PRISM 4.0 TRAINING KIT

Hands-On Lab

Dependency Injection

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277956366)

[Exercise 1 – Creating and registering the service 4](#_Toc277956367)

[Task 1 – Creating a Service 4](#_Toc277956368)

[Task 2 –Registering the Service 4](#_Toc277956369)

[Exercise 2 – Consuming a Service 6](#_Toc277956370)

[Task 1- Consuming the UIMessagesService 6](#_Toc277956371)

[Task 2- Decoupling Modules 8](#_Toc277956372)

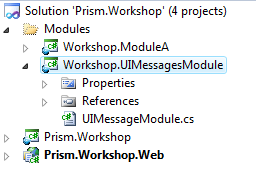
Overview

* 1. Applications based on the Prism Library are composite applications that potentially consist of many loosely coupled types and services. They need to interact to contribute content and receive notifications based on user actions. Because they are loosely coupled, they need a way to interact and communicate with one another to deliver the required business functionality.
  2. To tie together these various pieces, applications based on the Prism Library rely on a dependency injection container. Dependency injection containers reduce the dependency coupling between objects by providing a facility to instantiate instances of classes and manage their lifetime based on the configuration of the container. During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first. In some cases, the container itself is resolved as a dependency. For example, when using the Unity Application Block (Unity) as the container, modules have the container injected, so they can register their views and services with that container.

Exercise 1 – Creating and registering the service

* 1. In this exercise you will learn how to create a service and register it in a dependency injection container, in this case Unity.

Task 1 – Creating a Service

* 1. Open the Prism.Workshop.sln solution located in the \Dependency Injection\Exercise 1\Begin\ folder.
  2. Add a new module project to the solution, named **Workshop.UIMessagesModule**.For more information see the Modularity Hands-on-Lab located in **\Modularity\Modularity-HOL.docx**.
     1. 
     2. Solution Structure after adding Workshop.UIMessagesModule module
  3. Create the **UIMessagesService**. Add a new UIMessagesService class to the project using the following code.
     1. C#
     2. using System.Windows;
     3. namespace Workshop.UIMessagesModule.Services
     4. {
     5. public class UIMessagesService
     6. {
     7. public void ShowMessage(string message)
     8. {
     9. // Show a message
     10. MessageBox.Show(message);
     11. }
     12. }
     13. }

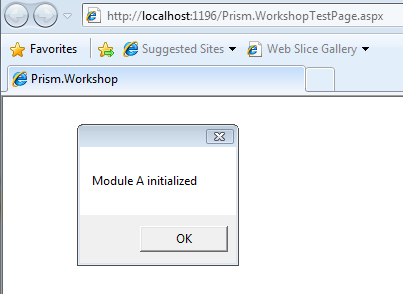
Task 2 –Registering the Service

* 1. Add the following references to the **Workshop.UIMessagesModule** project:
     + Unity assembly, located at **\Lib\Microsoft.Practices.Unity.Silverlight.dll**.
     + Prism assembly, located at **\Lib\ Microsoft.Practices.Prism.dll**.
  2. Add the following using statements to the **UIMessagesModule** class.
     1. C#
     2. using Microsoft.Practices.Unity;
     3. using Workshop.UIMessagesModule.Services;
     4. using Microsoft.Practices.Prism.Modularity;
  3. Use Property Injection to get an instance of the container. Add the following code to the **UIMessagesModule** class.
     1. C#
     2. /// <summary>
     3. /// DI Container.
     4. /// Unity will inject an instance of the container. IUnityContainer is registered in the container in the bootstrapper of the application.
     5. /// </summary>
     6. [Dependency]
     7. public IUnityContainer Container { get; set; }
  4. Register the service in the container as a singleton when the module is initialized. To do this, use the **Register** method. In the **UIMessageModule** implement the initialize method as illustrated in the following code.
     1. C#
     2. public void Initialize()
     3. {
     4. // register the UIMessagesService as singleton
     5. this.Container.RegisterType<UIMessagesService>(new ContainerControlledLifetimeManager());
     6. }
     7. **Note:** You can debug the application and use a Break Point on the initialize method to verify that the Container property is being set with an instance of **UnityContainer**.
  5. (Optional) Unity also allows the use of Constructor Injection for achieving the same results. To get an instance of the Unity Container using Constructor Injection, remove the previous method and property, and add the following code in the **UIMessagesModule** class.
     1. C#
     2. private readonly IUnityContainer container;
     3. /// <summary>
     4. /// Constructor for the UIMessagesModule.
     5. /// Unity will inject an instance of the container. IUnityContainer is registered in the container in the bootstrapper of the application.
     6. /// </summary>
     7. /// <param name="container"></param>
     8. public UIMessagesModule(IUnityContainer container)
     9. {
     10. this.container = container;
     11. }
     12. public void Initialize()
     13. {
     14. // register the UIMessagesService as singleton
     15. this.container.RegisterType<UIMessagesService>(new ContainerControlledLifetimeManager());
     16. }

Exercise 2 – Consuming a Service

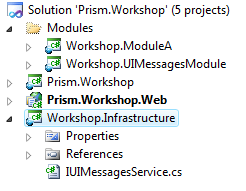
* 1. In this exercise, you will learn how to obtain a reference to a registered service and use it in your application.

Task 1- Consuming the UIMessagesService

* 1. Open the solution located in \Dependency Injection\Exercise 2\Begin\Prism.Workshop.sln
  2. In the **Workshop.ModuleA**, add a reference to the **Workshop.UIMessagesService** project. This reference will be removed in the last task.
  3. Add the following references to the **Workshop.UIMessagesModule** project:
     + Unity assembly located at **\Lib\Microsoft.Practices.Unity.Silverlight.dll**.
     + Prism assembly located at **\Lib\ Microsoft.Practices.Prism.dll**.
  4. In the **ModuleA** class add the following using statements:
     1. C#
     2. using System.Windows.Controls;
     3. using Microsoft.Practices.Prism.Modularity;
     4. using Microsoft.Practices.Prism.Regions;
     5. using Workshop.Infrastructure;
  5. Use Constructor Injection to get an instance of the container. In the **ModuleA** class, add the following code:
     1. C#
     2. public ModuleA(IRegionManager regionManager, IUnityContainer container)
     3. {
     4. this.regionManager = regionManager;
     5. }
  6. Use the container to obtain an instance of the **UIMessageService.** To do this,use the **Register** method as follows**:**
     1. C#
     2. private readonly UIMessagesService uiMessagesService;
     3. public ModuleA(IRegionManager regionManager, IUnityContainer container)
     4. {
     5. this.regionManager = regionManager;
     6. // Get an instance of the UIMessageService
     7. this.uiMessagesService = container.Resolve<UIMessagesService>();
     8. }
  7. To use the **UIMessageService**, add the following lines of code to the initialize the method.
     1. C#
     2. // Use the UIMessagesService to show a message
     3. this.uiMessagesService.ShowMessage("Module A initialized");
  8. (Optional) Unity allows you to inject registered services directly, instead of using the container. To get an instance of the **UIMessagesService** using Constructor Injection replace the **ModuleA** constructor with the following code.
     1. C#
     2. /// <summary>
     3. /// Constructor for Module A
     4. /// </summary>
     5. /// <param name="regionManager">Unity will inject an instance of the container. IRegionManager is registered in the container in the bootstrapper of the application.</param>
     6. /// <param name="uiMessagesService">Unity will inject an instance of the container. The UIMessagesService is registered by the UIMessagesModule.</param>
     7. public ModuleA(IRegionManager regionManager, UIMessagesService uiMessagesService)
     8. {
     9. this.regionManager = regionManager;
     10. // Get an instance of the UIMessageService
     11. this.uiMessagesService = uiMessagesService;
     12. }
     13. 
     14. Message Shown using the UIMessageService

Task 2- Decoupling Modules

ModuleA module has a reference to **UIMessageModule** module. To decouple these modules perform the following steps.

* 1. Create a new infrastructure project named **Workshop.Infrastructure**
     1. 
     2. Solution structure adding an infrastructure project.
  2. Add a new interface named **IUIMessageService** using the following code.
     1. C#
     2. namespace Workshop.Infrastructure
     3. {
     4. public interface IUIMessagesService
     5. {
     6. void ShowMessage(string message);
     7. }
     8. }
  3. In the **Workshop.UIMessageService** project, add a reference to the **Workshop.Infrastructure** project.
  4. Make the **UIMessageService** implement the **IUIMessageService** interface. In the **UIMessageService** class add bold code:
     1. C#
     2. using System.Windows;
     3. **using Workshop.Infrastructure;**
     4. namespace Workshop.UIMessagesModule.Services
     5. {
     6. public class UIMessagesService **: IUIMessagesService**
     7. {
     8. public void ShowMessage(string message)
     9. {
     10. // Show a message
     11. MessageBox.Show(message);
     12. }
     13. }
     14. }
  5. Register the service using the interface. On the **Initialize** method of the **UIMessagesModule** replace the service registration line as follows.
     1. C#
     2. // register the UIMessagesService as singleton
     3. this.Container.**RegisterType<IUIMessagesService, UIMessagesService>**(new ContainerControlledLifetimeManager());
  6. In the **Workshop.ModuleA** project, add a reference to the **Workshop.Infrastructure** project.
  7. Make the **ModuleA** have a dependency on the service interface instead of the concrete class.   
     To do this, change the the ModuleA constructor as follows:
     1. C#
     2. using Microsoft.Practices.Prism.Modularity;
     3. using Microsoft.Practices.Prism.Regions;
     4. using Workshop.UIMessagesModule.Services;
     5. **using Workshop.Infrastructure;**
     6. namespace Workshop.ModuleA
     7. {
     8. public class ModuleA : IModule
     9. {
     10. private readonly IRegionManager regionManager;
     11. private readonly IUIMessagesService uiMessagesService;
     12. /// <summary>
     13. /// Constructor for Module A
     14. /// </summary>
     15. /// <param name="regionManager">Unity will inject an instance of the container. IRegionManager is registered in the container in the bootstrapper of the application.</param>
     16. /// <param name="uiMessagesService">Unity will inject an instance of the container. The UIMessagesService is registered by the UIMessagesModule.</param>
     17. public ModuleA(IRegionManager regionManager, **IUIMessagesService uiMessagesService**)
     18. {
     19. this.regionManager = regionManager;
     20. // Get an instance of the UIMessageService
     21. this.uiMessagesService = uiMessagesService;
     22. }
     23. …
     24. }
  8. Make **ModuleA** have a dependency on the **UIMessagesService**, by adding the following code to the **ModuleCatalog.xaml**.
     1. XAML
     2. <Modularity:ModuleInfo Ref="Workshop.ModuleA.xap" ModuleName="ModuleA" ModuleType="Workshop.ModuleA.ModuleA, Workshop.ModuleA, Version=1.0.0.0" >
     3. **<Modularity:ModuleInfo.DependsOn>**
     4. **<sys:String>UIMessagesModule</sys:String>**
     5. **</Modularity:ModuleInfo.DependsOn>**
     6. </Modularity:ModuleInfo>
  9. On the **Workshop.ModuleA** project, remove the reference to **Workshop.UIMesssagesService**.