PRISM 4.0 TRAINING KIT

Hands-On Lab

Communication

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277956527)

[Exercise 1 – Using Commands 3](#_Toc277956528)

[Task 1 – Refactoring to the MVVM Pattern 3](#_Toc277956529)

[Task 2 – Executing a Command when a Product is Selected 7](#_Toc277956530)

[Exercise 2 – Communicating across modules using the Event Aggregator 12](#_Toc277956531)

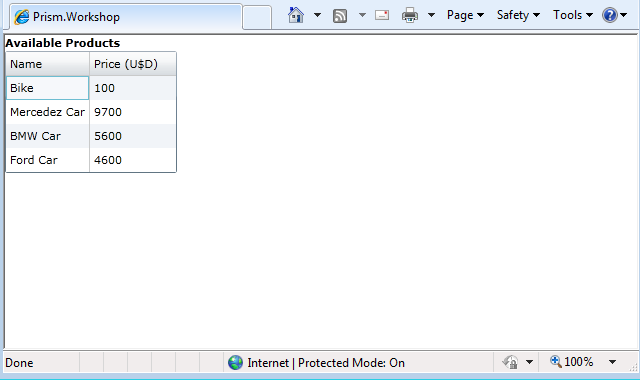
[Task 1 – Using the EventAggregator to Communicate between Modules 12](#_Toc277956532)

Overview

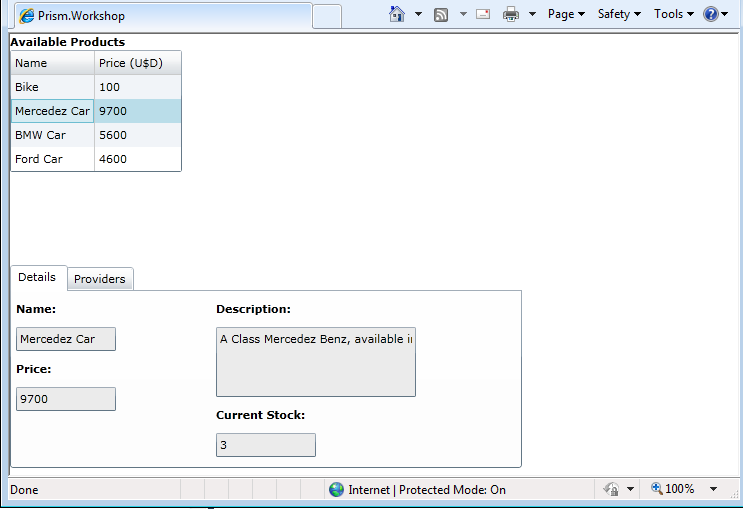
* 1. When building large complex applications, a common approach is to divide the functionality into discrete module assemblies. It is also desirable to minimize the use of static references between these modules. This allows the modules to be independently developed, tested, deployed, and updated, and it forces loosely coupled communication.
  2. 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. Some of the communication approaches provided by Prism are the following:
  + **Commanding**. Use when there is an expectation of immediate action from the user interaction.
  + **Event aggregation**. For communication across view models, presenters, or controllers when there is not a direct action-reaction expectation.

Exercise 1 – Using Commands

Task 1 – Refactoring to the MVVM Pattern

* 1. This task involves creating a **ViewModel** for the **ProductsListView** and using it to replace the existing Presenter.
  2. Open the solution located in the **\Communication\Exercise 1\Begin\Prism.Workshop.sln** folder**.**
  3. Add a new class, named **ProductsListViewModel**,to the **ProductsListView** folder which is located inside the **Views** folder of **Workshop**.**ModuleA** project.
  4. Add the following using statements to the created class.
     1. C#
     2. using Workshop.ModuleA.Model;
     3. using System.Collections.ObjectModel;
     4. using Workshop.ModuleA.Services;
  5. Add an "**ObservableCollection** of **Product"** property, named **Products**, to the **ProductsListViewModel**, as shown in the following code. The property will provide the available products to the View.
     1. C#
     2. public ObservableCollection<Product> Products { get; set; }
  6. Create a new constructor for the **ProductsListViewModel**. The constructor should populate the Products property from an instance of the **IProductsService**,and store an instance of an **IProductsController** to enable future communication with the rest of the module. The following code shows how to achieve this.
     1. C#
     2. private readonly IProductsController productsController;
     4. public ProductsListViewModel(IProductsService productsService, IProductsController productsController)
     5. {
     6. this.Products = productsService.GetProducts();
     7. this.productsController = productsController;
     8. }
  7. Add a new interface, named **IProductsListViewModel**, in the same folder as the **ProductsListViewModel**. The following code shows the interface's definition.
     1. C#
     2. namespace Workshop.ModuleA.Views.ProductsListView
     3. {
     4. public interface IProductsListViewModel
     5. {
     6. }
     7. }
  8. Implement that interface in the **ProductsListViewModel,** as shown in the following code.
     1. C#
     2. public class ProductsListViewModel : IProductsListViewModel
     3. {
     4. …
     5. }
  9. Open the class **ModuleA.**
  10. Update the **RegisterViews** method to register the mapping between the interface and **ViewModel**, as shown in the following code. The bold code shows the necessary update.
      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 viewmodel mappings**
      10. **this.container.RegisterType<IProductsListViewModel, ProductsListViewModel>();**
      11. // Register products controller
      12. this.container.RegisterType<IProductsController, ProductsController>();
      13. }
  11. Open **ProductsListView.xaml.cs**.
  12. Update the class implementation as shown in the following code. As you can notice, one of the benefits of moving to the MVVM pattern is having less code behind.
      1. C#
      2. namespace Workshop.ModuleA.Views.ProductsListView
      3. {
      4. public partial class ProductsListView : UserControl, IProductsListView
      5. {
      6. public ProductsListView(IProductsListViewModel viewModel)
      7. {
      8. this.DataContext = viewModel;
      9. InitializeComponent();
      10. }
      11. }
      12. }
  13. Update the XAML **DataGrid** definition to reflect the latest changes.
      + Remove the handler for the selection changed.
      + Modify the ItemsSource
      1. To do this, open **ProductsListView.xaml** and update the **DataGrid’s** definition with the one shown in the following code.
      2. XAML
      3. <Controls:DataGrid x:Name="EmployeesList" ItemsSource="{Binding Products}" ~~SelectionChanged="SelectedProductChanged"~~ AutoGenerateColumns="False" IsReadOnly="True">
  14. Remove the unnecessary methods from the **IProductsListView** interface. As shown in the following code, the interface should not define any methods.
      1. C#
      2. namespace Workshop.ModuleA.Views.ProductsListView
      3. {
      4. public interface IProductsListView
      5. {
      6. }
      7. }
  15. Open **ModuleA.cs.**
  16. Update the **ShowListView** method to use a View-First approach. This will also remove all usages of the **ProductsListPresenter**. The following code shows the outcome after the update.
      1. C#
      2. private void ShowListView()
      3. {
      4. // Get an instance of the ProductsListView using the container
      5. IProductsListView view = this.container.Resolve<IProductsListView>();
      6. // Get the main region
      7. IRegion mainRegion = this.regionManager.Regions["MainRegion"];
      8. // Add the Product List view to the main region (View Injection)
      9. mainRegion.Add(view);
      10. }
  17. Delete the **ProductsListPresenter** class.
  18. Rebuild and run the solution. You should be able to see the products, but nothing should happen when you select one of them.
      1. 

Task 2 – Executing a Command when a Product is Selected

* 1. In this task, you will create a command that will be invoked when a product is selected in the Available Products list.
  2. Add references to the following assemblies in the **Infrastructure** project:
     + System.Windows.Controls.dll
     + System.Windows.Controls.Data.dll
     + Microsoft.Practices.Prism.dll
  3. Add a new folder, named **Behaviors**, in the **Infrastructure** project.
  4. Add a new class, named **DataGridSelectedItemCommandBehavior**, inside the **Behaviors** folder. This class will contain the logic for the attached behavior that binds the **DataGrid** event with the **Command** in the **ViewModel**.
  5. Replace the automatically generated code for the class with the following.
     1. C#
     2. using System.Windows.Controls;
     3. using Microsoft.Practices.Prism.Commands;
     4. using System.Windows.Input;
     5. using System.Windows;
     6. namespace Workshop.Infrastructure.Behaviors
     7. {
     8. public static class DataGridSelectedItem
     9. {
     10. private static readonly DependencyProperty DataGridSelectedItemCommandBehaviorProperty
     11. = DependencyProperty.RegisterAttached(
     12. "DataGridSelectedItemCommandBehavior",
     13. typeof(DataGridSelectedItemCommandBehavior),
     14. typeof(DataGridSelectedItem),
     15. null);
     16. public static readonly DependencyProperty CommandProperty
     17. = DependencyProperty.RegisterAttached(
     18. "Command",
     19. typeof(ICommand),
     20. typeof(DataGridSelectedItem),
     21. new PropertyMetadata(OnSetCommandCallback));
     22. public static readonly DependencyProperty CommandParameterProperty
     23. = DependencyProperty.RegisterAttached(
     24. "CommandParameter",
     25. typeof(object),
     26. typeof(DataGridSelectedItem),
     27. new PropertyMetadata(OnSetCommandParameterCallback));
     29. public static ICommand GetCommand(DataGrid control)
     30. {
     31. return control.GetValue(CommandProperty) as ICommand;
     32. }
     33. public static void SetCommand(DataGrid control, ICommand command)
     34. {
     35. control.SetValue(CommandProperty, command);
     36. }
     37. public static void SetCommandParameter(DataGrid control, object parameter)
     38. {
     39. control.SetValue(CommandParameterProperty, parameter);
     40. }
     41. public static object GetCommandParameter(DataGrid control)
     42. {
     43. return control.GetValue(CommandParameterProperty);
     44. }
     45. private static void OnSetCommandCallback
     46. (DependencyObject dependencyObject, DependencyPropertyChangedEventArgs e)
     47. {
     48. DataGrid control = dependencyObject as DataGrid;
     49. if (control != null)
     50. {
     51. DataGridSelectedItemCommandBehavior behavior = GetOrCreateBehavior(control);
     52. behavior.Command = e.NewValue as ICommand;
     53. }
     54. }
     55. private static void OnSetCommandParameterCallback
     56. (DependencyObject dependencyObject, DependencyPropertyChangedEventArgs e)
     57. {
     58. DataGrid control = dependencyObject as DataGrid;
     59. if (control != null)
     60. {
     61. DataGridSelectedItemCommandBehavior behavior = GetOrCreateBehavior(control);
     62. behavior.CommandParameter = e.NewValue;
     63. }
     64. }
     65. private static DataGridSelectedItemCommandBehavior GetOrCreateBehavior(DataGrid control)
     66. {
     67. DataGridSelectedItemCommandBehavior behavior =
     68. control.GetValue(DataGridSelectedItemCommandBehaviorProperty) as DataGridSelectedItemCommandBehavior;
     69. if (behavior == null)
     70. {
     71. behavior = new DataGridSelectedItemCommandBehavior(control);
     72. control.SetValue(DataGridSelectedItemCommandBehaviorProperty, behavior);
     73. }
     74. return behavior;
     75. }
     76. }
     77. public class DataGridSelectedItemCommandBehavior : CommandBehaviorBase<DataGrid>
     78. {
     79. public DataGridSelectedItemCommandBehavior(DataGrid control)
     80. : base(control)
     81. {
     82. control.SelectionChanged += OnSelectionChanged;
     83. }
     84. private void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
     85. {
     86. if (e.AddedItems.Count > 0)
     87. {
     88. this.CommandParameter = e.AddedItems[0];
     89. }
     91. ExecuteCommand();
     92. }
     93. }
     94. }
  6. Open the **ProductsListView**.**xaml** file located in the **Workshop**.**ModuleA** project.
  7. Add a namespace definition for the previously created command's namespace, as shown in the following code.
     1. XAML
     2. xmlns:behaviors="clr-namespace:Workshop.Infrastructure.Behaviors;assembly=Workshop.Infrastructure"
  8. Update the DataGrid’s XAML definition to execute the command each time the selection changes. The following code shows the updated XAML.
     1. XAML
     2. <Controls:DataGrid **behaviors:DataGridSelectedItem.Command="{Binding SelectedItemCommand}"** x:Name="EmployeesList" ItemsSource="{Binding Products}" AutoGenerateColumns="False" IsReadOnly="True">
  9. Open the **ProductsListViewModel**.
  10. Add the following using statement to the file.
      1. C#
      2. using Microsoft.Practices.Prism.Commands;
  11. Create a new method that will be called when the command is executed. The method should receive the selected Product as a parameter. Use the following code.
      1. C#
      2. private void ExecuteSelectedItem(Product product)
      3. {
      4. this.productsController.OnProductSelected(product);
      5. }
  12. Add a "**DelegateCommand** of **Product"** property to the ViewModel, and instantiate it in the constructor, as shown in the following code.
      1. C#
      2. public ProductsListViewModel(IProductsService productsService, IProductsController productsController)
      3. {
      4. this.Products = productsService.GetProducts();
      5. this.productsController = productsController;
      6. **this.SelectedItemCommand = new DelegateCommand<Product>(this.ExecuteSelectedItem);**
      7. }
      9. **public DelegateCommand<Product> SelectedItemCommand { get; set; }**
  13. Rebuild and run the solution. When a customer is selected, the details view should be displayed.
      1. 
  14. **Note:** Another possible approach to achieve a similar functionality is binding a property in the **ViewModel** to the **DataGrid’s** SelectedItem and call the controller in the property setter.

Exercise 2 – Communicating across modules using the Event Aggregator

Task 1 – Using the EventAggregator to Communicate between Modules

* 1. In this exercise you will use the Event Aggregator mechanism to communicate two modules.
  2. Open the solution located at **\Communication\Exercise 2\Begin\Prism.Workshop.sln.**
  3. Compile and run the solution. You will notice that both **ModuleA** and **ModuleB** are initialized as two messages boxes should be displayed.
     1. **Note:** Notice that after selecting a **Product** in the Grid, there is only a single **TabItem** in the **TabControl** control. This is because **ModuleB** is not receiving any notification from **ModuleA** that the **Product** was selected.
  4. Add a new folder to the **Infrastructure** project, named **Events**.
  5. Add a new class, named **ProductSelectedEvent**, to the created folder in the previous step.
  6. Add the following using statements.
     1. C#
     2. using Workshop.Infrastructure.Model;
     3. using Microsoft.Practices.Prism.Events;
  7. Update the class' definition to inherit from the "**CompositePresentationEvent** of **Product"** class, as shown in the following code. The generic argument defines the Payload type for the event.
     1. C#
     2. public class ProductSelectedEvent : CompositePresentationEvent<Product>
     3. {
     4. }
  8. Open the **ProductsController** class, located inside the **Views** folder of the **Workshop**.**ModuleA** project.
  9. Add the following using statements.
     1. C#
     2. using Workshop.Infrastructure.Events;
     3. using Microsoft.Practices.Prism.Events;
  10. Update the class constructor to receive an **IEventAggregator** instance, and assign it to a local field, as shown in the following code.
      1. C#
      2. private readonly IRegionManager regionManager;
      3. private readonly IUnityContainer container;
      4. **private readonly IEventAggregator eventAggregator;**
      5. public ProductsController(IUnityContainer container, IRegionManager regionManger, IEventAggregator eventAggregator)
      6. {
      7. this.regionManager = regionManger;
      8. this.container = container;
      9. **this.eventAggregator = eventAggregator;**
      10. }
  11. Modify the **OnProductSelected** method to publish the event, as shown in the following code.
      1. C#
      2. public void OnProductSelected(Product product)
      3. {
      5. // Get the Bottom region (View Injection)
      6. var region = this.regionManager.Regions["BottomRegion"];
      7. // we will name views to register and retrieve them (using the product id). This avoids having several view instances for the same product.
      8. var viewName = product.ProductId.ToString();
      9. // Get a view from the region using the view's name.
      10. var productDetailsView = region.GetView(viewName);
      11. // If the view was never created, create it for the first time
      12. if (productDetailsView == null)
      13. {
      14. // Resolve the presenter and set the product
      15. var presenter = this.container.Resolve<ProductDetailsPresenter>();
      16. presenter.SetProduct(product);
      17. productDetailsView = presenter.View;
      18. // Add the view using its name. As the view contains a region, create a ScopedRegionManager.
      19. var bottomRegionManager = region.Add(productDetailsView, viewName, true);
      20. string regionManagerName = string.Format("Product{0}BottomRegionManager", product.ProductId);
      21. container.RegisterInstance<IRegionManager>(regionManagerName, bottomRegionManager);
      22. }
      23. **//Get the event through event aggregator**
      24. **var productSelected = this.eventAggregator.GetEvent<ProductSelectedEvent>();**
      25. **//Publish the event**
      26. **productSelected.Publish(product);**
      27. // Activate the view
      28. region.Activate(productDetailsView);
      29. }
  12. Open the **ProvidersController** class, located inside the **Views** folder of the **Workshop.ModuleB** project.
  13. Add the following using statements.
      1. C#
      2. using Microsoft.Practices.Prism.Events;
      3. using Workshop.Infrastructure.Events;
      4. using Workshop.ModuleB.Views.ProductProvidersView;
      5. using Microsoft.Practices.Prism.Regions;
      6. using Microsoft.Practices.Unity;
      7. using Workshop.Infrastructure.Model;
  14. Create a new method, named **OnProductSelected**, which will be in charge of adding the **ProvidersView** instance into the **BottomRegion** (using ViewInjection). The following code shows how to achieve this.
      1. C#
      2. public void OnProductSelected(Product product)
      3. {
      4. string regionManagerName = string.Format("Product{0}BottomRegionManager", product.ProductId);
      5. IRegionManager bottomRegionManager = container.Resolve<IRegionManager>(regionManagerName);
      6. string viewName = "ProvidersView";
      7. if (bottomRegionManager.Regions["DetailsRegion"].GetView(viewName) == null)
      8. {
      9. var presenter = this.container.Resolve<ProductProvidersPresenter>();
      10. presenter.SetProduct(product);
      11. bottomRegionManager.Regions["DetailsRegion"].Add(presenter.View, viewName);
      12. }
      13. }
  15. Add a new constructor for the class that receives an **IEventAggregator**, **IRegionManager** and **IUnityContainer** instance. This is shown in the following code.
      1. C#
      2. public ProvidersController(IEventAggregator eventAggregator, IRegionManager regionManager, IUnityContainer container)
      3. {

}

* 1. Assign the **IRegionManager** and **IUnityContainer** instances to local fields, as shown in the following code.
     1. C#
     2. **private readonly IRegionManager regionManager;**
     3. **private readonly IUnityContainer container;**
     4. public ProvidersController(IEventAggregator eventAggregator, IRegionManager regionManager, IUnityContainer container)
     5. {
     6. **this.regionManager = regionManager;**

**this.container = container;**

* + 1. }
  1. Subscribe to the **ProductSelectedEvent**, using the **IEventAggregator** instance.
     1. C#
     2. public ProvidersController(IEventAggregator eventAggregator, IRegionManager regionManager, IUnityContainer container)
     3. {
     4. this.regionManager = regionManager;
     5. this.container = container;
     7. **ProductSelectedEvent productSelected = eventAggregator.GetEvent<ProductSelectedEvent>();**
     8. **productSelected.Subscribe(this.OnProductSelected, true);**
     9. }
     10. **Note:** The boolean parameter determines if a strong reference should be kept to the subscriber. In this case, as there are no other references to the **ProviderController**, it is useful to make sure it is not Garbage Collected, and the event is never handled.
  2. Open **ModuleB.cs**.
  3. Add the following using statements.
     1. C#
     2. using Workshop.ModuleB.Views.ProductProvidersView;
     3. using Workshop.ModuleB.Views;
  4. Update the **Initialize** method to register the necessary mappings and resolve the **ProvidersController**. This is shown in the following code.
     1. C#
     2. public void Initialize()
     3. {
     4. this.messagesService.ShowMessage("Module B Initialized");
     5. **this.unityContainer.RegisterType<IProductProvidersView, ProductProvidersView>();**
     6. **this.unityContainer.Resolve<ProvidersController>();**
     7. }
  5. Rebuild and run the solution. When selecting a **Product** in the **DataGrid**, two views should be displayed in the **TabRegion**, as shown in the following picture.
     1. 