prism Library binaries have the required metadata files to use this feature as well as the libraries from the Silverlight SDK and the Silverlight Toolkit. To leverage this functionality for your own shared assemblies, you need to provide a metadata file, as described in the topic, "[How to: Use Application Library Caching](http://msdn.microsoft.com/en-us/library/dd833069(VS.95).aspx)" on MSDN.

## Preparing Your Web Server to Host Silverlight Applications

Silverlight applications can be hosted on most types of web servers, such as Internet Information Services (IIS) or Apache. However, most web servers are usually configured to serve only a few well-known file name extensions. To allow Silverlight applications to be served from your web server, you have to allow the MIME types in the following table to be served.

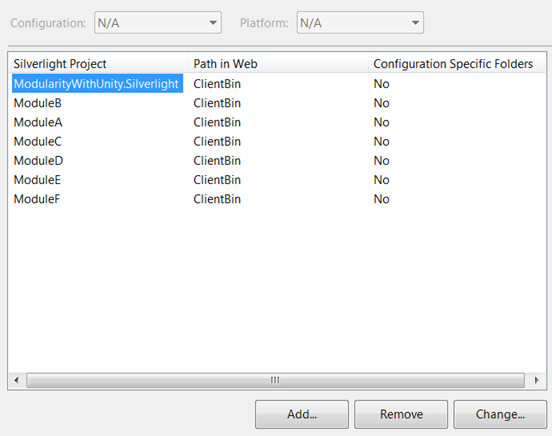
|  |  |
| --- | --- |
| Extension | MIME type |
| .xaml | application/xaml+xml |
| .xap | application/x-silverlight-app |

## Deploying the Application

To deploy a Silverlight application and the modules that are remotely loaded, the XAP files must be made accessible on the web server. There are several ways to accomplish this:

* You can manually copy all the XAP files to a public folder on the web server.
* You can include the Silverlight XAP files in a web project or website and publish from Microsoft Visual Studio. To do this with a Web Application project, the project needs to be part of the same solution as the Silverlight projects that create the XAP files. You then add the Silverlight projects to the **Silverlight Applications** tab in the web project settings, as shown in the following illustration. A copy of the XAP files from the included Silverlight projects will be placed in a \ClientBin subfolder of the published site. These files are synchronized in the web project each time you build.

**Adding the remote modules as Silverlight applications**



|  |
| --- |
| **g430856.note(en-us,PandP.40).gif注意:** |
| To avoid cross-domain call issues, the remote modules' XAP files should be located on the same domain as the main application; when deployed like this, the **Ref** property on the ModuleCatalog should be a Uniform Resource Identifier (URI) relative to the main XAP file location on the web server. |

In addition to publishing or locating the XAP files in the hosting website, the hosting web page will need to refer to the shell application XAP file in its object tag source parameter. Because the hosting page should be part of the same site that the XAP files are placed in for cross-domain reasons, the path specified in the host page should be a relative path.

**XAML**

<object data="data:application/x-silverlight-2," type="application/x-silverlight-2" width="100%" height="100%"> <param name="source" value="ClientBin/ModularityWithUnity.Silverlight.xap"/> ... </object>

# 部署 WPF Prism应用

A WPF Prism application can be composed of an executable and any number of additional DLLs. The main executable is the shell application project. Some of the additional DLLs will be the modules of the application. There may be some additional DLLs that are just shared assemblies used by the shell and modules of the application. In addition, you might have a set of resource or content files that get deployed along with the application.

To deploy a WPF Prism application, you have three choices:

* "XCopy deployment"
* ClickOnce deployment
* Windows Installer deployment

"XCopy deployment" is used as a general term for manual deployment through some sort of file copy operation, which may or may not include the use of the XCOPY command-line tool. If you choose to deploy the application in this way, it is up to you to manually package the files and move them to the target computer. The application should be ready to run as long as the expected folder structure and relative locations of the shell application executable, the module DLLs, and the content files are maintained.

Usually, a more automatic means of deployment is desired to ensure that things get placed in the right location and the user has easy access to run the application. To facilitate that, you can choose to use ClickOnce or Windows Installer (.msi files), depending on what additional installation requirements exist for the application.

The decision of whether to use ClickOnce or Windows Installer is often misunderstood. ClickOnce is not intended to be a one-size-fits-all deployment technology. It is intended for applications that need a low-impact install on a client computer. If your application needs to make computer-wide changes when it is installed—such as to install drivers, integrate with other applications, install services and other things that go outside the scope of just running your executable, ClickOnce is probably not an appropriate deployment choice. However, if you have a lightweight installation on the client computer and you want to benefit from network deployment and update of your WPF application, ClickOnce can be a great choice.

To create a Windows Installer deployment package (.msi file) for your application, you have a variety of choices, including Visual Studio Setup projects, Windows Installer XML (WiX) projects, or numerous third-party installer creation products.

## Deploying WPF Prism Application with ClickOnce

ClickOnce is a Windows Presentation Foundation (WPF) or Windows Forms deployment mechanism that has been part of the .NET Framework since version 2.0. ClickOnce enables automatic deployment and update of WPF applications over the network from a deployment server. WPF Prism applications can use ClickOnce to get the shell, modules, and any other dependencies deployed to the client computer. The main challenge with Prism applications is that the Visual Studio publishing process for ClickOnce does not automatically include dynamically loaded modules in the published application.

Deploying a WPF application with ClickOnce is a two-step process. First, you have to publish the application from Visual Studio, and then you can deploy it to client computers. Publishing the application generates two manifests (a deployment manifest and an application manifest), and it copies the application files to a publish directory. That publish folder can then be moved to another server that may not be directly accessible from the developer computer to make the published application accessible to client computers from a known location and URL. Deploying an application to a client computer simply requires providing a URL or link that the user can navigate to. The URL points to the deployment manifest on the publishing deployment server. When that URL is loaded in the browser, ClickOnce on the client computer downloads the manifests and the application files specified by the manifests. After the files are downloaded and stored under the user profile, ClickOnce then launches the application. If subsequent updates are published to the deployment server, ClickOnce can automatically detect those updates, download, and apply them, or there are settings that allow you to detect and apply updates on demand or in the background after the application has launched.

When you publish a WPF Prism application that has dynamically loaded modules, the shell project will typically not have project references to the dynamically loaded modules. As a result, the published ClickOnce application manifest also does not include those module files, and if you deploy the application using ClickOnce, the client computer will not get the module files. To address this, you must modify the application manifest to include the module files that are not referenced by the shell application project.

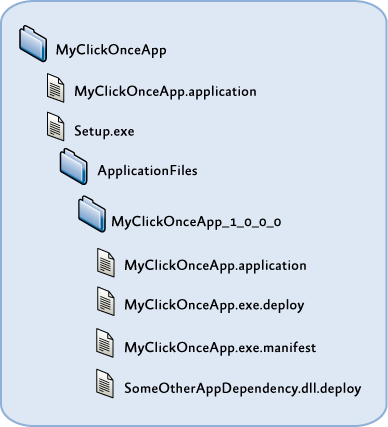
### ClickOnce Publishing Process

You can publish ClickOnce applications from Visual Studio 2010 using a .NET Framework SDK tool named the Manifest Generating and Editing tool (Mage) or a custom tool that uses the ClickOnce publishing APIs. Visual Studio exposes most of the capabilities needed for ClickOnce publishing. However, Visual Studio may not be available or desired for IT administrators who manage ClickOnce deployments on the server. Mage is designed to address most common administrative tasks for ClickOnce; it is a lightweight .NET Framework Windows-based application that can be given to your administrators. However, Mage requires too many detailed steps, performed in the correct order, to successfully complete common tasks such as modifying the application files listed in the application manifest. To make these tasks simpler, a custom utility is needed.

The Manifest Manager Utility sample utility demonstrates how to use the ClickOnce publishing API to manage deployment and application manifests in a simpler way. This utility is used for updating application manifest file lists and deployment manifest settings in a single user interface (UI) and its use is described in later sections in this chapter for initial deployment and update of a Prism application. The Manifest Manager Utility uses APIs exposed in the **Microsoft.Build.Tasks.Deployment** namespace to load, manipulate, and save modified manifest files for a ClickOnce deployment. You can download the Manifest Manager Utility from the Prism community site on Codeplex at<http://compositewpf.codeplex.com/releases/view/14771>. To learn the specific steps involved in publishing and updating a WPF Prism application that uses dynamic module loading, see the [WPF Prism Deployment Hands-On Lab: Publishing and Updating with ClickOnce](http://msdn.microsoft.com/en-us/library/gg405497(v=pandp.40)).

The following illustration shows the typical structure for a ClickOnce application publication, based on the way Visual Studio generates the deployment folders when you publish an application with ClickOnce. It includes a root folder for the application, which contains the default deployment manifest (.application file). The default deployment manifest usually points to the most recently published version when generated by Visual Studio, but it can be changed to point to whichever version the administrator chooses. The root folder also contains the Setup.exe bootstrapper, which allows you to deploy prerequisites for your application that might require an installer or executable to run before deploying the application using ClickOnce. There is then a subfolder for the application-specific files, under which you get a separate subfolder for each version that you publish. The publish version is a separate project setting and entry in the deployment manifest file for versioning the deployment as a whole, as opposed to the individual assembly versions of the contained assemblies. The publish version is used by ClickOnce to determine when there is an update available from a client that has already installed a ClickOnce application.

**ClickOnce publish folder structure**



Under each publish version's application files folder, you have another copy of the deployment manifest (.application file) that can be used to deploy specific versions to a client computer, or it can be copied to the root folder to cause a server-side rollback to a previous version. The application executable, in addition to any dependent libraries (such as Prism module assemblies) and resource files, will also be in this folder and will be automatically suffixed by a .deploy file name extension when published by Visual Studio. This is done to simplify the file extension mappings on the publishing web server so that you don't have to allow downloads of .dll, .exe, and a myriad of other potential file types that the application is composed of.

The application manifest (.exe.manifest) file is also contained in this folder and is referenced by the deployment manifest. It contains the list of files the application is composed of with hash values per file to assist in change detection; it also contains a list of permissions required by the application to run because ClickOnce can launch applications in a partial trust AppDomain if desired.

If you manually generate or update a ClickOnce application publication using either Mage or a custom tool, you are not constrained to this folder and file structure. For any particular ClickOnce publication, the chain of dependencies includes the following:

* It includes a deployment manifest that points to the application manifest through an embedded code base URL.
* It includes an application manifest that contains relative paths to each of the application files. These files must reside in the same folder or a subfolder from where the application manifest resides.

It includes the application files themselves, usually with a .deploy file name extension appended to the file name to simplify mapping these files to MIME types on the deployment server. ClickOnce automatically strips off the .deploy file name extension on the client side after the file is downloaded.

### ClickOnce Deployment and Update Process

The actual deployment of the application to a user via ClickOnce is almost always initiated by providing a URL or hyperlink to the deployment manifest of your published application on the deployment server. The user can click the hyperlink or enter the address in a browser, and the ClickOnce deployment process is invoked. After the manifest and application files are downloaded to the client computer, the application is launched. There are ClickOnce options that allow you to install the application during the initial deployment for offline use, or you can require the user to launch the application using the link or URL every time. When you publish a new version of the application to the deployment server, ClickOnce can automatically or manually check for updates and will download and apply the update for the next time the application launches.

# 更多

To learn how to use application library caching in Silverlight, see "How to: Use Application Library Caching" on MSDN:  
<http://msdn.microsoft.com/en-us/library/dd833069(VS.95).aspx>.

To download the Manifest Manager Utility from the Prism community site on Codeplex at<http://compositewpf.codeplex.com/releases/view/14771>.

To learn the specific steps involved in publishing and updating a WPF Prism application that uses dynamic module loading, see the [WPF Prism Deployment Hands-On Lab: Publishing and Updating with ClickOnce](http://msdn.microsoft.com/en-us/library/gg405497(v=pandp.40)).

# Appendix A: 词汇表

# Appendix B: Prism中的设计模式

# Appendix C: Prism库