Rozwój aplikacji modułowych

Paweł Brudnicki

Moduł jest podstawową jednostką funkcjonalności. Stanowi on część, która może być niezależnie pisana, testowana a nawet wdrażana.

W przewodniku dodamy niezależny moduł do aplikacji napisanej przez poprzedniego prelegenta.

Dodanie modułu

- Dodajemy do solucji nowy projekt typu WPF User Control Library o nazwie ModuleC. Następnie usuwamy domyślnie stworzoną kontrolkę UserControl1.xaml.
- Dodajemy dwie referencje do projektu : Prism oraz Prism.UnityExtensions
- Dodajemy do projektu klasę ModuleC.cs. Posłuży ona do rejestrowania widoków, view modeli itd..

```
public class ModuleC : IModule
{
   protected IRegionManager RegionManager { get; private set; }

   public ModuleC(IRegionManager regionManager, IUnityContainer container)
   {
       RegionManager = regionManager;
   }
   public void Initialize()
   {
       throw new NotImplementedException();
   }
}
```

Dodajemy widok projektu

- dodajemy User Control o nazwie ModuleCView
- Edytujemy jej kod

```
<UserControl x:Class="ModuleC.ModuleCView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:prism="clr-
namespace:Microsoft.Practices.Prism.Mvvm;assembly=Microsoft.Practices.Prism.Mvvm.Desktop"
             prism:ViewModelLocator.AutoWireViewModel="True"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="900">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="1*"/>
            <RowDefinition Height="5*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="550*"/>
            <ColumnDefinition Width="30*"/>
            <ColumnDefinition Width="300*"/>
            <ColumnDefinition Width="300*"/>
            <ColumnDefinition Width="300*"/>

        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Horizontal" Grid.Row="0" Grid.Column="0">
            <Button Content="Add to Cart" Command="{Binding AddToCartCommand}" CommandParameter="{Binding
ElementName=ListV, Path=SelectedItem}" Margin="0,0,5,0"/>
            <Button Content="Remove form  Cart" Command="{Binding
RemoveFromCartCommand}" CommandParameter="{Binding ElementName=Cart, Path=SelectedItem}" Width="114"/>
        </StackPanel>
        <StackPanel Grid.Row="1" Grid.Column="3">
            <Label>Suma</Label>
            <TextBox Text="{Binding Suma}" />
            <Button Content="Sum" Command="{Binding SumCommand}"/>
        </StackPanel>

        <ListView ItemsSource="{Binding Products}" Background="Azure" Grid.Row="1" x:Name="ListV">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Name}"/>
                        <TextBlock Text="{Binding Price}"/>
                    </StackPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
```

- Następnie w klasie ModuleC rejestrujemy widok

```
public void Initialize()
{
   Container.RegisterType<Object, ModuleCView>(typeof(ModuleCView).Name);
   RegionManager.RegisterViewWithRegion("MenuRegionC", typeof(ButtonViewC));
}
```

- W klasie Bootsraper dodajemy moduł do katalogu

```
protected override void ConfigureModuleCatalog()
{
   base.ConfigureModuleCatalog();
   ModuleCatalog moduleCatalog = (ModuleCatalog)this.ModuleCatalog;
   moduleCatalog.AddModule(typeof(ModuleA.ModuleA));
   moduleCatalog.AddModule(typeof(ModuleB.ModuleB));
   moduleCatalog.AddModule(typeof(ModuleC.ModuleC));
}
```

.

Dodanie viewModelu

ViewModel pozwoli nam na integrację modelu z widokiem.

- Tworzymy klasę o nazwie ModuleCViewViewModel
- Definiujemy własności
  Stworzenie własności wraz użyciem InotifyPropertyChanged umożliwi nam dynamiczną zmianę zawartości pól, które zostały stworzone w widoku.
  Własności bindowane są z textboxami.

```
public decimal Suma
{
  get
  {
    return suma;
  }
  set
  {
    SetProperty<decimal>(ref suma, value);
  }
}
public ObservableCollection<Product> ProductsInChart
{
  get
  {
    return cart;
  }
  protected set
  {
    SetProperty<ObservableCollection<Product>>(ref cart, value);
  }
}

public ObservableCollection<Product> Products
{
  get
  {
    return products;
  }
  protected set
  {
    SetProperty<ObservableCollection<Product>>(ref products, value);
  }
}
```

- Implementujemy konstruktor, w którym deklarujemy komendy oraz importujemy dane z repozytorium.

```
 protected IProductRepository productsRepository;
public ModuleCViewViewModel(IProductRepository productsRepository)
{
            this.productsRepository = productsRepository;
            AddToCartCommand = new DelegateCommand<Product>(AddToCart);
            RemoveFromCartCommand = new DelegateCommand<Product>(RemoveFromCart);
            SumCommand = new DelegateCommand(Sum);
            this.Products = new ObservableCollection<Product>(this.productsRepository.GetAll());
            cart = new ObservableCollection<Product>();
}
```

- na sam koniec dodajemy komendy , które są podpięte do przycisków

```
 private void AddToCart(Product product)
{
    ProductsInChart.Add(product);
}

private void RemoveFromCart(Product product)
{
   ProductsInChart.Remove(product);
}

private void Sum()
{
   Suma = ProductsInChart.Sum(x => x.Price);
}
```

Aplikacje modułowe dają programiście bardzo duże możliwości. Sprzyjają one pracy zespołowej, ponieważ programiści mogą tworzyć moduły niezależnie od siebie.

Ewentualne problemy z jednym modułem nie wpływają bowiem , na działanie innych.