Logger

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

```
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 muimy 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);
```

Rozszeżanie Nawigacji

W Prim mamy możliwość także rozszerzyć nawigację. Zobaczmy to na przykładzie w którym po przejściu do modułu A otwiera się moduł B, a po przejściu do B otwiera się moduł A.
Tworzymy klasę która dziedziczy po klasie RegionNavigationContentLoader. Nadpisujemy metodę GetContractFromNavigationContext.

```
internal 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;
        }
    }
```

Naszą metodę możemy zarejestrować w Bootstraperze w taki sposób:

```
protected override void InitializeShell()
        {
            this.Container.RegisterInstance<IRegionNavigationContentLoader>(new
ViewModelContentLoader(this.Container.Resolve<IServiceLocator>()));
            //...
        }
```

Adapter

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

```
<Window x:Class="WpfApplication1.Shell"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:prism="clr-
namespace:Microsoft.Practices.Prism.Regions;assembly=Microsoft.Practices.Prism"
        Title="Shell" Height="300" Width="300">
    <Grid>
        <StackPanel Orientation="Vertical">
            <StackPanel Orientation="Horizontal"
prism:RegionManager.RegionName="NavigationRegion" />
            <ContentControl prism:RegionManager.RegionName="ContentRegion" />
        </StackPanel>
    </Grid>
</Window>
```

W jakich sytuacjach 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. Zobaczmy na przykład:Uruchomienie aplikacji spowoduje 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:

- Tworzymy klasę dziedziczącą po RegionAdapterBase<T>
- 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)
- Implementacja metody Adapt
- Rejestracja stworzonego adaptera w Bootstrapperze

Żeby lepiej zobrazować powyższe kroki przejdziemy je podczas implementowania obsługi regionów w kontrolce StackPanel.

1. Klasa dziedzicząca po klasie RegionAdapterBase<T> :

```
public class StackPanelRegionAdapter : RegionAdapterBase<StackPanel>
 {
    protected override void  Adapt(IRegion region, StackPanel regionTarget)
    {
      throw new SystemNotImplementedException();
    }

    protected override IRegion  CreateRegion()
    {
      throw new SystemNotImplementedException();
    }
 }
```

2. Metoda 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.

3. Metoda CreateRegion:

```
protected override IRegion CreateRegion()
      {
         return new AllActiveRegion();
      }
```

Zwracamy informację że wszystkie elementy dodane do naszego StackPanel-a będą domyślne aktywne.

4. Dodanie konstruktora do tworzonej klasy StackPanelRegionAdapter

Nasza klasa wymaga jeszcze konstruktora - dodajemy go więc.

```
public StackPanelRegionAdapter(IRegionBehaviorFactory regionBehaviorFactory)
         : base(regionBehaviorFactory)
      {
      }
```

Tak więc pierwsza część za nami zostało tylko zarejestrowanie naszego adaptera w Bootstrapperze co przedstawia się następująco:

```
protected override RegionAdapterMappings ConfigureRegionAdapterMappings()
      {
         RegionAdapterMappings mappings = base.ConfigureRegionAdapterMappings();
         mappings.RegisterMapping(typeof(StackPanel),
Container.Resolve<StackPanelRegionAdapter>());

         return mappings;
      }
```

Za pomocą metody ConfigureRegionAdapterMappings dodajemy informację o naszym Adapterze StackPanel.