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

Most of the not trivial UI screens can benefit from UI composition - literally composing a complex screen from simpler sub screens. This breakage into sub screens helps to control the complexity of UI by allowing each sub screen to be focused on its primary task. It also promotes code reuse because each standalone sub screen can be used on different screens.

In WPF the simple component of reuse is UserControl. It has a View (presentation) and ViewModel (UI logic) and can be reused in other views. Please see below the example of a UserControl and its usage in a parent Window.

In many cases it is not enough to statically place a child UserControl in a parent view. We may need to conditionally decide in the run time what type of a child user control should be created and what data should be passed to it. Since a child UserControl consists of a View and a ViewModel we need to create both and connect between them. The existing MVVM frameworks usually simplify this process and promote one of the below approaches:

* View first - the application dynamically decides what child View should be shown and the framework instantiates and brings this View with its corresponding ViewModel.
* ViewModel first - child ViewModel is created by a parent ViewModel and then a framework instantiates and brings the corresponding View.

From this description we can immediately see the benefit of the ViewModel first approach: a parent explicitly creates child ViewModel and can pass the the necessary parameters to a child ViewModel constructor.

For this series of articles, we will develop a hypothetical items viewer screen. This screen will display a list of items with a suitable preview and editor for each item. In the first article will demonstrate how to create this application using a ViewModel fist approach. In the second article we will add unit testing to our screen. Our code will rely on Caliburn.Miro framework but the experience shall be similar with any framework that supports ViewModel first.

UI layer code was traditionally difficult for unit testing due to dependencies on UI controls. In WPF Microsoft has greatly simplified testing of UI logic by introducing MVVM pattern and isolating of View Model (UI logic) from the View (presentation). While unit testing of standalone View Model is relatively straightforward, in the real-world applications the UI is often a composition of many views and view models. Ideally, we would like to have a capability not only to tests the isolated view models but also the correct creation of composite UI. In this post we will focus on view model first approach, meaning that view models are created first and corresponding views are instantiated by a framework. We will use Caliburn.Micro framework with its accompanying SimpleContainer for dependency injection but the outlined approach should work with any view model first framework.

For demo purposes we will use a hypothetic application that allows to go through a list of items and show a preview suitable for the selected item type. It will also allow to open an editor window for the selected item.

Image1 Editing

Main Window

Image1 Preview

Text Item 1

Editor window is opened for the selected item

Preview is shown for the selected item inside the Main Window

Edit… Item 2

Text Item 2

Image Item 1

Let’s view now how Main Window is composed using ViewModel first approach:

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…>

</ListView>

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

</Grid>

Actual data is provided by the . The provider is

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

ObservableCollection<IContentItem> \_items;

public ObservableCollection<IContentItem> Items => \_items;

object \_previewViewModel;

public object PreviewViewModel

{

get => \_previewViewModel;

set

{

if (\_previewViewModel != value)

{

\_previewViewModel = value;

NotifyOfPropertyChange(nameof(PreviewViewModel));

}

}

}

IContentItem \_selectedItem;

public IContentItem SelectedItem

{

get => \_selectedItem;

set

{

if (\_selectedItem != value)

{

\_selectedItem = value;

UpdatePreview(\_selectedItem);

NotifyOfPropertyChange(nameof(SelectedItem));

}

}

}

…

private void UpdatePreview(IContentItem 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;

}

}

Now our purpose is to write a unit test that validates the correct creation of PreviewViewModel with the selected data item. We will write our tests with MSTest for mocking purposes we will use Moq which is an excellent mock creation library for .NET. First, let’s see the test setup that creates a