PRISM 4.0 TRAINING KIT

Hands-On Lab

UI Composition

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277956497)

[Exercise 1 – Showing a View in a Region using ViewInjection 4](#_Toc277956498)

[Task 1 – Creating a View using Model-View-Presenter and Presenter-First approach 4](#_Toc277956499)

[Task 2 – Show the view in a Region 7](#_Toc277956500)

[Exercise 2 - Showing a View in a Region Using ViewDiscovery 10](#_Toc277956501)

[Task 1 - Adding a new region to the MainView 10](#_Toc277956502)

[Task 2 – Register the view in the region 11](#_Toc277956503)

[Task 3 – Wiring up the List and Details views using a Controller 12](#_Toc277956504)

[Exercise 3 - Showing a view in a Scoped Region 18](#_Toc277956505)

[Task 1 – Wiring the providers details view 18](#_Toc277956506)

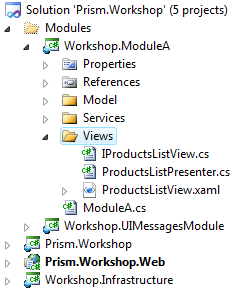
[Task 2 – Using scoped regions 23](#_Toc277956507)

Overview

* 1. 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.
  2. 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.

Exercise 1 – Showing a View in a Region using ViewInjection

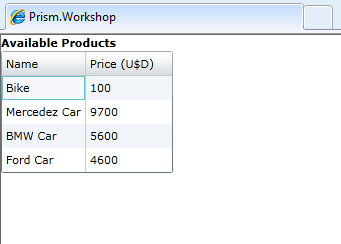
Task 1 – Creating a View using Model-View-Presenter and Presenter-First approach

* 1. In this task you will create a view and its corresponding presenter using the Presenter-First approach.
  2. Open the solution located at **\UIComposition\Exercise 1\Begin\Prism.Workshop.sln.**
  3. Add a new view to the **ModuleA** project, inside the **Views\ProductsView** folder. You will need to create a new folder named **Views** and add a UserControl named **ProductsListView.xaml.**
  4. Add a Presenter class named **ProductsListPresenter** to the **ModuleA** project inside the **Views\ProductsView** folder.
  5. Add a new Interface named **IProductsView** to the **ModuleA** project inside the **Views\ProductsView**. This interface will be used to communicate between the view and the presenter. The solution structure should be similar to the following.
     1. 
     2. Solution structure for the ProductList using Model-View-Presenter pattern.
  6. Implement the **ProductsListView** view. Add the following using statements to the **IProductListView.cs** file:
     1. C#
     2. using System.Collections.ObjectModel;
     3. using Workshop.ModuleA.Model;
  7. Add an "**ObservableCollection** of **Product**" property to the view’s interface as shown in the following code.
     1. C#
     2. ObservableCollection<Product> Model { get; set; }
  8. Add the following using statements to the **ProductListView.cs** file.
     1. C#
     2. using System.Collections.ObjectModel;
     3. using System.Windows.Controls;
     4. using Workshop.ModuleA.Model;
  9. Open the **ProductListView** code behind file, and update the class signature to implement the **IProductListView** interface as shown in the following code snippet.
     1. C#
     2. public partial class ProductsListView : UserControl, **IProductsListView**
     3. {
     4. public ProductsListView()
     5. {
     6. InitializeComponent();
     7. }
     8. }
  10. Add a **Model** property to the **ProductListView** code behind that sets the DataContext in its setter, as show in the following code.
      1. C#
      2. public ObservableCollection<Product> Model
      3. {
      4. get { return this.DataContext as ObservableCollection<Product>; }
      5. set { this.DataContext = value; }
      6. }
  11. Add a reference to the **System.Windows.Controls.Data** assembly in the **Workshop.ModuleA** project.
  12. Open the **ProductsListView.xaml** file and update its code as illustrated in the following code.
      1. XAML
      2. <UserControl x:Class="Workshop.ModuleA.Views.ProductsListView"
      3. xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      4. xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      5. xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      6. xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      7. xmlns:Controls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"
      8. mc:Ignorable="d"
      9. d:DesignHeight="300" d:DesignWidth="400">
      10. <StackPanel>
      11. <TextBlock Width="Auto" Height="Auto" Text="Available Products" Padding="0,0,0,0" FontWeight="Bold" Margin="0,0,0,0" HorizontalAlignment="Left" VerticalAlignment="Top"/>
      12. <Controls:DataGrid x:Name="EmployeesList" ItemsSource="{Binding}" AutoGenerateColumns="False" IsReadOnly="True">
      13. <Controls:DataGrid.Columns>
      14. <Controls:DataGridTextColumn Header="Name" Binding="{Binding Path=Name}" />
      15. <Controls:DataGridTextColumn Header="Price (U$D)" Binding="{Binding Path=Price}" />
      16. </Controls:DataGrid.Columns>
      17. </Controls:DataGrid>
      18. </StackPanel>
      19. </UserControl>
  13. Implement the **ProductsListPresenter**. To do this, open the **ProductsListPresenter.cs** file and update its implementation as shown in the following code.
      1. C#
      2. public class ProductsListPresenter
      3. {
      4. public ProductsListPresenter(IProductsListView view, IProductsService productsService)
      5. {
      6. this.View = view;
      7. // retrieve products list
      8. var products = productsService.GetProducts();
      9. // set the view's model
      10. this.View.Model = products;
      11. }
      12. public IProductsListView View { get; set; }
      13. }
      14. Notice that the **ProductsListPresenter** constructor has a dependency on the **IProductsListView** interface and a dependency on the **IProductsService** interface**.** As both dependencies are interfaces it is easier to create unit tests for the presenter class.
  14. Configure the container to register the product’s view and service. Open the **ModuleA.cs** file and update the **Initialize** method as shown in the following code.
      1. C#
      2. public void Initialize()
      3. {
      4. TextBlock textBlock = new TextBlock();
      5. textBlock.Text = "I'm in the Shell";
      6. this.regionManager.Regions["MainRegion"].Add(textBlock);
      7. // Use the UIMessagesService to show a message
      8. this.uiMessagesService.ShowMessage("Module A initialized");
      9. **// Register the neccessary services**
      10. **this.RegisterServices();**
      11. **// Register the neccessary views**
      12. **this.RegisterViews();**
      13. }
      14. **private void RegisterViews()**
      15. **{**
      16. **// Register the product list view in the container.**
      17. **this.container.RegisterType<IProductsListView, ProductsListView>();**
      18. **}**
      19. **private void RegisterServices()**
      20. **{**
      21. **// Register the products service in the container as Singleton.**
      22. **this.container.RegisterType<IProductsService, ProductsService>(new ContainerControlledLifetimeManager());**
      23. **}**
  15. Build the solution.

Task 2 – Show the view in a Region

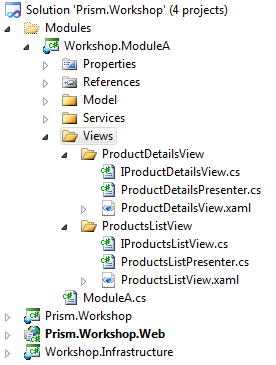
* 1. In this task you will show the view created in the previous task in a region.
  2. Make **ModuleA** receive the container in its constructor. Update **ModuleA** constructor to receive the container as a parameter, and store it in private field, as shown in the following code.
     1. C#
     2. **private readonly IUnityContainer container;**

public ModuleA(IRegionManager regionManager, IUIMessagesService uiMessagesService, **IUnityContainer container**)

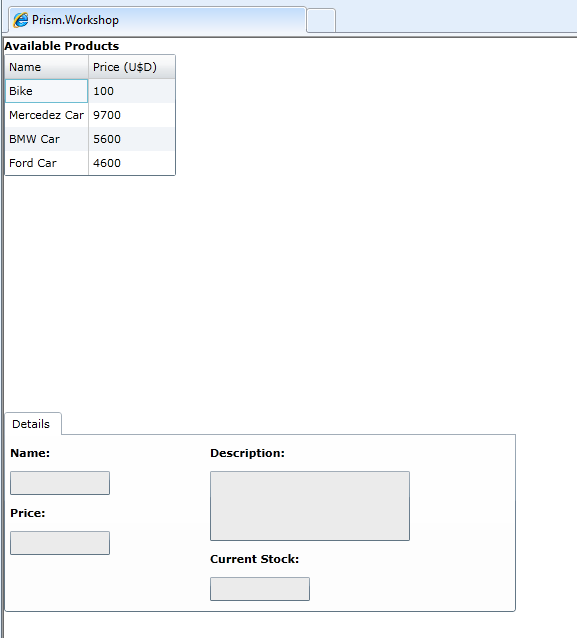
* + 1. {
    2. this.regionManager = regionManager;
    3. **this.container = container;**
    4. // Get an instance of the UIMessageService
    5. this.uiMessagesService = uiMessagesService;
    6. }
  1. Obtain an instance of the **ProductsListView** view using the container. On the Initialize method add the following line of code.
     1. C#
     2. // Get an instance of the ProductsListPresenter using the container
     3. ProductsListPresenter presenter = this.container.Resolve<ProductsListPresenter>();
  2. Obtain the main region using the **RegionManager**. To do this, on the Initialize method add the following line of code.
     1. C#
     2. // Get the main region
     3. IRegion mainRegion = this.regionManager.Regions["MainRegion"];
  3. Add the view to the main region using ViewInjection. Update the Initialize method as shown in the following code.
     1. C#
     2. public void Initialize()
     3. {
     4. // Use the UIMessagesService to show a message
     5. this.uiMessagesService.ShowMessage("Module A initialized");
     6. // Register the neccessary services
     7. this.RegisterServices();
     8. // Register the neccessary views
     9. this.RegisterViews();
     10. // Get an instance of the ProductsListPresenter using the container
     11. ProductsListPresenter presenter = this.container.Resolve<ProductsListPresenter>();
     12. // Get the main region
     13. IRegion mainRegion = this.regionManager.Regions["MainRegion"];
     14. **// Add the Product List view to the main region (View Injection)**
     15. **mainRegion.Add(presenter.View);**
     16. }
  4. Compile and run the solution.
     1. 
     2. Products List View

Exercise 2 - Showing a View in a Region Using ViewDiscovery

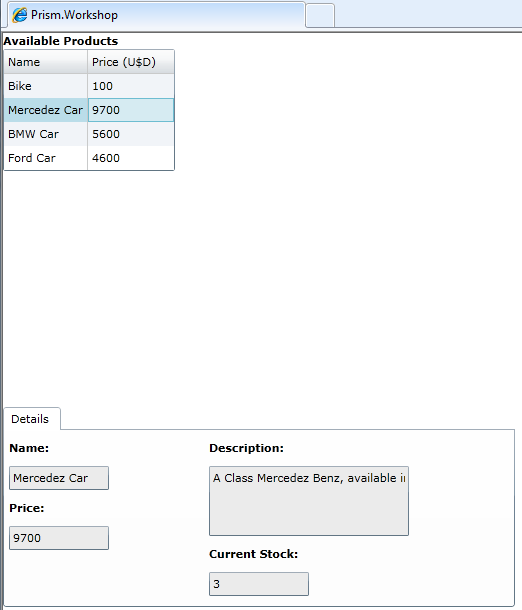
Task 1 - Adding a new region to the MainView

* 1. In this task you will create a region in the MainView, that will hold other views.
  2. Open the solution located at **\UIComposition\Exercise 2\Begin\Prism.Workshop.sln.**Notethatthissolution contains an implementation of the **ProductsDetailsView** view, which can be constructed in a similar way as the **ProductsListView** in the previous exercise.
     1. 
  3. Open the **Shell.xaml** file of **the Prism.Workshop** project, and add row definitions and a **ContentControl** to hold the new region, named **BottomRegion**, as shown in the following code.
     1. XAML
     2. <Grid x:Name="LayoutRoot" Background="White">
     3. **<Grid.RowDefinitions>**
     4. **<RowDefinition/>**
     5. **<RowDefinition/>**
     6. **</Grid.RowDefinitions>**
     7. <ContentControl **Grid.Row="0"** regions:RegionManager.RegionName="MainRegion"/>
     8. <ContentControl Grid.Row="1" regions:RegionManager.RegionName="**BottomRegion**"/>
     9. </Grid>

Task 2 – Register the view in the region

* 1. In this task you will register the view in the region created in the previous step.
  2. Update the **RegisterViews** method on the **ModuleA** class to register the **ProductDetailsView** in the container, as shown in the following code.
     1. C#
     2. private void RegisterViews()
     3. {
     4. // Register views in the container.
     5. this.container.RegisterType<IProductsListView, ProductsListView>();
     6. **this.container.RegisterType<IProductDetailsView, ProductDetailsView>();**
     7. }
  3. Update the **Initialize** method on the **ModuleA** class to register the **ProductDetailsView** into the details region. The following code shows how to achieve this using ViewDiscovery:
     1. C#
     2. // Add the Details List view to the details region (View Discovery)
     3. this.regionManager.RegisterViewWithRegion("BottomRegion",
     4. () => this.container.Resolve<ProductDetailsPresenter>().View);
     5. Note that the **RegisterViewWithRegion** is an extension method, which uses the **RegionViewRegistry** to register view into regions. You could also use the **RegionViewRegistry** directly as in the following code.
     6. C#
     7. // Can also be done using the RegionViewRegistry as follows: this.container.Resolve<IRegionViewRegistry>().RegisterViewWithRegion("BottomRegion", () => this.container.Resolve<ProductDetailsPresenter>().View);
  4. Compile and run the solution. The details view will not show any information, as the **ProductDetailsPresenter** is never set with a product. The list and the details view will be wired in the next task.
     1. 
     2. ProductListView and ProductDetailsView

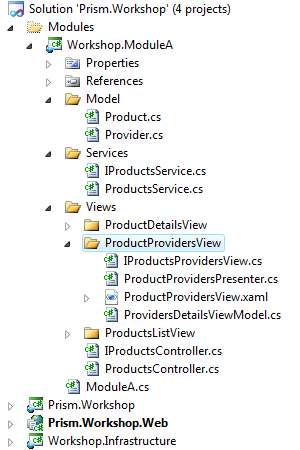
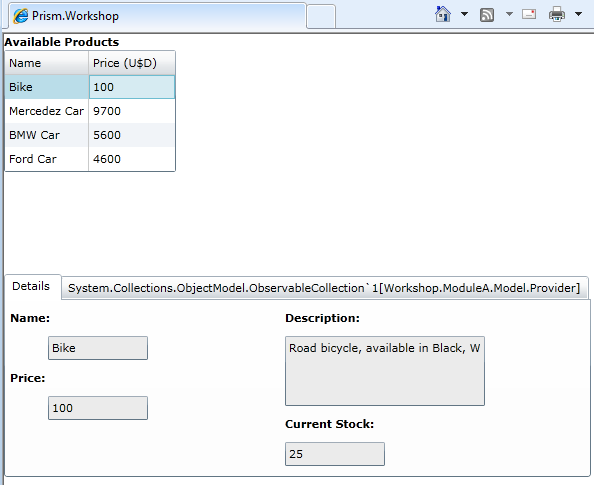
Task 3 – Wiring up the List and Details views using a Controller

* 1. In this task, you will wire the list view and the details view to display the details of the select product.
  2. Create an interface for the Controller. Inside the **Views** folder create an interface named **IProductsController**. Add the **OnProductSelected** method to the interface, as shown in the following code.
     1. C#
     2. using Workshop.ModuleA.Model;
     3. namespace Workshop.ModuleA.Views
     4. {
     5. public interface IProductsController
     6. {
     7. void OnProductSelected(Product product);
     8. }
     9. }
  3. Create the Controller class. Inside the **Views** folder create a class named **ProductsController** that implements the **IProductsController** interface as shown in the following code.
     1. C#
     2. using Workshop.ModuleA.Model;
     3. namespace Workshop.ModuleA.Views
     4. {
     5. public class ProductsController : IProductsController
     6. {
     7. public void OnProductSelected(Product product)
     8. {
     10. }
     11. }
     12. }
  4. Register the Controller in the container. In the **RegisterView** method of the **ModuleA** class add the following line of code.
     1. C#
     2. // Register products controller
     3. this.container.RegisterType<IProductsController, ProductsController>();
  5. Make the **ProductsListPresenter** receive a Controller instance in its constructor. Update the **ProductsListPresenter**  constructor to receive a **IProductsController** parameter and store it in a private field, as shown in the following code.
     1. C#
     2. private readonly IProductsController productsController;
     3. public ProductsListPresenter(IProductsListView view, IProductsService productsService**, IProductsController productsController**)
     4. {
     5. this.View = view;
     6. **this.productsController = productsController;**
     7. ...
     8. }
  6. Make the **ProductsListView** notify when a product is selected. You will need to add a **ProductSelected** event in the **ProductsListView.cs** file and in the **IProductsListView** interface, as shown in the following code.
     1. C# - ProductsListView.cs
     2. public event EventHandler<DataEventArgs<Product>> ProductSelected = delegate { };
     3. C# - IProductsListView.cs
     4. event EventHandler<DataEventArgs<Product>> ProductSelected;
  7. Add an event handler for the selection changed in the view. In the **ProductsListView.xaml** file, add the **SelectionChanged** to the DataGrid control as shown in the following code.
     1. XAML
     2. Controls:DataGrid x:Name="EmployeesList" ItemsSource="{Binding}" **SelectionChanged="SelectedProductChanged**" AutoGenerateColumns="False" IsReadOnly="True">
  8. Add the handler in the view’s code behind. In the **ProductsListView.cs** file add the following methods.
     1. C#
     2. private void SelectedProductChanged(object sender, SelectionChangedEventArgs e)
     3. {
     4. if (e.AddedItems.Count > 0)
     5. {
     6. Product selectedProduct = e.AddedItems[0] as Product;
     7. if (selectedProduct != null)
     8. {
     9. RaiseProductSelected(selectedProduct);
     10. }
     11. }
     12. }
     13. private void RaiseProductSelected(Product product)
     14. {
     15. EventHandler<DataEventArgs<Product>> handler = this.ProductSelected;
     16. if (handler != null)
     17. {
     18. handler(this, new DataEventArgs<Product>(product));
     19. }
     20. }
  9. Handle the ProductSelected event in the Presenter class and call the controller. On the **ProductListPresenter.cs** file add the following lines in bold.
     1. C#
     2. public ProductsListPresenter(IProductsListView view, IProductsService productsService, IProductsController productsController)
     3. {
     4. this.View = view;
     5. this.productsController = productsController;
     6. // retrieve products list
     7. var products = productsService.GetProducts();
     8. // set the view's model
     9. this.View.Model = products;
     10. **// Handle the Product Selection changed event from the view**
     11. **this.View.ProductSelected += View\_ProductSelected;**
     12. }
     13. **private void View\_ProductSelected(object sender, DataEventArgs<Product> e)**
     14. **{**
     15. **this.productsController.OnProductSelected(e.Value);**
     16. **}**
  10. Implement the controller to create and show the details view when a product is selected. Update the **OnProductSelected** method on the **ProductsController** class as shown in the following code.
      1. C#
      2. public void OnProductSelected(Product product)
      3. {
      4. **// Get the Details region (View Injection)**
      5. var region = this.regionManager.Regions["**BottomRegion**"];
      6. **// we will name views to register and retrieve them (using the product id). This avoids having several view instances for the same product.**
      7. **var viewName = product.ProductId.ToString();**
      8. **// Get a view from the region using the view's name.**
      9. **var productDetailsView = region.GetView(viewName);**
      10. **// If the view was never created, create it for the first time**
      11. **if (productDetailsView == null)**
      12. **{**
      13. **// Resolve the presenter and set the product**
      14. **var presenter = this.container.Resolve<ProductDetailsPresenter>();**
      15. **presenter.SetProduct(product);**
      16. **productDetailsView = presenter.View;**
      17. **// Add the view using its name**
      18. **region.Add(productDetailsView, viewName);**
      19. **}**
      20. **// Activate the view**
      21. **region.Activate(productDetailsView);**
      22. }
  11. Update ModuleA to not show the details view, as this is now done by the controller. On the **Initialize** method of the **ModuleA.cs** file comment or delete the code that shows thedetails. The resulting code should look like the following.
      1. C#
      2. public void Initialize()
      3. {
      4. // Use the UIMessagesService to show a message
      5. this.uiMessagesService.ShowMessage("Module A initialized");
      6. // Register the neccessary services
      7. this.RegisterServices();
      8. // Register the neccessary views
      9. this.RegisterViews();
      10. // Add the List view to the main region
      11. ShowListView();
      12. // Add the Details view to the details region
      13. //this.regionManager.RegisterViewWithRegion("**~~BottomRegion~~**",
      14. **// ~~() => this.container.Resolve<ProductDetailsPresenter>().View);~~**
      15. }
  12. Compile and run the application. Now, the details view appears and shows the details of the selected product.
      1. 
      2. Product List and Product Details synchronized

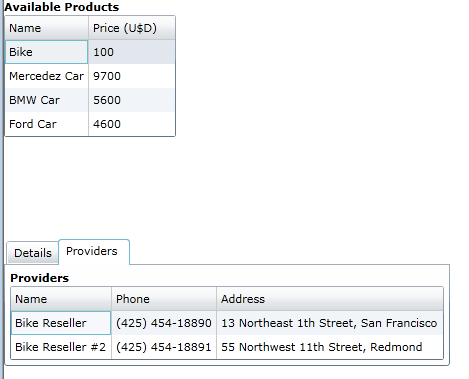
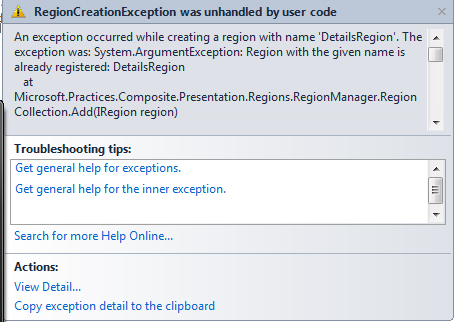
Exercise 3 - Showing a view in a Scoped Region

* 1. In this exercise we will make the details extensible, giving the flexibility to dynamically add new tabs. We will add a new tab to show the providers for the selected product.

Task 1 – Wiring the providers details view

* 1. Open the solution located at **\UIComposition\Exercise 3\Begin\Prism.Workshop.sln.**Notethatthissolution contains an implementation of the **ProvidersDetailsView** view, which can be constructed in a similar way as the **ProductsListView** in the first exercise. It also adds the **Provider** model class, and extends the **ProductsService** service to return providers.
     1. 
  2. Add the following using statements to the **ModuleA.cs** file, located in the **Workshop.ModuleA** project. This is shown in the folowing
     1. C#
     2. using Workshop.ModuleA.Views.ProductProvidersView;
  3. Register the new view in the container. To do this, update the **RegisterViews** method of the **ModuleA** class, as shown in the following code.
     1. C#
     2. private void RegisterViews()
     3. {
     4. // Register views in the container.
     5. this.container.RegisterType<IProductsListView, ProductsListView>();
     6. this.container.RegisterType<IProductDetailsView, ProductDetailsView>();
     7. **this.container.RegisterType<IProductProvidersView, ProductProvidersView>();**
     9. // Register products controller
     10. this.container.RegisterType<IProductsController, ProductsController>();
     11. }
  4. Update the controller to show the providers details. To do this, update the **OnProductSelected** method of the **ProductsController** class, as shown in the following code.
     1. C#
     2. public void OnProductSelected(Product product)
     3. {
     4. // Get the Details region (View Injection)
     5. var region = this.regionManager.Regions["BottomRegion"];
     6. // we will name views to register and retrieve them (using the product id). This avoids having several view instances for the same product.
     7. var viewName = product.ProductId.ToString();
     8. // Get a view from the region using the view's name.
     9. var productDetailsView = region.GetView(viewName);
     10. // If the view was never created, create it for the first time
     11. if (productDetailsView == null)
     12. {
     13. // Resolve the presenter and set the product
     14. var presenter = this.container.Resolve<ProductDetailsPresenter>();
     15. presenter.SetProduct(product);
     16. productDetailsView = presenter.View;
     17. // Add the view using its name
     18. region.Add(productDetailsView, viewName);
     19. **// Add providers View to the details**
     20. **this.AddProviderDetailsView(product);**
     21. }
     22. // Activate the view
     23. region.Activate(productDetailsView);
     24. }
  5. Add the **AddProviderDetailsView** method with the following implementation.
     1. C#
     2. private void AddProviderDetailsView(Product product)
     3. {
     4. // Get the Details regions
     5. var region = this.regionManager.Regions["DetailsRegion"];
     6. // Resolve the presenter and set the product
     7. var presenter = this.container.Resolve<ProductProvidersPresenter>();
     8. presenter.SetProduct(product);
     9. // Add the view using its name
     10. region.Add(presenter.View);
     11. }
  6. Add the regions namespace to the view. To do this, in the **ProductDetailsView.xaml** file, add the following xml namespace.
     1. XAML
     2. <UserControl x:Class="Workshop.ModuleA.Views.ProductDetailsView.ProductDetailsView"
     3. …
     4. **xmlns:Regions="clr-namespace:Microsoft.Practices.Prism.Regions;assembly=Microsoft.Practices.Prism"**
     5. …
     6. d:DesignHeight="300" d:DesignWidth="400">
     7. …
     8. Add the details region to the tabs control, by adding the following attached property to the **ProductDetailsView.xaml** file.
     9. C#
     10. <Controls:TabControl Margin="0,5,0,0" HorizontalAlignment="Stretch" **Regions:RegionManager.RegionName="DetailsRegion"**>
  7. Compile and run the application. You will notice that the tab header for the provider’s details does not look good. That will be fixed it in the following steps. Also, if you select a second product, the application will throw an exception. This is because you are not using scoped regions. This issue will be fixed adding scoped region in the following task.
     1. 
  8. Add the following xaml code to the **ProductDetailsView.xaml**.
     1. XAML
     2. …
     3. <Controls:TabControl Margin="0,5,0,0" HorizontalAlignment="Stretch" Regions:RegionManager.RegionName="DetailsRegion">
     4. **<Style TargetType="Controls:TabItem">**
     5. **<Setter Property="HeaderTemplate">**
     6. **<Setter.Value>**
     7. **<DataTemplate>**
     8. **<TextBlock Text="{Binding Title}" />**
     9. **</DataTemplate>**
     10. **</Setter.Value>**
     11. **</Setter>**
     12. **</Style>**
     13. <Controls:TabItem Header="Details">
     14. …

The preceding code binds to the **Title** property to show header. If you check the **ProviderDetailsPresenter**, you will notice that the **SetProduct** method is creating a view model with the **Title** property set to "*Providers".*

* + 1. C#
    2. public void SetProduct(Product product)
    3. {
    4. // Create a view model wrapping the actual model, to provide additional fields to the view (in this case the Title, that is used by the TabControl to render its header)
    5. var viewModel = new ProvidersDetailsViewModel();
    6. viewModel.Providers = productsService.GetProviders(product.ProductId);
    7. **viewModel.Title = "Providers";**
    8. this.View.Model = viewModel;
    9. }
  1. Compile and run the application. Now, the tab header looks better and the providers view shows the details of the selected product. Also, if you select a second product, the application will still throw an exception because we are not using scoped regions.
     1. 
     2. Providers Details view
     3. 
     4. Exception when selecting a second product, stating that a region with DetailsRegion name already exists.

Task 2 – Using scoped regions

* 1. In this task you will add scoped regions to be able to view the details of the different products.
  2. Update the controller to use scoped regions. To do this, update the **OnProductSelected** method on the **ProductsController** class to create a scoped region when adding the details view. Use the overloaded **Add** method that receives a Boolean parameter, as shown in the following code.
     1. C#
     2. // Add the view using its name. As the view contains other regions, create a ScopedRegionManager
     3. **var bottomRegionManager =** region.Add(productDetailsView, viewName**, true**);
  3. Use the **ScopedRegionManager** to register the providers view. To do this, update the **AddProviderDetailsView** method to receive an **IRegionManager** parameter, and use it to register the view, as shown in the following code.
     1. C#
     2. private void AddProviderDetailsView(Product product, **IRegionManager scopedRegionManager**)
     3. {
     4. // Get the Details regions
     5. var region = **scopedRegionManager**.Regions["DetailsRegion"];
     6. // Resolve the presenter and set the product
     7. var presenter = this.container.Resolve<ProductProvidersPresenter>();
     8. presenter.SetProduct(product);
     9. // Add the view using its name
     10. region.Add(presenter.View);
     11. }
  4. Update the invocation of the **AddProviderDetailsView** to pass the scopedRegionManager created when adding the details view. The complete **OnProductSelected** method of the **ProductsController** should be as following one.
     1. C#
     2. public void OnProductSelected(Product product)
     3. {
     4. // Get the Bottom region (View Injection)
     5. var region = this.regionManager.Regions["BottomRegion"];
     6. // we will name views to register and retrieve them (using the product id). This avoids having several view instances for the same product.
     7. var viewName = product.ProductId.ToString();
     8. // Get a view from the region using the view's name.
     9. var productDetailsView = region.GetView(viewName);
     10. // If the view was never created, create it for the first time
     11. if (productDetailsView == null)
     12. {
     13. // Resolve the presenter and set the product
     14. var presenter = this.container.Resolve<ProductDetailsPresenter>();
     15. presenter.SetProduct(product);
     16. productDetailsView = presenter.View;
     17. // Add the view using its name. As the view contains a region, create a ScopedRegionManager.
     18. var bottomRegionManager = region.Add(productDetailsView, viewName, true);
     19. // Add providers View to the details
     20. this.AddProviderDetailsView(product**, bottomRegionManager**);
     21. }
     22. // Activate the view
     23. region.Activate(productDetailsView);
     24. }
  5. Compile and run the application. Now you should be able to select several products, and see its providers without getting an error.
     1. 