Support
support@visualcomponents.com

Community
community.visualcomponents.net

Adding new panes

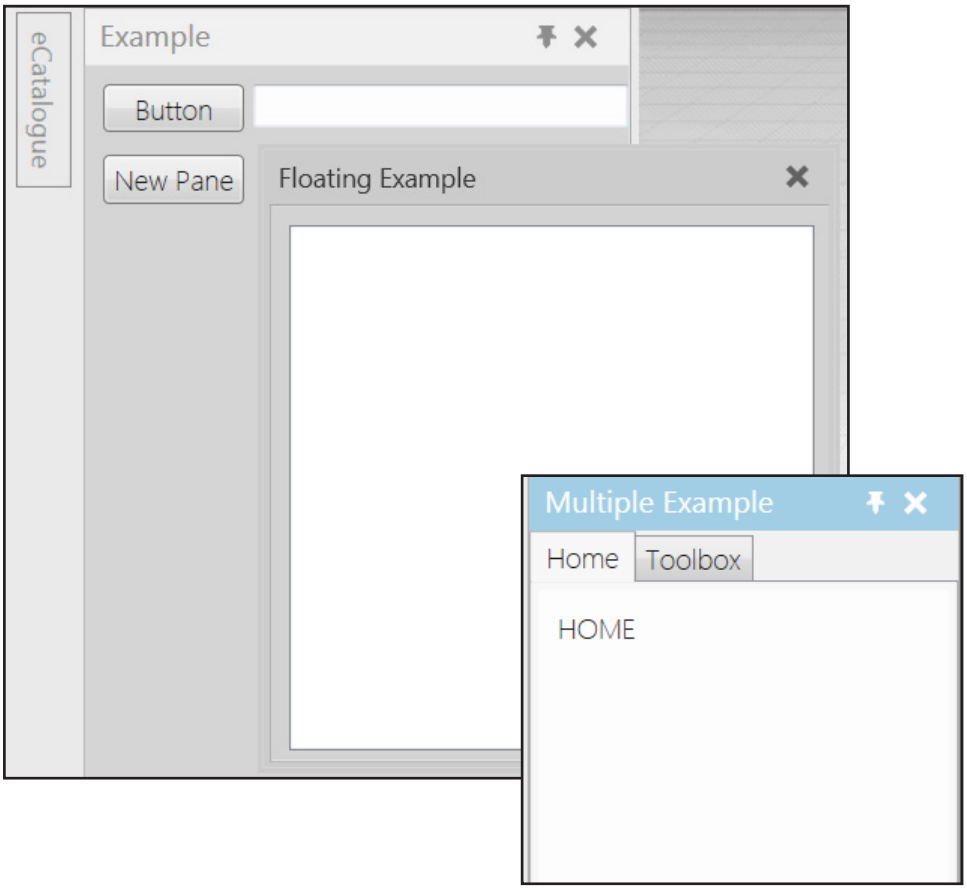
Next Generation | Version: November 20, 2015 | Example: Available upon request

Customizable panes can be added to Essentials as extensions for performing services and executing new functionality.

The setup for creating a pane is minimal and requires some adherence to naming conventions and the MVVM design pattern. Each pane is a class implemented as a ViewModel that is composed of Views containing controls. The class itself can inherit the DockableScreen class and be exported as a type of IDockableScreen to be loaded automatically at runtime.

The topics covered in this tutorial include:

- Minimal setup for adding a pane to the main window of Esssentials.
- Using the IDockAwareWindowManager to add panes on demand.
- Using the Conductor class to add panes with multiple views.



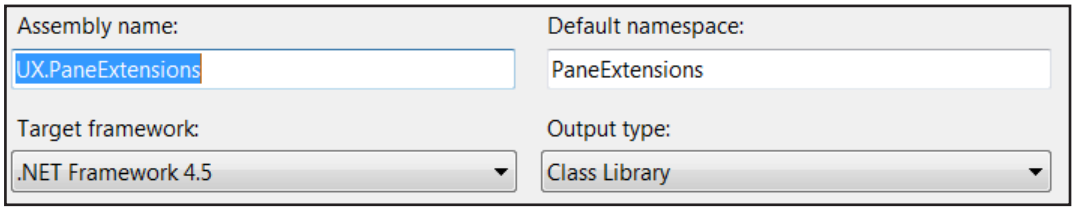
Creating a project in Visual Studio

Visual Studio 2013 and Visual C# are used to create a Class Library of panes that can be used in Essentials.

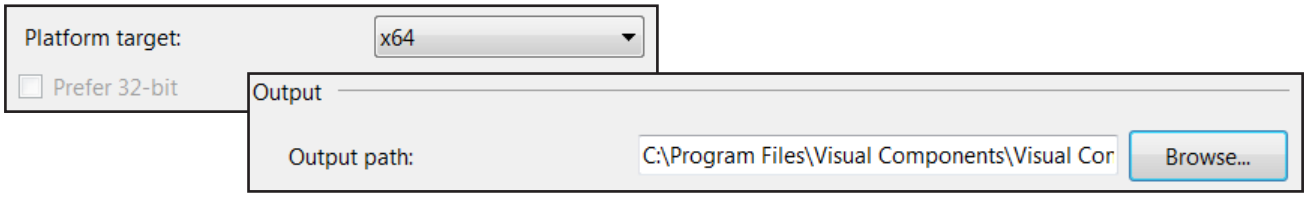
1. Run **Visual Studio** as an **administrator** in order to write to program files on your device.

2. Create a **Class Library** project, and then access the **properties** of your project.

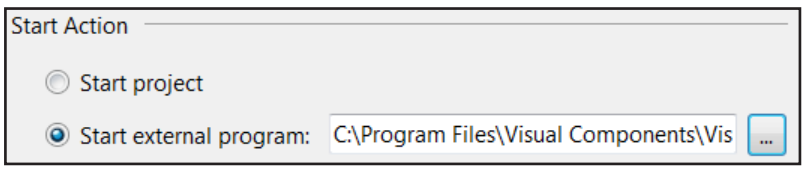
3. In the **Application** tab, set **Assembly name** to have a prefix of **UX.** in order to id your assembly as an extension of the VisualComponents.UX namespace.



4. In the **Build** tab, set **Platform target** to either **x86** or **x64** and then set **Output path** to be the path to your Essentials program files in order for your assembly to be discovered by the MEF and loaded at runtime.



5. In the **Debug** tab, set **Start external program** to execute your development version of Essentials.



A pane may require screen logic, docking functionality, and composition as an extension of Essentials.

6. In your project, add references to:

   - **Caliburn.Micro** that is available in your Essentials program files and online.
   - **System.ComponentModel.Composition** that is available in the .NET Framework.
   - **System.Xaml** that is available in the .NET Framework.
   - **UX.Shared** that is available in your Essentials program files.

Creating a ViewModel

A ViewModel is the abstraction of a pane and contains source code for controls in a View.

1. In your project, add a **Class** and name that class to have a suffix of **ViewModel** in order to follow a naming convention.

2. Add using statements for:

   - **Caliburn.Micro** to handle MVVM related issues.

   - **System.ComponentModel.Composition** to handle MEF related issues.

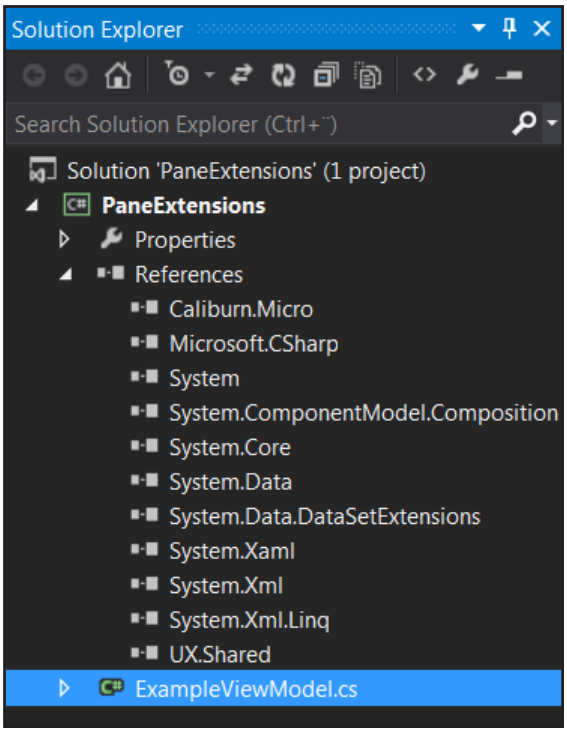
   - **VisualComponents.UX.Shared** to handle development issues related to Essentials.

3. Export your class as a type of **IDockableScreen** in order to fulfill a contract with the main window of Essentials.

4. Define your class as a subclass of **DockableScreen** to inherit screen logic and docking functionality.

5. Create a **default constructor** that provides a **DisplayName** for each instance of the class.

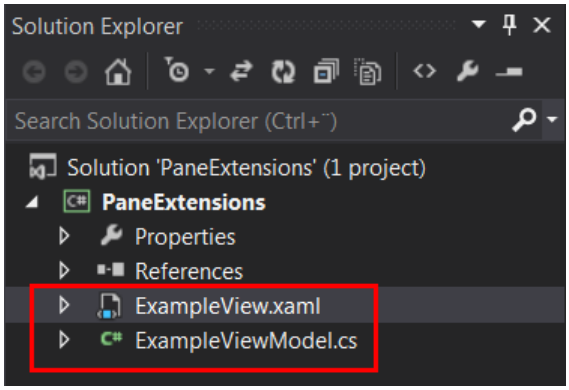
```
namespace PaneExtensions
{
    using Caliburn.Micro;
    using System.ComponentModel.Composition;
    using VisualComponents.UX.Shared;

    [Export(typeof(IDockableScreen))]
    class ExampleViewModel: DockableScreen
    {
        public ExampleViewModel()
        {
            this.DisplayName = "Example";
        }
    }
}
```



Creating a View

A View is the presentation of a pane and contains controls that are targets for source code in a ViewModel.

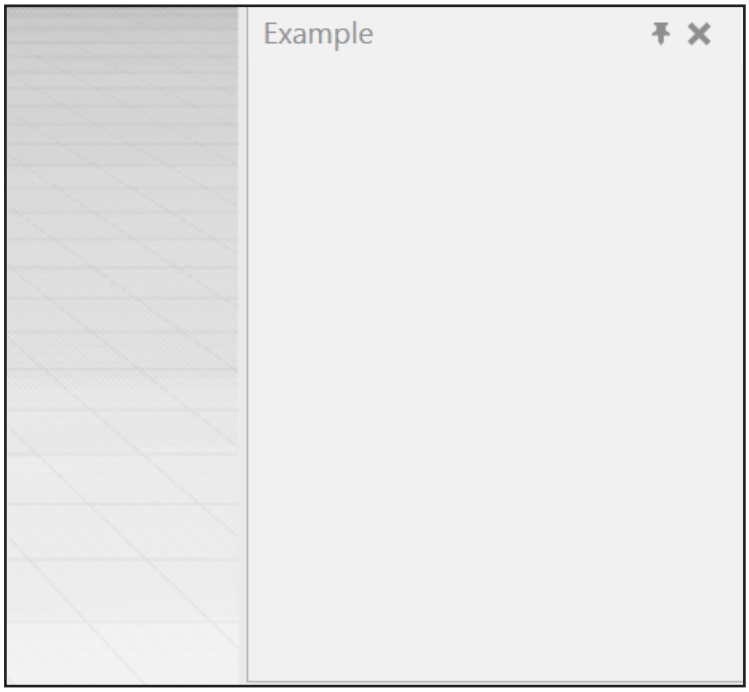
1. In your project, add a **WPF User Control** and use the name of your class **sans Model** in order to follow a naming convention.



At this stage, a pane has the basic requirements to be added to the main window of Essentials. By default, a docked pane is positioned to the right of the 3D world.

2. Start debugging the application.

3. In **Essentials**, verify a new pane has been docked to the right of the 3D world.



4. Stop debugging the application.

Binding targets and sources

You may want to customize a View by binding controls and dependency properties to members in a ViewModel.

1. In your ViewModel, create a **public property** and **public method** that allows you to edit the displayed message in a **VcMessageBox**.

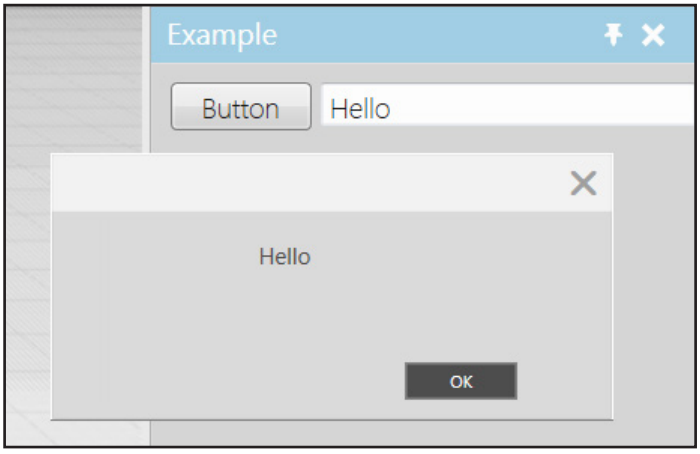
```
public void ShowMyMessage()
{
    VcMessageBox.Show(MyMessage);
}

private string _myMessage;
public string MyMessage
{
    get { return _myMessage; }
    set
    {
        _myMessage = value;
        NotifyOfPropertyChange(() => MyMessage);
    }
}
```

2. In your View, set the Grid element **Background** property to a type of **Color**, and then add a **Button** and **TextBox**.

3. Modify the XAML markup to bind the Button element to your source method and the TextBox.Text property to your source property.

```
<Grid Background="LightGray">
    <Button x:Name="ShowMyMessage"
            Content="Button" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Width="75"/>
    <TextBox Text="{Binding Path=MyMessage, Mode=TwoWay}"
             HorizontalAlignment="Left" Height="23" Margin="90,10,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="200"/>
</Grid>
```

4. Debug the application to test the data binding between the ViewModel and View.



Customizing a pane

You can edit inherited properties of a pane to define a preferred location, group and context for a pane at runtime.

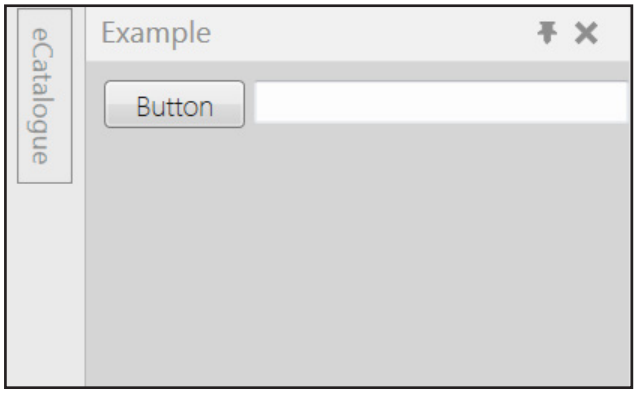
1. In your project, add a reference to **Create3D.Shared** that is located in your Essentials program files.

2. In your ViewModel, add a using statement for **VisualComponents.Create3D**.

3. In the ViewModel constructor, set the **PaneLocation**, **TabGroupId** and **ContextFilter** properties.

```
...
    using VisualComponents.UX.Shared;
    using VisualComponents.Create3D;

    [Export(typeof(IDockableScreen))]
    class ExampleViewModel : DockableScreen
    {
        public ExampleViewModel()
        {
            this.DisplayName = "Example";
            this.PaneLocation = (int) DesiredPaneLocation.DockedLeft;
            this.TabGroupId = "ExampleGroup";
            this.ContextFilter = Contexts.All;
        }
```

4. Debug the application to verify the updated changes to your pane in Essentials.



Adding panes on demand

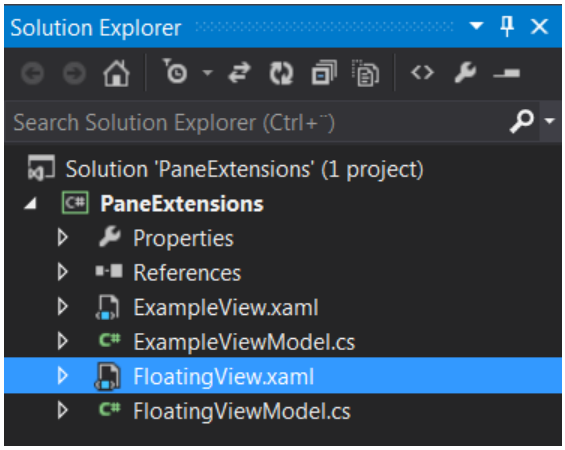
A floating or docking pane can be created on demand during runtime by using the IDockAwareWindowManager.

1. In your project, create a new **ViewModel** that is not exported as a type of IDockableScreen.

```
namespace PaneExtensions
{
    using Caliburn.Micro;
    using System.ComponentModel.Composition;
    using VisualComponents.UX.Shared;

    class FloatingViewModel : DockableScreen
    {
        public FloatingViewModel()
        {
            this.DisplayName = "Floating Example";
        }
    }
}
```

2. Create a new **View** that is associated with your new ViewModel, and then add a **ListBox**.

```
<UserControl x:Class="PaneExtensions.FloatingView"
             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"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid Background="LightGray">
        <ListBox HorizontalAlignment="Left" Height="280" Margin="10,10,0,0" VerticalAlignment="Top" Width="280"/>
    </Grid>
</UserControl>
```



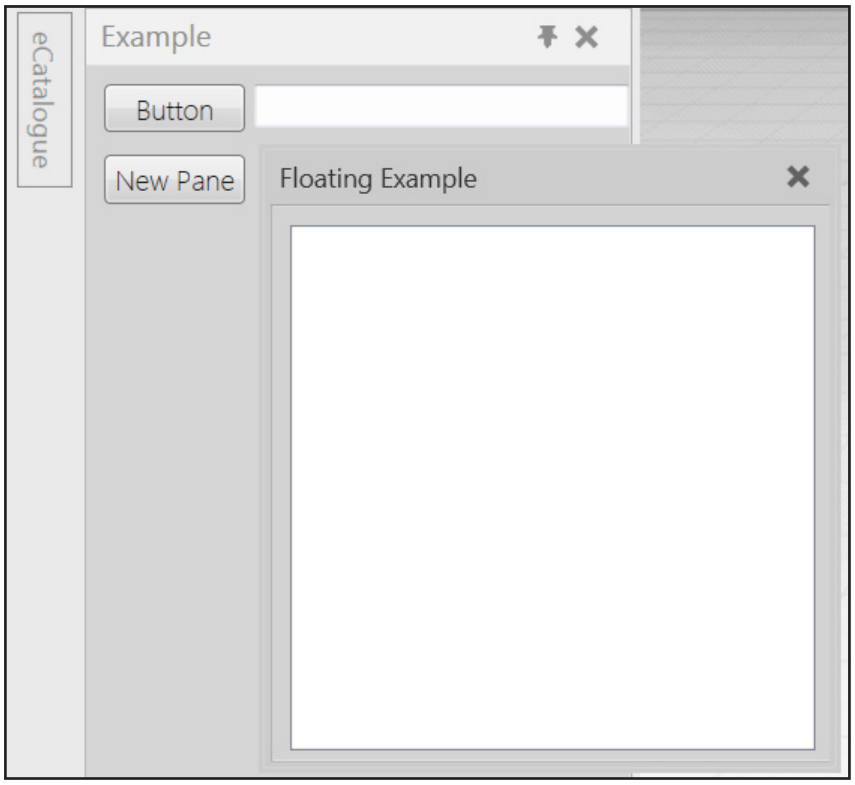
3. In your exported ViewModel, import a type of **IDockAwareWindowManager**, and then add a **public method** that allows you to create a new floating pane on demand.

```
    [Import]
    private Lazy<IDockAwareWindowManager> _windowManager { get; set; }
...
    public void AddFloatingPane()
    {
        _windowManager.Value.ShowFloatingWindow(String.Empty, new FloatingViewModel());
    }
```

4. In your exported View, add a **Button** that is binded to your source method for creating panes.

```
<Grid Background="LightGray">
    <Button x:Name="ShowMyMessage"
            Content="Button" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Width="75"/>
    <TextBox Text="{Binding Path=MyMessage, Mode=TwoWay}"
             HorizontalAlignment="Left" Height="23" Margin="90,10,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="200"/>
    <Button x:Name="AddFloatingPane"
        Content="New Pane" HorizontalAlignment="Left" Margin="10,48,0,0" VerticalAlignment="Top" Width="75"/>
</Grid>
```

5. Debug the application to test the creation of floating panes in your extension.



Adding a pane with multiple views

A floating or docking pane can use a Conductor class to handle multiple views.

1. In your project, create a **ViewModel** and **View** to represent a Home screen in a pane.

```
namespace PaneExtensions
{
    using Caliburn.Micro;

    class HomeViewModel: Screen
    {
        public HomeViewModel() { this.DisplayName = "Home"; }
    }
}
...
<UserControl x:Class="PaneExtensions.HomeView"
             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"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid Background="White">
        <TextBlock HorizontalAlignment="Left" Margin="10,10,0,0" TextWrapping="Wrap" Text="HOME" VerticalAlignment="Top"/>
    </Grid>
</UserControl>
```

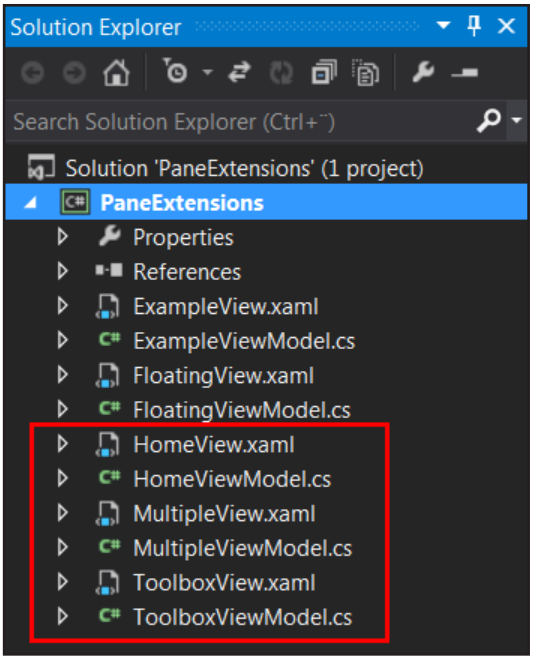
2. Create another **ViewModel** and **View** to represent a Toolbox screen in a pane.

```
namespace PaneExtensions
{
    using Caliburn.Micro;

    class ToolboxViewModel: Screen
    {
        public ToolboxViewModel() { this.DisplayName = "Toolbox"; }
    }
}
...
<UserControl x:Class="PaneExtensions.ToolboxView"
             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"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid Background="White">
        <TextBlock HorizontalAlignment="Left" Margin="10,10,0,0" TextWrapping="Wrap" Text="TOOLBOX" VerticalAlignment="Top"/>
    </Grid>
</UserControl>
```

3. Create another **ViewModel** and **View** to be the conductor of screens in a pane.

```
namespace PaneExtensions
{
    using Caliburn.Micro;
    using System.ComponentModel.Composition;
    using VisualComponents.UX.Shared;
    using VisualComponents.Create3D;

    class MultipleViewModel: Conductor<Screen>.Collection.OneActive
    {
        BindableCollection<Screen> views = new BindableCollection<Screen>();
        public MultipleViewModel()
        {
            this.DisplayName = "Multiple Example";
            this.ContextFilter = Contexts.All;
            views.Add(new HomeViewModel());
            views.Add(new ToolboxViewModel());
            views.Select(v => { v.Parent = this; return v; }).ToList();
            Items.AddRange(views);
        }
    }
}
...
<UserControl x:Class="PaneExtensions.MultipleView"
             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"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <TabControl x:Name="Items" />
    <!--Generally, a ContentControl element is used to show ActiveItem in Items collection-->
</UserControl>
```

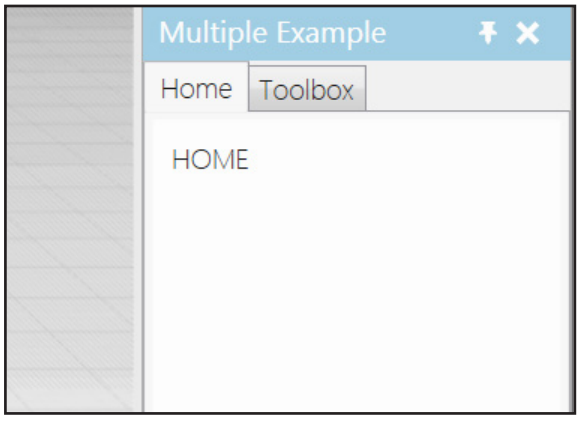
At this stage, your conductor ViewModel can be called on demand or exported as a type of IDockableScreen.



4. Export your conductor ViewModel as a type of **IDockableScreen**, and then implicitly implement members in **IDockableScreen**.

```
    [Export(typeof(IDockableScreen))]
    class MultipleViewModel: Conductor<Screen>.Collection.OneActive, IDockableScreen
...
        #region IDockableScreen members
        private string _contextFilter = "";
        public string ContextFilter
        {
            get { return _contextFilter; }
            set { _contextFilter = value; }
        }

        public int DesiredPanePosition { get { return (int) DesiredPaneLocation.DockedRight; } }
        public double Height { get { return double.NaN; } }
        public bool IsPinned
        {
            get { return true; }
            set { }
        }

        public DesiredPaneLocation PaneLocation { get { return DesiredPaneLocation.DockedRight; } }
        public System.Windows.Controls.Orientation PaneOrientation
        {
            get { return System.Windows.Controls.Orientation.Vertical; }
        }

        public string PanelId { get { return "multiplePanel"; } }
        public string SplitPaneId { get { return ""; } }
        public string TabGroupId { get { return ""; } }
        public double Width { get { return double.NaN; } }
        public bool IsVisible { get { return true; } set { } }
        #endregion
```

5. Debug the application to test that one of your panes has multiple views.



This concludes the tutorial.