## Building a composite WPF UI using the View Model first approach

What do we do when a logic of a UI screen becomes overly complex? Or what do we do when a there is a certain part of the screen that we want to reuse on other screens? In both cases there is the same answer: UI composition. We break our screen into smaller parts. Each part is focused on its primary goal and can be reused in other screens in the application. This article is the first one in the two articles Blog series. Here we will demonstrate a simple composite WPF MVVM application. In the second article we will enhance our app and add Unit Testing capabilities.

In WPF the primary mean for UI composition is UserControl. It encapsulates UI and connected logic and can be placed on other screens like the following:

<UserControl x:Class="UICompositionSample.Views.ChildUserControl"

... >

<TextBlock>Child control</TextBlock>

</UserControl>

*Code Listing 1: ChildUserControl.xaml*

<Window x:Class="UICompositionSample.Views.ParentView"

xmlns:local="clr-namespace:UICompositionSample.Views"

... >

<Grid>

<local:ChildUserControl/>

</Grid>

</Window>

*Code Listing 2: ParentWindow.xaml*

Static placing of UserControls into parent Windows can be enough in simple cases but for more complex scenarios we need something more powerful. Let’s design a small hypothetical application. It is a viewer that allows a user to navigate through a list of items and shows a preview for the selected item. Depending on the item content it can be a Text or Image or other kind of a preview.

Text Item 1

Image Item 1

Text Item 2

Image1 Preview

Preview is shown for the selected item inside the Main Window

Main Window

*Image1: The structure of the Viewer app*

We will design our application in the following way:

* For each type of Image Preview we will develop its own UserControl
* In the main Window we will have a preview area. Depending on the selected item type we will dynamically create the correct preview UserControl and initialize it with the selected item data
* We will develop our application with MVVM so that in the future it can be easily unit tested

*A small detour into MVVM. Model-View-ViewModel approach was introduced by Microsoft in WPF and features a full separation of UI (View), UI Logic (ViewModel) and Data (Model). A UserControl built with MVVM will have a View and ViewModel. To create a new instance of the UserControl in the run time we will need to create its View, ViewModel and connect between them. To simplify these task as well as other routine tasks that arise in WPF we will use a Caliburn.Micro framework. For UI composition MVVM frameworks employ one of the 2 approaches:*

* *View First – in the run time your code specifies the View to display in a designated area. The framework automatically creates the View and ViewModel and connects between them*
* *ViewModel First – in the runtime your code creates the ViewModel and binds it into a designated area. The framework automatically creates the View and connects it with the ViewModel*

*One of the advantages of the ViewModel First approach is that we create a child ViewModel in our code and cass pass all needed parameters directly into its constructor.*

Now let’s see some code. In the MainView we will have a grid with 2 columns. The column 0 will display a list of items and the column 1 will contain a placeholder for a preview:

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="2\*"/>

</Grid.ColumnDefinitions>

<ListView Grid.Column="0" ItemsSource="{Binding Items}" SelectedItem="{Binding SelectedItem, Mode=TwoWay}">

<ListView.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding Name}"/>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

<!--this is the way to tell Caliburn.Micro to inject here a view

that corresponds to the PreviewViewModel-->

<ContentControl Grid.Column="1" cal:View.Model="{Binding PreviewViewModel}"/>

</Grid>

*Code Listing 3: XAML markup from the MainView*

The preview placeholder is bound to PreviewViewModel property of the MainViewModel. Upon selection change the correct ViewModel is created and loaded into PreviewViewModel:

//each item derive from IContentItem

ObservableCollection<IContentItem> \_items;

public ObservableCollection<IContentItem> Items => \_items;

IContentItem \_selectedItem;

public IContentItem SelectedItem

{

get => \_selectedItem;

set

{

if (\_selectedItem != value)

{

\_selectedItem = value;

//Update preview on changing of the selected item

UpdatePreview(\_selectedItem);

NotifyOfPropertyChange(nameof(SelectedItem));

}

}

}

//preview ViewModel property for binding into view

object \_previewViewModel;

public object PreviewViewModel

{

get => \_previewViewModel;

set

{

if (\_previewViewModel != value)

{

\_previewViewModel = value;

NotifyOfPropertyChange(nameof(PreviewViewModel));

}

}

}

private void UpdatePreview(IContentItem item)

{

//correct preview ViewModel is created

//according to the type of the selected item

if (item is TextItem)

{

PreviewViewModel = new TextPreviewViewModel(item as TextItem);

}

else if (item is ImageItem)

{

PreviewViewModel = new ImagePreviewViewModel(item as ImageItem);

}

else

{

PreviewViewModel = null;

}

}

*Code Listing 3: Code from MainViewModel*

As you can see the logic of UI composition is very straight forward. Each time when we select an item we create the corresponding ViewModel and place it into PreviewViewModel property. This property is bound to ContentControl in the MainView. This instructs Caliburn.Micro to find the View that matches the created ViewModel and inject it into the MainView.

We can easily add new types of items and new types of previews by adding new ifs into the UpdatePreview function. We also have a complete control on the data that is passed to the child ViewModels. In our application it is only the selected data item but in the real life it can be an arbitrary data and services that allow complex interactions between various parts of UI.

In the second article of the blog series we will enhance our UI and see how the logic of the MainViewModel can be easily tested.

The fill code of the sample app is available at