**Dependency Injection/IOC/Prism**

[Dependency Injection 2](#_Toc362880523)

[Problem Statement 2](#_Toc362880524)

[Key Terms 2](#_Toc362880525)

[Putting It Together 3](#_Toc362880526)

[Examples from Videos/Slides 3](#_Toc362880527)

[Dependency Injection 4](#_Toc362880528)

[More Examples 5](#_Toc362880529)

[Comparison for DI Methodology 6](#_Toc362880530)

[IOC Container – What Does It Do 7](#_Toc362880531)

[Introduction to Unity 7](#_Toc362880532)

[Using Unity Steps 8](#_Toc362880533)

[1. Registration 8](#_Toc362880534)

[2. Instance Creation 9](#_Toc362880535)

[Resolved Object LifeTime 10](#_Toc362880536)

[Why not Use Factory Pattern instead of DI 11](#_Toc362880537)

[DI Caution 11](#_Toc362880538)

[Additional Information 12](#_Toc362880539)

[Links to Pursue 13](#_Toc362880540)

[To Do 13](#_Toc362880541)

[Prism 14](#_Toc362880542)

Dependency Injection

Problem Statement

1. Composition Relationship – one class references another class (and instantiates it)
2. Tempting to create instances of dependent classes in code

* Tightly couples you to implementation
* Difficult to test
* Beware of new keyword

1. Decoupling – how this helps to decouple code from architecture/testing perspective

Key Terms

Inversion of Control: IOC is a pattern used to invert interfaces, flow and dependencies

Dependency Injection: Implementation of IOC to invert dependencies

Inversion of Control Container: Framework to do dependency injection

Interface: Externally exposed way to interact with something

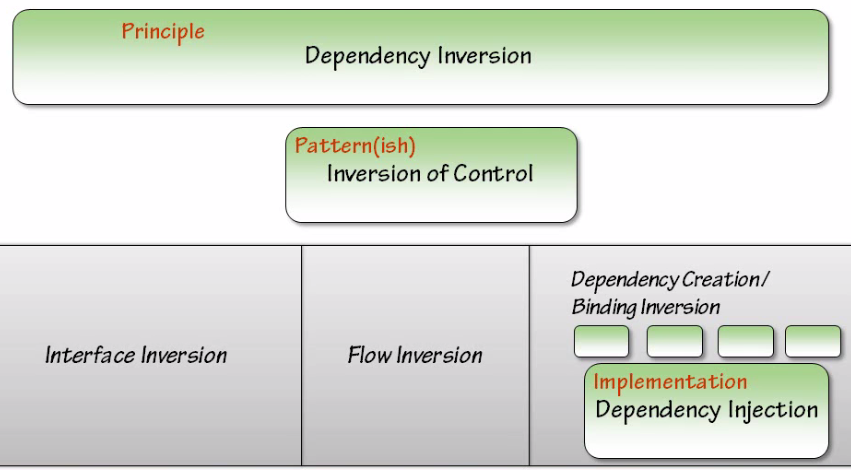
|  |  |
| --- | --- |
| **Dependency Not Inverted** | |
|  |  |
| **Dependancy Inverted** | |
|  |  |

Example

|  |  |  |
| --- | --- | --- |
|  |  |  |

Putting It Together

|  |  |
| --- | --- |
|  |  |
| Ven though we have decoupled MyClass from its concrete dependency still MyClass has to create/new the Dependency class within it. It may have a variable IDependency but still will do something like this:  IDependency aa = new new Dependency();  hence its not truly independent. | Dotted line is now elimiated. Injector knows about all the 3 things now – IDependancy, MyClass, Dependancy.  Injectors responsibility –   * instantiate myClass * instantiate Dependency * pass Dependency to MyClass (can be done in 4 ways) |



Creation Inversion: Creating objects outside of the class they are being used in (we are interested in this)

Examples from Videos/Slides

|  |  |  |
| --- | --- | --- |
| Example 1: | | |
|  |  |  |

|  |  |
| --- | --- |
| Example 2: | |
|  |  |

Dependency Injection

A type of IOC where we move the creation and binding of a dependency outside of the class that depends on it. There are **3 types** of DI

|  |  |
| --- | --- |
| 1. Constructer Injection 2. Setter Injection 3. Interface Injection |  |

|  |  |
| --- | --- |
| **Constructer Injection** | |
| 1. Instantiate your Dependency outside the class 2. When creating an instance of Shopper class (pass the dependency in) 3. The constructer will get it |  |

|  |  |
| --- | --- |
| **Setter Injection** | |
| 1. Create a setter on the dependent class 2. Use the setter to set the dependency |  |

|  |  |
| --- | --- |
| **Interface Injection** | |
| 1. Dependent class implements an interface 2. Injector uses the interface to set the dependency   (similar to setter injection partially) |  |

**Service Locater**: Your parent (main) object uses a locater which uses a place from where it can get the object of the child. Service Locater does not create a instance of child object (like Address) but it provides a methodology to register and then it goes and helps to find the services which create the object – like a proxy.

More Examples

Customer/Address Scenario –

|  |  |
| --- | --- |
| **Constructer Injection** | **Setter Injection** |
|  |  |
| **Interface Injection** | **Service Locater** |
|  |  |

From Training -

|  |  |
| --- | --- |
| **Constructer Injection** | **Setter Injection** |
|  |  |
| **Interface/Parameter Injection** | |
|  | |

Comparison for DI Methodology

Constructer Injection:

* This methodology is not suited for clients who can only use default constructers like Remoting – where u cannot have parameterized constructers.
* With constructer injection – as soon as the instance of the class is created – the Dependency is injected because constructer is always called.
* Can also have default constructor with concrete type
* Maintains interface if changing an existing code base

Setter Injection:

* It breaks encapsulation OOP because the objects are publicly exposed. It u are going to provide a public property by which you can set a object – anybody can come from the client and set it.
* With setter injection – It’s possible to create the class and also use the class – before the dependency is set through setter. Advantage: It gives some flexibility where u can change the dependency after the class is instantiated (swap out and put something else)
* Good for optional dependencies
  + Consider null pattern to remove null checks from code

Interface Injection: Pass dependency as method parameter and is useful for operation specific strategies.

IOC Container – What Does It Do

A container is an abstraction responsible for object management, instantiation and configuration. So you can configure the objects using the container rather than writing client code like factory patterns to implement object management.

In essence – container is a mid-man who does 3 things:

1. Will register address and customer objects as separate entity
2. Later the container creates the customer and address object
3. Last it injects the address object into customer

Introduction to Unity

* IOC container from Microsoft
* Part of Enterprise Library
* Also available as NuGet package
* Extendable
* Current Version: Unity 2.1 (for .Net 3.5 & 4.0) and Unity 3.0 (for .Net 4.5) (http://unity.codeplex.com/)



* You can use Nuget to get Unity dlls straight into your application:

|  |  |
| --- | --- |
|  |  |

* The dlls which can get added are as follows:

|  |  |
| --- | --- |
|  |  |

* Many IoC Containers available for .NET
  + Unity
  + Castle Windsor
  + StructureMap
  + Ninject
  + Spring.NET

Using Unity Steps

1. Registration

You can create mappings in 2 ways:

1. **Create Mappings in code**

|  |  |
| --- | --- |
| RegisterType: |  |
| RegisterInstance: | Can map abstract types to instances – Allows creation of objects with parameters only known at runtime |

1. **Create Mappings in configuration**

|  |  |
| --- | --- |
| Step 1 |  |
| Step 2 | Creating Actual Mappings in Config |
| Step 3 | Loading the config in code/Configuring the container |

Difference between RegisterType and RegisterInstance

RegisterType:

* The RegisterType method registers a type with the container. At the appropriate time, the container will build an instance of the type you specify. This could be in response to dependency injection initiated through class attributes or when you call the Resolve method.
* The lifetime of the object it builds will correspond to the lifetime you specify in the parameters of the method. If you do not specify a value for the lifetime, the type is registered for a transient lifetime—the container will create a new instance on each call to Resolve.

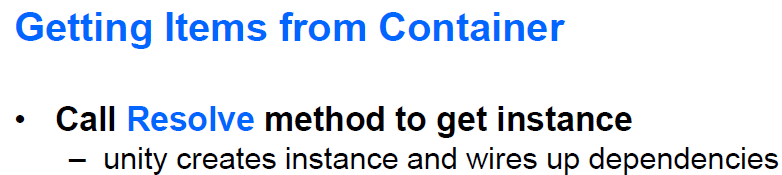




RegisterInstance:

* The RegisterInstance method registers with the container an existing instance of a type that you specify, with the lifetime that you specify. The container will return the exiting instance during that lifetime. If you do not specify a value for the lifetime, the instance will have the default container-controlled lifetime.
* Instance registration is similar to type registration, except that you first create the instance directly and then use the RegisterInstance method to add that instance to the container. Therefore, the container does not need to create the instance on the first Resolve request. These methods are useful if you already have an instance of an object you have previously configured.
* Unity contains default LifetimeManager types for singleton and instance registration.

1. Instance Creation



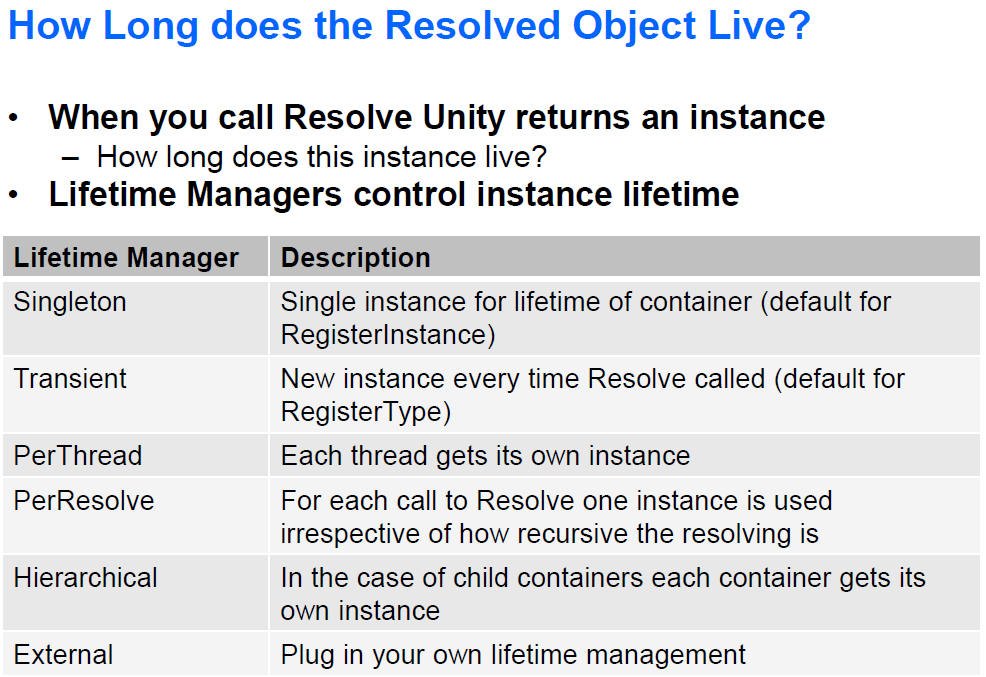


Resolved Object LifeTime

So when you do this – all the dependencies will get resolved and instances will be created. The below is the default behavior which is Transient (which means new instance every time)







Other Options

Default:



Singleton:



PerThread/PerResolve:



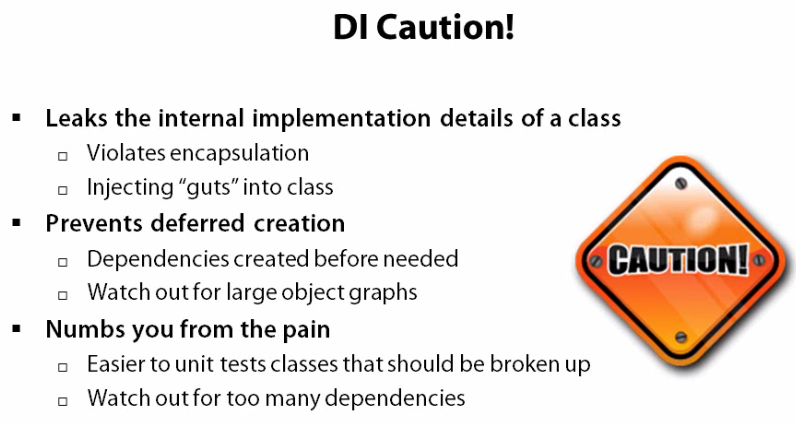
External: Some other resources controlling the lifetime of the object



Why not Use Factory Pattern instead of DI

1. Everything is hard-coded: The biggest issues with factory are it can not be reused across applications. All the options are hard-coded in the factory itself which makes the factory stringent to a particular implementation
2. Everything is compile time: All dependent objects for ana object in the factory have to be known at compile time
3. Factories are custom: They are very much custom to a particular implemenation
4. Interface Dependent – The base on which factories stands are common interfaces. Interfaces decouple the implementation and the object creation procedure. But then all the classes should implement a common interface. This is a limitation by itself again.

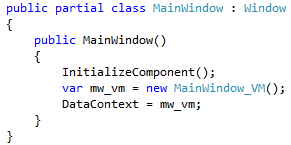
DI Caution



Practical Use-Case in WPF Project

In a normal MVVM project – you would create an instance of the MainWindowViewModel in the MainWindow.cs page



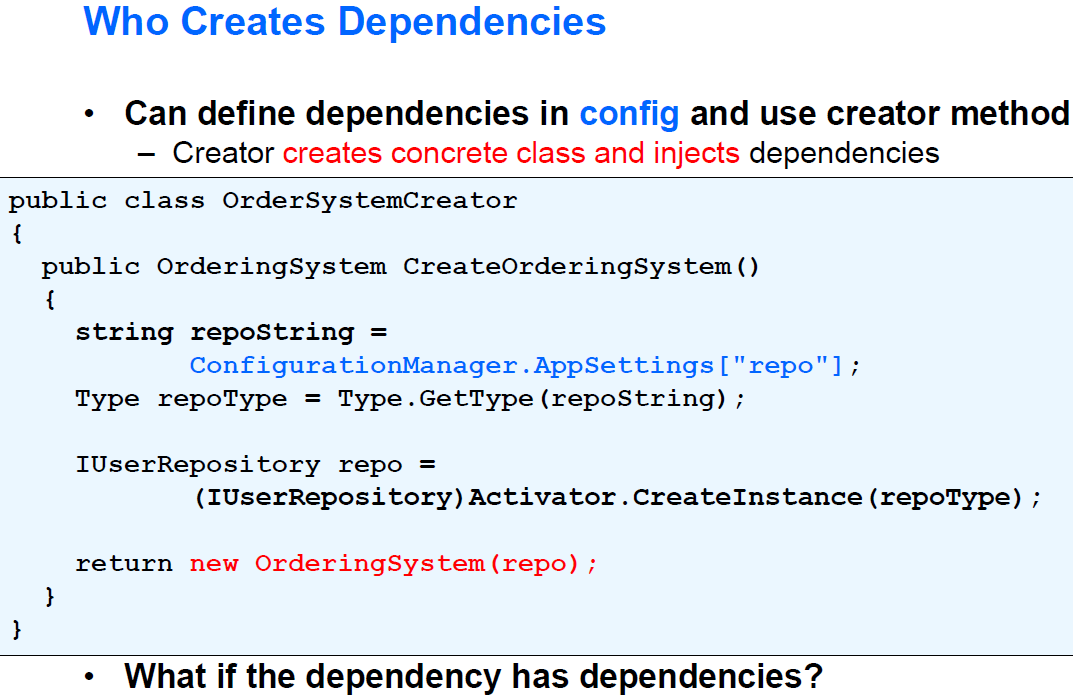


But if you want to use Unity Container to create an instance of MainWindowViewModel – then you need to the following 2 steps

|  |  |
| --- | --- |
| Step 1 -  Remove the startupuri from App.xaml page |  |
| Step 2 -  Override the OnStartup method on App.xamls.cs where you do multiple steps as shown |  |

Additional Information

1. Internal creation of Objects



1. Watch the following 2 videos for config based loading:

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=363&propid=7&saleid=subscription

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=364&propid=8&saleid=subscription

1. Videos where content is useful/taken from:

QuestPond DVD

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=25&propid=25&saleid=subscription

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=49&propid=9&saleid=subscription

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=363&propid=7&saleid=subscription

http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=364&propid=8&saleid=subscription

Pluralsight





Links to Pursue

Developer Code Samples

<http://code.msdn.microsoft.com/site/search?f%5B0%5D.Type=User&f%5B0%5D.Value=Srigopal%20Chitrapu&pageIndex>=2

Dependency Injection

<http://code.msdn.microsoft.com/Dependency-Injection-with-5702acaf>  
<http://martinfowler.com/articles/injection.html>  
<http://www.springframework.net/doc-latest/reference/html/quickstarts.html>

Container

<http://msdn.microsoft.com/en-us/library/ff921140(v=pandp.20)>

More Videos

<http://www.questpondvd.com/02456subcategquureutnpaotnhd9854.php?pp=286&propid=4&saleid=subscription>

Slot Machine video/code –

<http://rockycherry.net/blog/?p=117>

<http://unity.codeplex.com/wikipage?title=Webcast%20demos>

Articles -

<http://unity.codeplex.com/>

<http://www.sidarok.com/web/blog/content/2008/05/15/a-basic-hands-on-introduction-to-unity-di-container.html>

<http://stackoverflow.com/questions/2072094/are-there-good-tutorial-walkthroughs-for-unity-that-dont-use-configuration-file>

<http://www.c-sharpcorner.com/UploadFile/akkiraju/dependency-injection-techniques-explained-using-structurem/>

<http://www.refactorthis.net/post/2012/10/25/Dependency-Injection-and-Inversion-of-Control-Ioc-With-the-Microsoft-Unity-Container.aspx>

<http://stackoverflow.com/questions/4247022/design-for-cross-platform-classes-in-c-sharp?lq=1>

<http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html>

<http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection>

Mock-

Mocking with MOQ

<http://www.pluralsight.com/training/Courses/TableOfContents/mocking-with-moq>

Rhino Mock Fundamentals

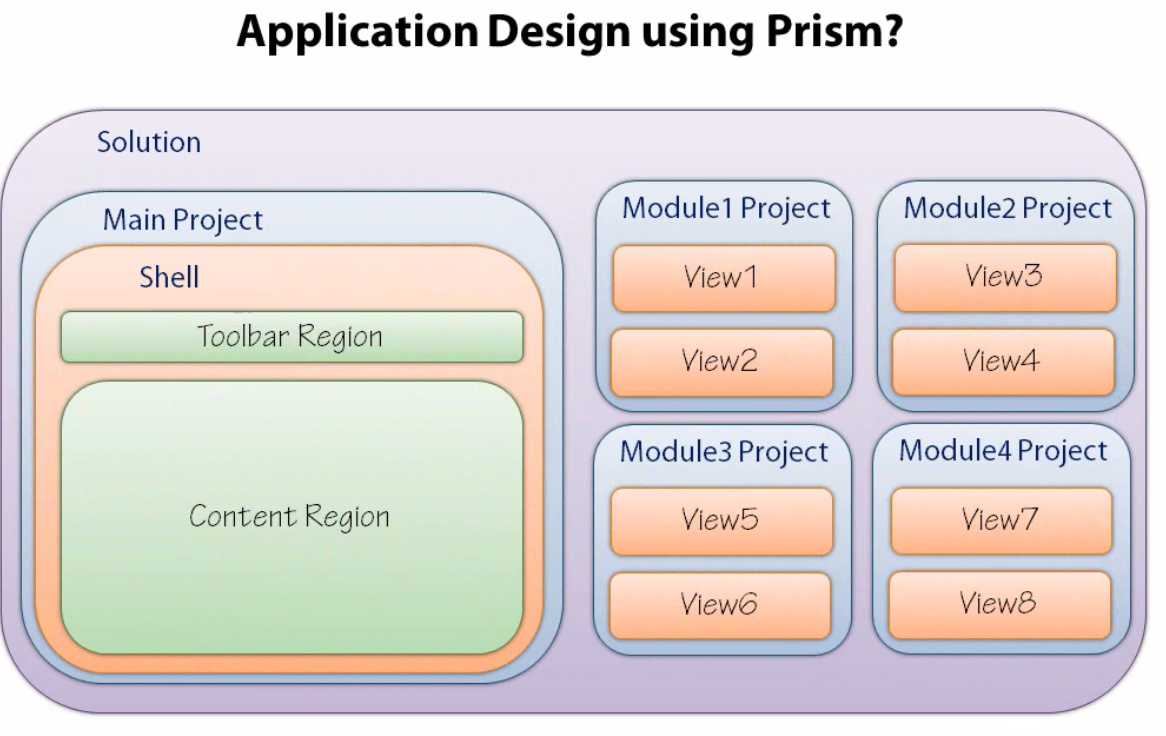
<http://www.pluralsight.com/training/Courses/TableOfContents/rhinomock-fundamentals>

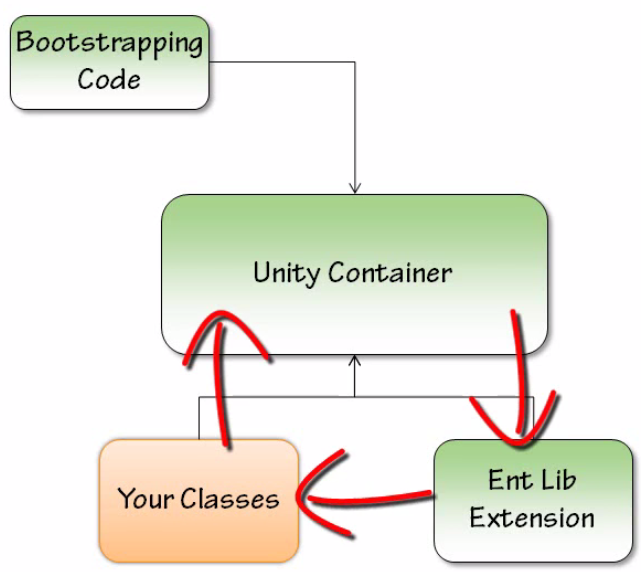
To Do

1. Practice the exercise from Training
2. Look up the code sample from Developer Code Samples above
3. Practice Setter Injection properly

Prism

First Prism Project - <http://msdn.microsoft.com/en-us/library/ff921141(v=pandp.20)>





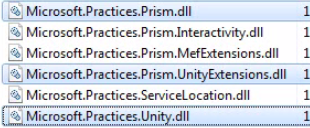
App.xaml

* Remove Startup uri from app.xaml
* Override the OnStartup method
  + You are creating an instance of BootStrapper Class and then the Run method is called

|  |  |
| --- | --- |
|  |  |

BootStrapper

1. Add References to the Prism Libraries

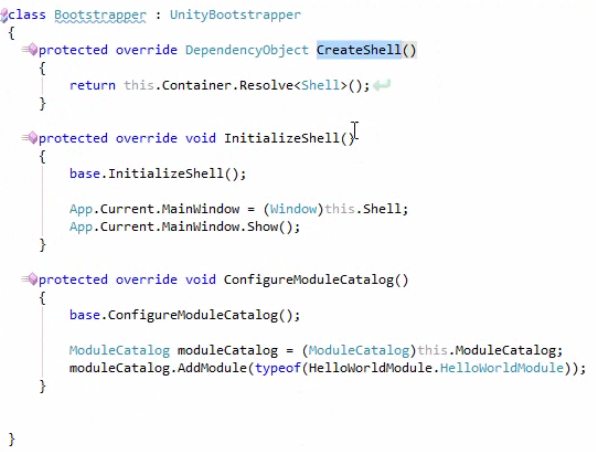


1. Is a class which is responsible for initializing our application
   1. Core Services – non application specific services that Prism provides

|  |  |
| --- | --- |
| * + 1. IModuleManager     2. IModuleCatalog     3. IModuleInitializer     4. IRegionManager     5. IEventAggregator     6. ILoggerFacade |  |

* 1. Application Specific Services –
     1. RSSFeedService (Example)

1. Ties Shell and Module together



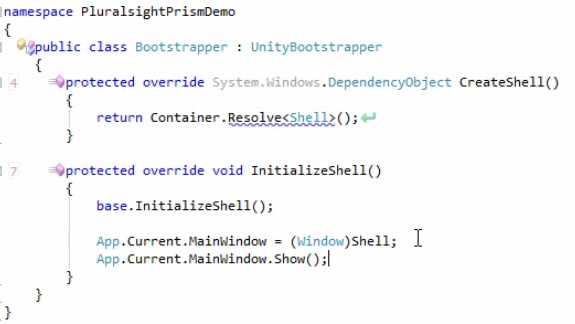
This is how we add our module to module catalog

Code to do in Bootstrapper

Step 1: override the CreateShell method

|  |  |
| --- | --- |
| You call the constainer.Resolve on Shell (root object) |  |

Step 2: Override the InitializeShell method



You have to cast it because a DependancyObject is returned in CreateShell



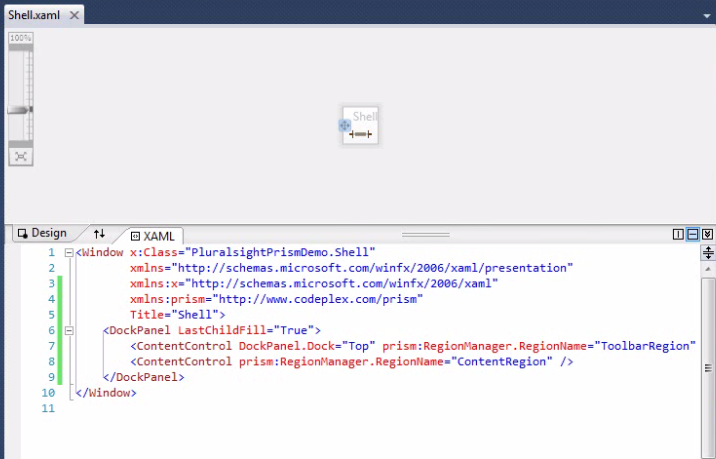
Shell.xaml

* Simply a main window/page
* Analogous to a ASP.Net master page
* Contain regions where views will be injected

This below line shows that Shell will have a region which is mentioned below:

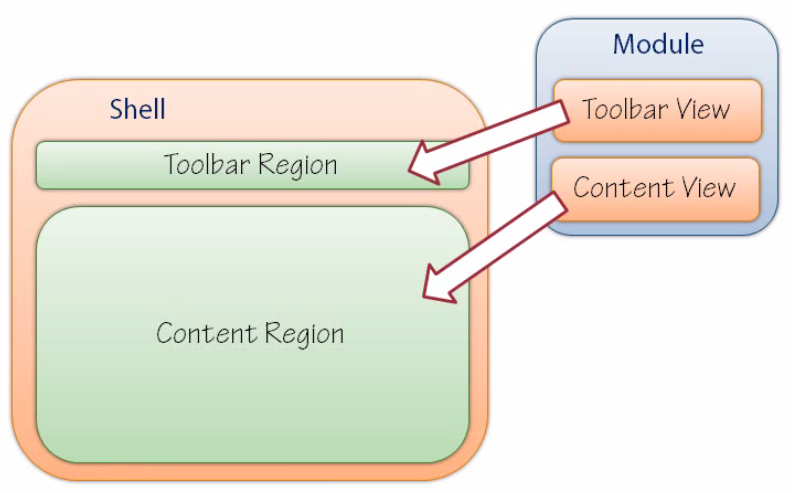


Another Example:



Regions

* “Placeholder” for dynamic content
* Shell will contain regions
* Each region should be explicitly named



* Region have no knowledge of views



* Region can be created by code or XAML
* Region is applied to a control through an attached property via RegionName attached property via RegionManager
* Implements IRegion

Region Manager

* Maintains collections of regions
* Provides a RegionName attached property

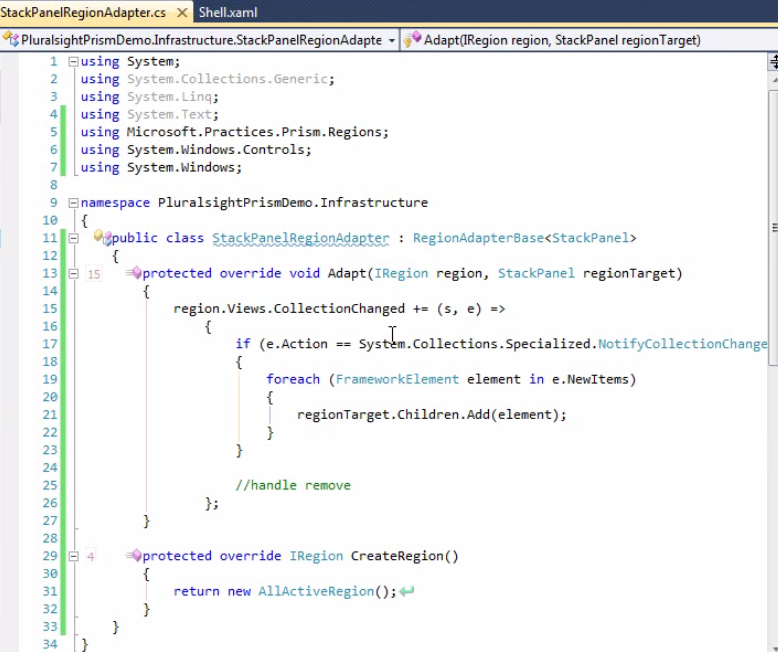


* Maps RegionAdapter to controls – Prism provides 4 Region Adapters
* ContentControlRegionAdaptor
* ItemsControlRegionAdaptor
* SelectorREgionAdaptor
* TabControlRegionAdapter (Silverlight only)
* Provides a REgionContext attached property (share data between parent view and child view)

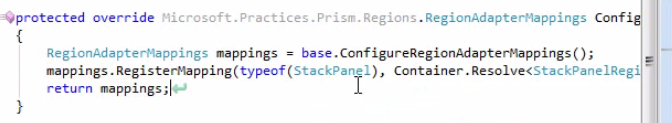
If the above 4 adapters are not sufficient – then you can create your own RegionAdapter



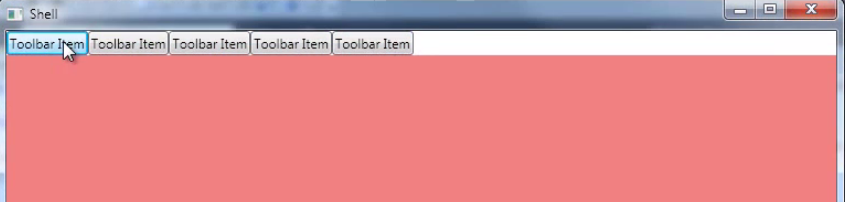
First 3 steps screenshot



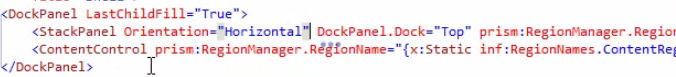
4th Step – registering the adapter. You have to do that in bootstrapper



Now it will work



So the shell will have something like this



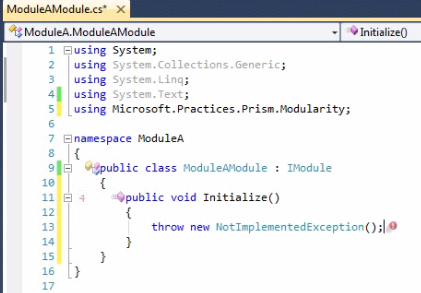
So instead of the standard 4 regionadapters – you have now created your own RegionAdapter StackPanel with Horizontal Orientation

Modules

3 steps:

* Registering Modules
* Loading Modules
* Initialize Module

When you inherit from IModule Interface – helps Prism identify it as a module. Contains a single Initialize method



Registering Modules

All the modules which needed to be loaded at runtime, they have to be registered with ModuleCatalog.

So you have register every module with ModuleCatalog

Loading Modules

* + - When available
    - OnDemand

Initialize Module

This is done in BootStrapper.

Multiple ways to initialize a Module

1. Within Code
2. From Directory
3. From XAML
4. From app.config
5. Register and Load Module From Code





1. Register and Load Module From Directory

|  |  |
| --- | --- |
|  | Inside the Module cs file |

1. Register and Load Module from XAML file

|  |  |
| --- | --- |
|  | Create a ResourceDictionary File and change tags |

1. Register and Load Module from app.config

|  |  |
| --- | --- |
|  | Create app.config and then type following: |

How is module initialized

* Initialized method is called and that registers the View HelloWorldView with Main Region

|  |  |
| --- | --- |
|  |  |

Another Example

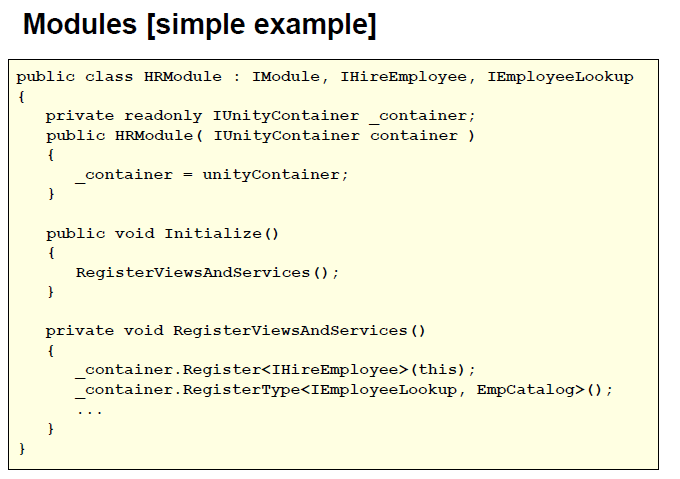
|  |  |
| --- | --- |
| This registers ToolbarView with Toolbar Region and ContentView with ContentRegion |  |

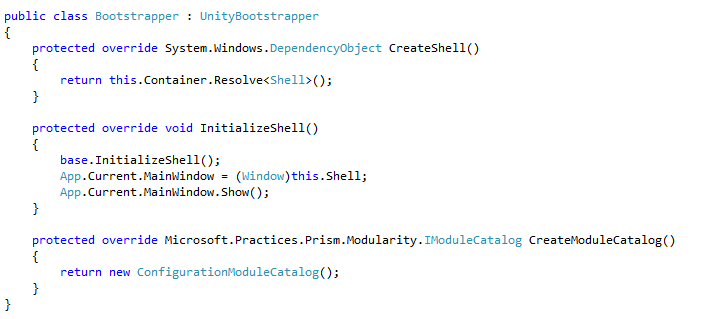
Above you saw that module and Region are hooked up (when you register)

Region eventually injects into the view eventually



Another example







Links

<http://compositewpf.codeplex.com/>

Fakes and Mocks

<http://en.wikipedia.org/wiki/Mock_object>