Logger

Na początku stworzymy własny logger. Do tego calu trzeba utworzyć klasę naszego loggera, która będzie implementować interfejs ILoggerFacade. Oto kod tej klasy:

using System.Diagnostics;

using Microsoft.Practices.Prism.Logging;

 public class SimpleLogger: ILoggerFacade

 {

 public void Log(string message, Category category, Priority priority)

 {

 switch (category)

 {

 case Category.Debug:

 Debug.Write(Debug: + message);

 break;

 case Category.Warn:

 Debug.Write(Warn: + message);

 break;

 case Category.Exception:

 Debug.Write(Exception: + message);

 break;

 case Category.Info:

 Debug.Write(Info: + message);

 break;

 }

 }

 }

Następnie w klasie Bootstrapper musimy zarejestrować nasz nowy logger:

protected override Microsoft.Practices.Prism.Logging.ILoggerFacade CreateLogger()

{

 return new SimpleLogger();

}

W ten sposób możemy dołączyć do aplikacji dowolny logger, np. Log4Net.

Do naszego loggera można dostać się bezpośrednio z IoC container, albo użyć ServiceLocator.

Przykład użycia ServiceLocator:

var logger = (SimpleLogger)ServiceLocator.Current.GetInstance(typeof(ILoggerFacade));

logger.Log(Inicjalizacja aplikacji zakończona, Category.Info, Priority.None);

Rozszerzanie nawigacji

Tworzymy klasę ViewModelContentLoader, która nadpiszę standardowe zachowanie nawigacji, do tego celu musi dziedziczyć po klasie RegionNavigationContentLoader. Nadpiszemy metodę GetContractFromNavigationContext, w której odwrócimy nawigację. Po przejściu do modułu A wyświetli się moduł B i na odwrót. Oto kod tej klasy:

 public class ViewModelContentLoader : RegionNavigationContentLoader

 {

 private IServiceLocator serviceLocator;

 public ViewModelContentLoader(IServiceLocator serviceLocator)

 :base(serviceLocator)

 {

 }

 protected override string GetContractFromNavigationContext(NavigationContext navigationContext)

 {

 string contract = base.GetContractFromNavigationContext(navigationContext);

 if (contract.Equals(ModuleBView, StringComparison.OrdinalIgnoreCase))

 {

 return typeof(ModuleA.ModuleAView).Name;

 }

 if (contract.Equals(ModuleAView, StringComparison.OrdinalIgnoreCase))

 {

 return typeof(ModuleB.ModuleBView).Name;

 }

 return contract;

 }

 }

Teraz wystarczy zarejestrować nasze rozszeżenie w klasie bootstraper:

 protected override void InitializeShell()

 {

 this.Container.RegisterInstanceIRegionNavigationContentLoader(new ViewModelContentLoader(this.Container.ResolveIServiceLocator()));

 //...

 }

Adapter

W Prism mamy dostępnych kilka adapterów regionów od ręki - pozostałe musimy sami stworzyć.

Kiedy może zajść konieczność stworzenia własnego adaptera regionów? Jeżeli chcemy użyć jakiejś kontrolki jako miejsca w które Prism będzie ładował widoki, a obecnie nie ma takiej obsługi w prosty sposób możemy taką obsługę dodać. Oto przykładowy kod:

 Window x:Class=Shell.ShellWindow

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

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

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

 Title=Shell

 StackPanel Orientation=Horizontal

 prism:RegionManager.RegionName=MenuRegion

 StackPanel Orientation=Vertical

 prism:RegionManager.RegionName=MainRegion

 /StackPanel

 /StackPanel

/Window

Kod ten umieszczamy w ShellWindow.xaml. Uruchomienie aplikacji spowoduje następujący błąd. W wolnym tłumaczeniu wyjątek oznacza iż dana kontrolka nie obsługuje regionów. Tak więc nie pozostaje nam nic innego jak dodać do kontrolki obsługę regionów.

Dodanie obsługi danego regionu składa się z 4 kroków:

1.Tworzymy klasę dziedziczącą po RegionAdapterBaseT

 public class StackPanelRegionAdapter : RegionAdapterBaseStackPanel

 public StackPanelRegionAdapter(IRegionBehaviorFactory regionBehaviorFactory)

 : base(regionBehaviorFactory)

 {

 }

 {

 protected override void Adapt(IRegion region, StackPanel regionTarget)

 {

 throw new SystemNotImplementedException();

 }

 protected override IRegion CreateRegion()

 {

 throw new SystemNotImplementedException();

 }

 }

2.Implementujemy metodę CreateRegion która zwraca jeden z trzech wyników determinujących zachowania regionu:

* SingleActiveRegion - jeden aktywny region w danym momencie (ContentControl)
* AllActiveRegion - wszystkie regiony aktywne (ItemsControls)
* Region - wiele aktywnych regionów (SelectorControls)

 protected override IRegion CreateRegion()

 {

 return new AllActiveRegion();

 }

3.Implementacja metody Adapt

protected override void Adapt(IRegion region, StackPanel regionTarget){

 region.Views.CollectionChanged += (s, e) =

 {

 if (e.Action == NotifyCollectionChangedAction.Add)

 {

 foreach (FrameworkElement item in e.NewItems){

 regionTarget.Children.Add(item);

 }

 }

 else if (e.Action == NotifyCollectionChangedAction.Remove)

 {

 foreach (FrameworkElement item in e.NewItems)

 {

 if (regionTarget.Children.Contains(item))

 {

 regionTarget.Children.Remove(item);

 }

 }

 }

 };

 }

Tak więc w metodzie tej nasłuchujemy zdarzenia zmiany kolekcji. Na podstawie typu zdarzenia odpowiednio reagujemy w przypadku dodawania nowego elementu - dodajemy go do naszego StackPanel-u za pomocą właściwości Children w przypadku usuwania - usuwamy odpowiedni element ze StackPanel-u.

4.Rejestracja stworzonego adaptera w Bootstrapperze

Za pomocą metody ConfigureRegionAdapterMappings dodajemy informację o naszym Adapterze StackPanel.

 protected override RegionAdapterMappings ConfigureRegionAdapterMappings()

 {

 RegionAdapterMappings mappings = base.ConfigureRegionAdapterMappings();

 mappings.RegisterMapping(typeof(StackPanel), Container.ResolveStackPanelRegionAdapter());

 return mappings;

 }