CSC310-A

Module 2

Implementing a Dynamic User-Interface Framework

1 Application Shell

1.1 Application

In this module we will begin by creating an application project to build the application shell using the following information. Once the project has been created, there will be two XAML documents named **App.xaml** and **MainWindow.xaml**, and the associative source files **App.xaml.cs** and **MainWindow.xaml.cs** for the code-behind classes for each document. One extends **Application** class and another extends **Window** class. You can rename the XAML documents to **MyApp** and **Shell** but that does not change the class name. You have to specifically rename the classes as they do not need to be the same as the XAML document name.

Application project information

```
Project Name: SymBank
Project Type: Visual C# | Windows | WPF Application
Location    : C:\CSDEV\SRC
Solution    : Module2
```

In this module you will begin to implement a smart client application basing on the infrastructure that you have implemented in the previous module. You can start by setting up the solution for the application, the framework, plus standard and optional modules. Bring the **Symbion** and **SymBank.Banking** projects into the solution and then add in the following new projects.

Once completed, add another project named **SymBank.Shared** to contain all types and interfaces to be shared between the application and the other modules. Since the application reference modules, those modules cannot reference the application. Types should be placed in a separate assembly to be accessible for all assemblies.

Library project information

```
Project Name: SymBank.Shared
Project Type: Visual C# | Windows | WPF User Control Library
Location    : C:\CSDEV\SRC\Module2
```

Dependencies

```
SymBank         : Symbion, SymBank.Shared, SymBank.Banking
SymBank.Banking : Symbion, SymBank.Shared
SymBank.Shared  : Symbion
```

Also add the **Modules.xml** document into the application and as **Content** and set it to copy to output directory if file has been changed. This will be used to load in all the modules configured for the application.

In **MyApp.xaml**, we attach handlers for **Startup**, **DispatcherUnhandledException** and **Exit** events to the **Application** object. You can check the code-behind class in **MyApp.xaml.cs** to see the generated event handlers. As we have changed the name of the Window XAML document make sure to update the **StartupUri** property.

Attaching Application event handlers: MyApp.xaml

```
<Application x:Class="SymBank.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Shell.xaml"
    Startup="Application_Startup"
    DispatcherUnhandledException="Application_DispatcherUnhandledException"
    Exit="Application_Exit">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Setup initial services,handle exceptions and load modules: MyApp.xaml.cs

```
private void Application_Startup(object sender, StartupEventArgs e) {
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    new PrincipalAuthorization().Add<IAuthorization>();
    LoggerFactory.CreateInstance().Add();
    ModuleLoader.Load("Modules.xml");
}

private void Application_DispatcherUnhandledException(object sender, …) {
    if (Debugger.IsAttached) Debugger.Break(); else {
        var log = ServiceRepository.Get<ILogger>()
        if (log != null) log.Failure(e.Exception.ToString());
    }
}

private void Application_Exit(object sender, ExitEventArgs e) {
    ModuleLoader.Exit();
}
```

1.2 Application Shell

The application shell represents the container for the UI of the application. Since the modules may need to access the shell, it is crucial to initialize the modules only when the shell is ready. You can do this by attaching an event handler to the loaded event of the **Window** class.

Properties and events for main window: Shell.xaml

```
<Window x:Class="SymBank.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SymBank System"
    WindowStartupLocation="CenterScreen"
    Width="600" Height="400"
    Loaded="Window_Loaded">
    <Grid></Grid>
</Window>
```

Basic WPF window class: Shell.xaml.cs

```
namespace SymBank {
    public partial class Shell : Window {
        public Shell() {
            InitializeComponent();
        }
        private void Window_Loaded(object sender, RoutedEventArgs e) {
            ModuleLoader.Init();
        }
    }
}
```

We will now add some basic UI elements in our window. The **Grid** and **DockPanel** are layout panels. They are used to allocate window space so that you can place multiple UI elements in the window. They can help to resize and rearrange the UI elements automatically when the window size is changed. In the window, we will add a **Menu**, a **ToolBar** and a **StatusBar**. In the **StatusBar** we will have a **Label** control named as **lblStatus**. Giving UI elements a name will allow you to access them from program code later. There is also a inner Grid that is used to contain the rest of the content.

Basic UI layout and controls for the window

```
<Grid x:Name="shellLayout">
    <DockPanel>
            <Menu x:Name="mbrMain" DockPanel.Dock="Top"></Menu>
            <ToolBar x:Name="tbrMain" DockPanel.Dock="Top"></ToolBar>
            <StatusBar x:Name="sbrMain" DockPanel.Dock="Bottom">
                <Label x:Name="lblStatus" Content="Ready." />
            </StatusBar>
            <Grid x:Name="shellWorkspace"></Grid>
    </DockPanel>
</Grid>
```

Modules, services and views commonly need to interact with the user-interface. We can use the main window as the application shell that will provide UI services to these components and allow UI components to be displayed inside the shell window. Add an **IShell** interface to Symbion to expose some basic UI features to display status and messages.

We will add more features at a later time. Once the interface is declared, you can return to the SymBank project and implement the interface in the **Shell** class. Since it is now a service, you can now register it with **ServiceRepository**. Since the **Add** method is an extension method for **IService**, which is supported by the shell, you should see this method appearing in the members of the shell. You can call the method in the constructor to register the service.

Declaring a basic shell service: IShell.cs

```
namespace Symbion {
    public interface IShell : IService {
        object Status { set; }
        void Success(string message);
        void Failure(string message);
    }
}
```

Implementing the service: SymBank\Shell.xaml.cs

```
public partial class Shell : Window, IShell {
    public Shell() {InitializeComponent(); }
    public object Status { set { lblStatus.Content = value; }}
    public void Success(string message) {
        MessageBox.Show(this, message, "Information",
            MessageBoxButton.OK, MessageBoxImage.Information); }
    public void Failure(string message) {
        MessageBox.Show(this, message, "Error",
            MessageBoxButton.OK, MessageBoxImage.Error);  }
                    :
}
```

Self-registration the shell service

```
public Shell() {
    InitializeComponent();
    this.Add<IShell>();
}
```

Once the service is registered, it will only take one single statement to fetch it from anywhere in the entire application, module, service or view. However since the shell is a standard service, we can provide code to access the shell that can be inherited by all modules and services. Both classes can provide a **Shell** property to automatically fetch the shell service and retain it for future calls so that it does not have to lookup the object again.

Class to share code between modules: Symbion\BaseModule.cs

```
using System;

namespace Symbion {
    public class BaseModule : IModule {
        private static IShell _shell;
        public static IShell Shell {
            get {
                return _shell ?? (_shell =
                    ServiceRepository.Get<IShell>());
            }
        }
        public virtual void Init() { }
        public virtual void Exit() { }
    }
}
```

Class to share code between all services: BaseService.cs

```
namespace Symbion {
    public class BaseService : IService {
        private static IShell _shell;
        public static IShell Shell {
            get {
                return _shell ?? (_shell =
                    ServiceRepository.Get<IShell>()); }
        }
    }
}
```

2 Regions & Views

2.1 Regions & Views

To create a very flexible and dynamic user-interface, we can implement regions and views. Regions represent the locations in the application that allows you to insert views. A view represents a self-contained user-interface dedicated to completing a specific business process.

Inside our shell, we have a Grid as the main content area inside the DockPanel. We can use the Grid to split up the area into multiple columns and rows where each cell can be a separate region. In each region, you need have a control to assign the view to. You can use a **ContentControl** if the region can only have one view. A **Label** is a common ContentControl. However, to be able to assign multiple views to a single region, use an **ItemsControl** instead. **ListBox** and **TabControl** are different types of ItemsControl. For our example, we will create two regions in our shell. There will be three columns in the grid for both region controls and also a GridSplitter to allow the user to resize the columns.

Creating regions in the shell: Shell.xaml

```
<Grid x:Name="shellWorkspace">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <ListBox Grid.Column="0"
        x:Name="sideRegion"
        Visibility="Collapsed" />
    <GridSplitter Grid.Column="1"
        HorizontalAlignment="Left"
        Width="2" />
    <TabControl Grid.Column="2"
        x:Name="mainRegion"
        Visibility="Collapsed" />
</Grid>
```

You need to identity each region either by name or by a number. In our project, we would use strings to identity regions. You should add a class to declare constants to store the identity of the available regions. Since it depends on the actual application to decide the available regions, the class should be added to **SymBank.Shared** rather than **Symbion**.

Class to declare constants for available regions: ShellRegions.cs

```
namespace SymBank.Shared {
    public static class ShellRegions {
        public const string SideRegion = "side";
        public const string MainRegion = "main";
    }
}
```

We will now declare an interface in Symbion to describe the basic requirements for all views. This ensures that we can obtain enough information for the view to be used through the shell. Each view should specify which region it should be displayed in and an optional header to show. A **Show** method can be called to initialize and display the view and **Close** method to close the view.

Basic interface for views: Symbion\IView.cs

```
namespace Symbion {
    public interface IView {
        string Region { get; set; }
        object Header { get; set; }
        void Show();
        void Close();
    }
}
```

2.2 Region Adapters

Adapters are middle-tier or wrapper components that allows us to interface to objects that by themselves are not compatible because they lack certain features. Adapters will provide the implementation for those missing requirements. We will first declare an interface to provides region support for views.

A adapter to integrate views to regions: Symbion\IRegionAdapter.cs

```
public interface IRegionAdapter {
    public void Add(IView view);
    public void Remove(IView view);
}
```

Regions adapters can then be exposed from the shell through the IShell interface so external code can add views or remove views to any region. Add a **Regions** property to **IShell** interface that returns a dictionary of **IRegionAdapters** that you can locate a specific adapter for the region by using its identity.

Adding view support to shell service: Symbion\IShell.cs

```
namespace Symbion {
    public interface IShell : IService {
        object Status { set; }
        void Success(string message);
        void Failure(string message);
        Dictionary<string, IRegionAdapter> Regions { get; }
    }
}
```

Example of fetching a region adapter from the shell

```
IShell shell = ServiceRepository.Get<IShell>();
string regionId = ShellRegions.SideRegion;
IRegionAdapter regionAdapter = shell.Regions[regionId];
regionAdapter.Add(new ExampleView());
```

We will now implement two region adapters to support **ListBox** and **TabControl**. This will be UI platfom dependent. So far our Symbion library is not dependent on using a particular UI system. If you wish to support multiple UI systems, UI dependent types should be placed into separate assemblies; e.g. Symbion.WPF, Symbion.Silverlight, Symbion.WinForms. For our example, the Symbion library supports only WPF so we can integrate UI dependent types into the same library project. You must add the references to the following assemblies to use WPF types in your library.

WPF assemblies to reference:

```
PresentationCore
PresentationFramework
System.Xaml
WindowsBase
```

Views are not directly added to the regions. You need to have a host control for each view. It is the host control that is added to the region. You have to decide which type of host control you would prefer to use for each region. For our side region, we will use **Expander** control. For main region, you really do not have a choice since only **TabItem** control can be used. In our code, we need to have a dictionary for each region to associate the host control with the view so that we can remove the host later.

Region adapter for WPF ListBox control: ListBoxRegionAdapter.cs

```
public class ListBoxRegionAdapter : IRegionAdapter {
    private ListBox _region;
    private Dictionary<IView, Expander> _views;
    public ListBoxRegionAdapter(ListBox region) {
        _views = new Dictionary<IView, Expander>();
        _region = region;
    }
    public void Add(IView view) {
        var host = new Expander();
        _views.Add(view, host);
        host.Header = view.Header;
        host.Content = view;
        _region.Items.Add(host);
        _region.SelectedItem = host;
        if (!_region.IsVisible) _region.Visibility = Visibility.Visible;
    }

    public void Remove(IView view) {
        var host = _views[view];
        _region.Items.Remove(host);
        _views.Remove(view);
        if (_views.Count == 0) _region.Visibility = Visibility.Collapsed;

    }
}
```

Region adapter for TabControl: TabControlRegionAdapter.cs

```
public class TabControlRegionAdapter : IRegionAdapter {
    private TabControl _region;
    private Dictionary<IView, TabItem> _views;
    public TabControlRegionAdapter(TabControl region) {
        _views = new Dictionary<IView, TabItem>();
        _region = region;
    }
    public void Add(IView view) {
        var host = new TabItem();
        _views.Add(view, host);
        host.Header = view.Header;
        host.Content = view;
        _region.Items.Add(host);
        _region.SelectedItem = host;
        if (!_region.IsVisible) _region.Visibility = Visibility.Visible;
    }
    public void Remove(IView view) {
        var host = _views[view];
        _region.Items.Remove(host);
        _views.Remove(view);
        if (_views.Count == 0) _region.Visibility = Visibility.Collapsed;
    }
}
```

Implementing support for regions in shell: Shell.xaml.cs

```
private Dictionary<string, IRegionAdapter> _regions;

public Dictionary<string, IRegionAdapter> Regions {
    get { return _regions; }
}

public Shell() {
    InitializeComponent();
    _regions = new Dictionary<string, IRegionAdapter>();
    _regions.Add(ShellRegions.SideRegion, new ListBoxRegionAdapter(sideRegion));
    _regions.Add(ShellRegions.MainRegion, new TabControlRegionAdapter(mainRegion));
    this.Add<IShell>();
}
```

We will now provide a default implementation of **IView** to be inherited by all of the views. We normally use **UserControl** elements as views so you will need to extend your **BaseView** class from it. You can then implement an example of a view. Add a UserControl to **SymBank.Shared** project named **WebBrowserView**. To inherit code from BaseView instead, you need to first register a XML namespace since CLR namespaces cannot be used inside a XAML file. Then use the XML namespace and **BaseView** class to replace UserControl in the XAML file. You can then change the base class in the class source file. In the constructor, you can specify what region the view should be displayed in and also provide a title for the view. This will satisfy the basic requirement for the view.

Default implementation for views: Symbion\BaseView.cs

```
using System;
using System.Windows.Controls;

namespace Symbion {
    public class BaseView : UserControl, IView {
        private static IShell _shell;
        private string _region;
        private object _header;
        public virtual string Region {
            get { return _region; }
            set { _region = value; }
        }
        public virtual object Header {
            get { return _header; }
            set { _header = value; }
        }
        public static IShell Shell {
            get { return _shell ?? (_shell = ServiceRepository.Get<IShell>()); }
        }
        public virtual void Show() {
            IRegionAdapter region = _shell.Regions[Region];
            region.Add(this);
        }
        public virtual void Close() {
            IRegionAdapter region = _shell.Regions[Region];
            region.Remove(this);
        }
    }
}
```

Registering and changing the base class in XAML: WebBrowserView.xaml

```
<fx:BaseView x:Class="SymBank.Shared.WebBrowserView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:fx="clr-namespace:Symbion;assembly=Symbion"
    Loaded="BaseView_Loaded">
</fx:BaseView>
```

Changing base class and setting up the view: WebBrowser.xaml.cs

```
using System;
using System.Windows;
using System.Windows.Controls;
using Symbion;

namespace SymBank.Shared {
    public partial class WebBrowserView : BaseView {
        public WebBrowserView() {
            InitializeComponent();
            Region = ShellRegions.MainRegion;
            Header = "Web Browser";
        }
    }
}
```

In the view, we will have a toolbar with buttons that contain images. First add a folder called *Images* into the **SymBank.Shared** project. Then add a group of bitmap image files provided by the instructor into the older. Open up the WebBrowserView XAML file and add the following DockPanel and content into the Grid layout. Then add the following code to the source file for the view.

To call up a view from the shell, we will add a menu item and a button on the toolbar. Both the menu item and the toolbar button will have an image. Just like the previous project, add *Images* folders into the SymBank project. Then add the image provided by the instructor into the *Images* folder.

Main content of the view: WebBrowserView.xaml

```
<DockPanel>
    <ToolBar DockPanel.Dock="Top">
        <Button x:Name="btnPrevious" Click="btnPrevious_Click">
            <Image Stretch="None" Source="Images/previous.png" />
        </Button>
        <Button x:Name="btnNext" Click="btnNext_Click">
            <Image Stretch="None" Source="Images/next.png" />
        </Button>
        <Button x:Name="btnRefresh" Click="btnRefresh_Click">
            <Image Stretch="None" Source="Images/refresh.png" />
        </Button><Separator />
        <TextBox x:Name="txtLocation" MinWidth="400" />
        <Button x:Name="btnOpen" Click="btnOpen_Click">
            <Image Stretch="None" Source="Images/world.png" />
            </Button>
        <Button x:Name="btnClose" Click="btnClose_Click">
            <Image Stretch="None" Source="Images/cancel.png" />
        </Button>
    </ToolBar><WebBrowser x:Name="webBrowser"/>
</DockPanel>
```

Method to open a site with the web browser

```
public void Open(string path) {
    try { Uri location = new Uri(path, UriKind.Absolute); webBrowser.Navigate(location); }
    catch (Exception ex) { Shell.Failure("Cannot open site. " + ex.Message); }
}
```

Ensure text box has focus when view is loaded

```
private void BaseView_Loaded(object sender, RoutedEventArgs e) { txtLocation.Focus(); }
```

Event handlers for buttons in the toolbar

```
private void btnRefresh_Click(object sender, RoutedEventArgs e) { webBrowser.Refresh(); }
private void btnPrevious_Click(object sender, RoutedEventArgs e) {
    if (webBrowser.CanGoBack) webBrowser.GoBack();
}
private void btnNext_Click(object sender, RoutedEventArgs e) {
    if (webBrowser.CanGoForward) webBrowser.GoForward();
}
private void btnOpen_Click(object sender, RoutedEventArgs e) { Open(txtLocation.Text); }
private void btnClose_Click(object sender, RoutedEventArgs e) {  Close(); }
```

MenuItem to be used to open the view: Shell.xaml

```
<Menu x:Name="menuBar" DockPanel.Dock="Top">
       <MenuItem Header="_Tools">
         <MenuItem
             x:Name="mnuWebBrowser"
             Header="_Web Browser"
             Click="mnuWebBrowser_Click">
             <MenuItem.Icon>
                     <Image Stretch="None"
                        Source="Images/world.png" />
             </MenuItem.Icon>
         </MenuItem>
     </MenuItem>
</Menu>
```

Button to be used to open the view

```
<ToolBar x:Name="toolBar" DockPanel.Dock="Top">
        <Button x:Name="btnWebBrowser"
            Tooltip="Open web browser"
            Click="mnuWebBrowser_Click">
            <Image Stretch="None"
               Source="Images/world.png" />
        </Button>
</ToolBar>
```

The event handler to instantiate and show the view

```
private void mnuWebBrowser_Click(object sender, RoutedEventArgs e) {
//      new WebBrowserView().Show();
        WebBrowserView view = new WebBrowserView();
        view.Open("http://www.google.com");
        view.Show();
}
```

3 Commands & Builders

3.1 Commands

Instead of using event handlers to run code, we can implement commands that are pre-programmed to perform certain actions. You can then instantiate them and bind them to **MenuItem** or **Button** controls using the **Command** property. There is also a **CommandParameter** property allowing you to pass a single object to the command. Any class can be a command as long as it implements the **ICommand** interface. This interface has two methods called **CanExecute** and **Execute**, and an event named **CanExecuteChanged**. CanExecute is called automatically when CommandParameter changes or when event is raised. If CanExecute returns true, then Execute method will be called when the user selects the menu item or click on the button otherwise the UI element will be disabled.

In the following example, we will create a command to open the web browser view. The parameter will be the default URL to be used in the browser when it is opened. Add a *Commands* folder to SymBank.Shared project. In the folder, add a new class named **OpenBrowserCommand**. Extend from **BaseCommand** class to inherit the existing code. Override both CanExecute and Execute methods with the code shown in the following page. The command should be executable as long as the parameter is not null. Once the command is finished, we need to make it available to XAML. The simplest way to expose a command is to declare a static property to assign the command object. XAML can use a **Static** binding expression to bind to static fields or properties.

Implementing a command: Commands\OpenBrowserCommand.cs

```
using System;
using Symbion;

namespace SymBank.Shared.Commands {
    public class OpenBrowserCommand : ICommand {
        public event EventHandler CanExecuteChanged;
        public override bool CanExecute(object parameter) {
            return parameter != null;
        }
        public override void Execute(object parameter) {
            WebBrowserView view = new WebBrowserView();
            view.Open(parameter.ToString());
            view.Show();
        }
    }
}
```

Exposing command object with a static field: MyCommand.cs

```
namespace SymBank.Shared.Commands {
    public static class MyCommands {
        public static readonly OpenBrowserCommand OpenBrowser =
            new OpenBrowserCommand();
    }
}
```

To use the above command object in the shell, first register the XML namespace for the CLR namespace of **MyCommands** class. Set Command and CommandParameter properties of a menu item or button. Use a **Static** binding expression to bind to the command object. You can then test out the binding by selecting the menu item or click on the button.

Register namespace for commands in shell: Shell.xaml

```
<Window x:Class="SymBank.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:SymBank.Shared.Commands;assembly=SymBank.Shared"
            :
```

Binding to a command from a button

```
<Button ToolTip="Open web browser"
        CommandParameter="http://www.google.com"
        Command="{x:Static c:MyCommands.OpenBrowser}">
        <Image Stretch="None"
               Source="Images/world.png" />
</Button>
```

3.2 Delegate Command

Implementing a class for each command may be a bit tedious especially if there is very little action code in the command. There is a special **DelegateCommand** class provided in Prism framework to simplify command implementation. Even though we are not using Prism it will be very easy to create this class ourselves. Basically the class uses delegates to run the action. This will allow us to just use one class but provide different actions.

Implementation of Delegate command: Symbion\DelegateCommand.cs

```
using System;

namespace Symbion {
    public class DelegateCommand : ICommand {
        private Predicate<object> _canExecute;
        private Action<object> _execute;

        public DelegateCommand(Action<object> execute) {
            _canExecute = null;
            _execute = execute;
        }
        public DelegateCommand(
            Predicate<object> canExecute,
            Action<object> execute) {
            _canExecute = canExecute;
            _execute = execute;
        }
        public bool CanExecute(object parameter) {
            if (_canExecute == null) return true;
            return _canExecute(parameter);
        }
        public void Execute(object parameter) {
            if (_execute != null)
                _execute(parameter);
        }
        public event EventHandler CanExecuteChanged;
        public void NotifyCanExecuteChanged() {
            if (CanExecuteChanged != null)
                CanExecuteChanged(this,
                EventArgs.Empty);
        }
    }
}
```

We can now add in a new command that can run an external process. You can pass a string that contains the information required to launch the process. The following shows how to use a DelegateCommand. Add a menu item to the *Tools* menu to be used to launch *notepad* application using **Open** command we created below.

Creating command for custom action: MyCommand.cs

```
namespace SymBank.Shared.Commands {
    public static class MyCommands {
        public static readonly ICommand Open = new DelegateCommand(
            p => p != null, p => Process.Start(p.ToString()));
```

Binding to a DelegateCommand

```
<MenuItem Header="_Notepad"
    CommandParameter="notepad"
    Command="{x:Static c:MyCommands.Open}" />
```

3.3 Builders

While we can directly add menu items and buttons to open up standard views, this will not be possible for dynamic views. If a dynamic module is not loaded, then the views will not be available and thus menu items and buttons will also be unavailable. Menu items and buttons should only be created when and if the module is loaded. To make it easy for modules to create menu items and buttons and add them to the shell, we will implement builder classes. The following is code for a **MenuBarBuilder**. You can use the **Create** method to create a menu or menu item and pre-bind it to a command. Then the item can be added to the menu bar by using **AddMenu** or add as a sub-item for an existing menu by using **AddItem**. If the menu is shared between modules, you can use **GetMenu** or **GetItem** to access existing menu or item rather than creating it.

Class to help build a menu system: Symbion\MenuBarBuilder.cs

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Symbion {
    public class MenuBarBuilder {
        private Menu _menuBar;

        public MenuBarBuilder(Menu menuBar) { _menuBar = menuBar; }

        public MenuItem Create(
            string name,
            string text,
            ICommand command = null,
            object commandParameter = null,
            string icon = null) {
            MenuItem item = new MenuItem();
            item.Name = name;
            item.Header = text;
            item.Command = command;
            item.CommandParameter = commandParameter;
            if (icon != null) {
                Image image = new Image();
                image.Stretch = Stretch.None;
                image.Source = new BitmapImage(
                    new Uri(icon, UriKind.Relative));
                item.Icon = image;
            }
            return item;
        }
        public void AddMenu(MenuItem item) {
            _menuBar.Items.Add(item);
        }
        public void AddItem(MenuItem menu, MenuItem item) {
            menu.Items.Add(item);
        }
        public void AddSeparator(MenuItem menu) {
            menu.Items.Add(new Separator());
        }
        public MenuItem GetMenu(string name) {
            foreach (FrameworkElement item in _menuBar.Items)
                if (item.Name.Equals(name))
                    return (MenuItem)item;
            return null;
        }
        public MenuItem GetItem(MenuItem menu, string name) {
            foreach (FrameworkElement item in menu.Items)
                if (item.Name.Equals(name))
                    return (MenuItem)item;
            return null;
        }
    }
}
```

The following is the code for a **ToolBarBuilder**. We can use **GetItem** to attempt to retrieve an existing control on the toolbar. The toolbar can be assigned any kind of control, not only buttons. However, in this example we only provide a **CreateButton** method to create specifically a button control. You can add additional methods like **CreateTextBox** or **CreateCheckBox** methods to add support for other types of controls. Once you have created the right control you can call the **AddItem** method to add the control into the toolbar.

Class to help build the toolbar: Symbion\ToolBarBuilder.cs

```
using System;
using System.Windows.Controls;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media.Imaging;
using System.Windows.Media;

namespace Symbion {
    public class ToolBarBuilder {
        private ToolBar _toolBar;

        public ToolBarBuilder(ToolBar toolBar) { _toolBar = toolBar; }

        public FrameworkElement GetItem(string name) {
            foreach (FrameworkElement item in _toolBar.Items)
                if (item.Name.Equals(name)) return item;
            return null;
        }
        public Button CreateButton(
            string name,
            string text,
            ICommand command = null,
            object commandParameter = null,
            string icon = null) {
            Button item = new Button();
            item.Name = name;
            item.ToolTip = text;
            item.Command = command;
            item.CommandParameter = commandParameter;
            if (icon != null) {
                Image image = new Image();
                image.Stretch = Stretch.None;
                image.Source = new BitmapImage(
                    new Uri(icon, UriKind.Relative));
                item.Content = image;
            }
            return item;
        }
        public void AddItem(FrameworkElement item) {
            _toolBar.Items.Add(item);
        }
        public void AddSeparator() { _toolBar.Items.Add(new Separator()); }
    }
}
```

We can expose the builders to modules, services and views through **IShell** interface. Modify the interface and add two properties; **MenuBars** and **ToolBars** to expose one or more builders using dictionaries. We can then create and expose the builders in our actual **Shell** class. In the XAML file, we can access the controls using their names to encapsulate them in the builders.

Exposing UI builders through the shell service: IShell.cs

```
public interface IShell : IService {
            :
    Dictionary<string, MenuBarBuilder> MenuBars { get; }
    Dictionary<string, ToolBarBuilder> ToolBars { get; }
}
```

Code to create and expose the builders: Shell.xaml.cs

```
private Dictionary<string, MenuBarBuilder> _menuBars;
private Dictionary<string, ToolBarBuilder> _toolBars;

public Shell() {
        :
    _menuBars = new Dictionary<string, MenuBarBuilder>();
    _toolBars = new Dictionary<string, ToolBarBuilder>();
    _menuBars.Add("main", new MenuBarBuilder(mbrMain));
    _toolBars.Add("main", new ToolBarBuilder(tbrMain));
}

public Dictionary<string, MenuBarBuilder> MenuBars { get { return _menuBars; }}
public Dictionary<string, ToolBarBuilder> ToolBars { get { return _toolBars; }}
```

To test the builders we will have to perform some work on our dynamic module. In the **SymBank.Banking** project, add *Images* folders to contain images for the menu items or buttons that we will create dynamically. Add the images provided by the instructor. Then add two user controls into the project and then turn them into views by changing the base class to **BaseView**.

Creating a new view: AddAccountView.xaml

```
<fx:BaseView x:Class="SymBank.Banking.AddAccountView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:fx="clr-namespace:Symbion;assembly=Symbion"
    Loaded="BaseView_Loaded">
    <Grid></Grid>
</fx:BaseView>
```

Extending UserControl from BaseView: SearchAccountsView.xaml

```
<fx:BaseView x:Class="SymBank.Banking.SearchAccountsView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:fx="clr-namespace:Symbion;assembly=Symbion"
    Loaded="BaseView_Loaded">
    <Grid>
</Grid>
</fx:BaseView>
```

Extending and initializing the view: AddAccountView.xaml

```
using System;
using System.Windows;
using System.Windows.Controls;
using SymBank.Shared;
using Symbion;

namespace SymBank.Banking {
    public partial class AddAccountView : BaseView {
        public AddAccountView() {
            InitializeComponent();
            Region = ShellRegions.MainRegion;
            Header = "New Account";
        }
    }
}
```

Extending and initializing the view: SearchAccountsView.xaml.cs

```
public partial class SearchAccountsView : BaseView {
    public SearchAccountsView() {
        InitializeComponent();
        Region = ShellRegions.MainRegion;
        Header = "Account Search";
    }
}
```

When a module is loaded by **ModuleLoader**, the module will be initialized after the shell has been created by calling the **Init** method on the module. A module can then implement code in this method to add additional menu items and also buttons to the shell by using the builders. Following is code for **Init** method of the **BankingModule** that obtains an existing or create a new menu to assign a menu item to trigger a command to open up the above view. We will also add a button to the toolbar to perform the same command. Note that to access embedded resources in a WPF application or library you need to know how to construct a URL to the resource by using the *Pack URI* scheme. You can find this in your MSDN documentation.

Using builders in a dynamic module: BankingModule.cs

```
namespace SymBank.Banking {
    public class BankingModule : BaseModule {
        public override void Init() {
            MenuBarBuilder mb = Shell.MenuBars["main"];
            ToolBarBuilder tb = Shell.ToolBars["main"];
            var menu = mb.GetMenu("menu_accounts");
            if (menu == null) {
                menu = mb.Create("menu_accounts", "_Accounts");
                mb.AddMenu(menu);
            }
            var addAccountCommand = new DelegateCommand(p => new AddAccountView().Show());
            var addAccountIcon = "/SymBank.Banking;component/Images/user_add.png";
            var addAccountMenuItem = mb.Create("menuitem_addaccount",
                "_Add new Account", addAccountCommand, null, addAccountIcon);
            var addAccountButton = tb.CreateButton("button_addaccount",
                "Add new account", addAccountCommand, null, addAccountIcon);
            mb.AddItem(menu, addAccountMenuItem);
            tb.AddSeparator();
            tb.AddItem(addAccountButton);

            var searchAccountsMenuItem = mb.Create("menuitem_searchaccounts",
                "_Search for Accounts", new DelegateCommand(
                p => new SearchAccountsView().Show()));
            mb.AddItem(menu, searchAccountsMenuItem);

        }
    }
}
```

Run the application now and if the current user is authorized to use the module, then the menu items and button will appear and then the user can use them to open up the view. However if the module is not loaded, then the menu items and buttons will not be created. We will now finish up designing the content of the view and make it closeable.

Visual content of the view: AddAccountView.xaml

```
<ScrollViewer VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto">
    <Border CornerRadius="8"
        BorderThickness="2"
        BorderBrush="SteelBlue">
        <StackPanel Margin="8">
            <Label x:Name="txtCode">Code</Label>
            <TextBox Text="" />
            <Label>Customer Name</Label>
            <TextBox Text="" />
            <Label>Account Type</Label>
            <ComboBox SelectedIndex="0">
                <ComboBoxItem>Savings</ComboBoxItem>
                <ComboBoxItem>Fixed Deposit</ComboBoxItem>
                <ComboBoxItem>Checking</ComboBoxItem>
            </ComboBox>
            <Label>Zip Code</Label>
            <TextBox Text="" />
            <Label>Opening Balance</Label>
            <TextBox Text="" />
            <StackPanel Margin="8"
                HorizontalAlignment="Center"
                Orientation="Horizontal">
                <Button x:Name="btnAdd"
                    Content="OK"
                    Click="btnAdd_Click" />
                <Button x:Name="btnCancel"
                    Content="Cancel"
                    Click="btnCancel_Click" />
            </StackPanel>
        </StackPanel>
    </Border>
</ScrollViewer>
```

Event handler to close the view: AddAccountView.xaml.cs

```
private void btnCancel_Click(object sender, RoutedEventArgs e) { Close(); }
```

Visual content for view

```
<DockPanel>
    <ToolBar DockPanel.Dock="Top">
        <Label>Search</Label>
        <TextBox x:Name="txtSearch" BorderBrush="SteelBlue" MinWidth="300" />
        <Button x:Name="btnSearch" Click="btnSearch_Click">
            <Image Stretch="None" Source="Images/magnifier.png" />
        </Button><Separator />
        <Button x:Name="btnCancel"
            Click="btnCancel_Click">
            <Image Stretch="None" Source="Images/cancel.png" />
        </Button>
    </ToolBar>
    <ListBox x:Name="lsbAccounts" />
</DockPanel>
```

Event handler to close the view: SearchAccountsView.xaml.cs

```
private void btnCancel_Click(object sender, RoutedEventArgs e) { Close(); }
```

We cannot complete the views yet as we are still missing certain components. Type of components depends on which application architecture that you are going for. If you wish to develop an MVC application; we need to have Model and Controller support. If we wish to develop an MVVM application, we need to support Model and ViewModel. You can also begin with MVC and then replace the Controller with ViewModel. You can still retain the Controller and use it from the ViewModel by using the MVVM+C hybrid architecture.