WPF201-B

Module 5

Implementing MVVM Applications with WPF

Copyright ©
Symbolicon Systems
2011-2017

1 Commands & ViewModels

1.1 ICommand Interface

While a RoutedCommand is a command processed by an UI element, any object can a self-executing command by implementing **ICommand** interface. Following example shows a comand to open a view.

A self-executing command: Commands\OpenTextEditorCommand.cs

```
using System;
using System.Windows.Input;
using SymBank.Views;

namespace SymBank.Commands {
    public class OpenTextEditorCommand : ICommand {
        public event EventHandler CanExecuteChanged;
        public bool CanExecute(object parameter) {
            return true;
        }
        public void Execute(object parameter) {
            new TextEditorView().Show();
        }
    }
}
```

Commands are usually exposed through a View-Model in a MVVM application. If you don't have any View-Model or you want to implement a globally accessible command, you can also expose in a resource dictionary or use a static property or field. You can use **Static** binding markup extension to bind to constants, static properties and fields. Add a **MyCommands** class to the **Commands** folder to expose the above command by just using a static readonly field.

Exposing commands to XAML with static fields: Commands\MyCommands.cs

```
namespace SymBank.Commands {
    public static class MyCommands {
        public static readonly OpenTextEditorCommand OpenTextEditor =
            new OpenTextEditorCommand();
    }
}
```

Commands can be optionally passed a **CommandParameter**. In the following code, the parameter is checked in **CanExecute** to ensure that it is an integer and then used in **Execute** as the exit code.

Checking and using parameter: Commands\ExitAppCommand.cs

```
using System;
using System.Windows;
using System.Windows.Input;

namespace SymBank.Commands {
    public class ExitAppCommand : ICommand {
        public event EventHandler CanExecuteChanged;
        public bool CanExecute(object parameter) {
            var exitCode = 0;
            return parameter != null &&
                int.TryParse(parameter.ToString(), out exitCode);
        }
        public void Execute(object parameter) {
            Application.Current.Shutdown(int.Parse(parameter.ToString()));
        }
    }
}
```

Expose command through MyCommands class: MyCommands.cs

```
public static readonly ExitAppCommand ExitApp = new ExitAppCommand();
```

Use the **Command** and **CommandParameter** properties in **MenuItem** or **Button** to bind and pass data to commands. You can register an XMLNS namespace to the **Commands** CLR namespace to access the **MyCommands** class.

Registering Commands namespace prefix: 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.Commands"
                :
```

Binding to commands in menu items

```
<MenuItem Header="E_xit"
    CommandParameter="0"
    Command="{x:Static c:MyCommands.ExitApp}">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource CancelImage}" />
    </MenuItem.Icon>
</MenuItem>
                :
<MenuItem Header="_Text Editor"
    Command="{x:Static c:MyCommands.OpenTextEditor}">
    <MenuItem.Icon>
        <Image Stretch="None" Source="{StaticResource NotepadImage}" />
    </MenuItem.Icon>
</MenuItem>
```

Binding to commands in buttons

```
<Button ToolTip="Open text editor"
    Command="{x:Static c:MyCommands.OpenTextEditor}">
    <Image Stretch="None" Source="{StaticResource NotepadImage}" />
</Button>
```

1.2 RelayCommand

The problem with commands is that one class is required for each command. This can be solved by not putting operation code in the command. Instead you can implement just one command class that will use delegates to run external code. You can declare fields to assign delegates for **CanExecute** and **Execute**. Add **RelayCommand** class to use **Action<object>** and **Predicate<object>** delegates to run external code. By default **CanExecute** is called only once unless **CanExecuteChanged** event is fired. Add a **NotifyCanExecuteChanged** method to simplify firing of the event to update the can execute state when required.

Using delegates to run operations: Commands\RelayCommand.cs

```
using System;
using System.Windows.Input;

namespace SymBank.Commands {
    public class RelayCommand : ICommand {
        private Action<object> _execute;
        private Predicate<object> _canExecute;

        public RelayCommand(Action<object> execute = null,
            Predicate<object> canExecute = null) {
            _execute = execute; _canExecute = canExecute;
        }

        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);
        }
    }
}
```

You can now remove both previous two command classes and modify **MyCommands** to use **RelayCommand** instead. All code can be placed within a single class such as **MyCommands** as named or anonymous methods that uses lambda expressions.

Using a single command class: Commands\MyCommands.cs

```
public static readonly RelayCommand OpenTextEditor =
    new RelayCommand(p => new TextEditorView().Show());

public static readonly RelayCommand ExitApp =
    new RelayCommand(
        p => Application.Current.Shutdown(int.Parse(p.ToString())),
        p => {
            var exitCode = 0;
            return p != null && int.TryParse(p.ToString(), out exitCode);
        }
    );
```

1.3 InputBindings

A RoutedCommand can be assigned key gestures that can be used to activate them. Even though the **ICommand** interface does not provide any support for this, you can use **InputBindings** on UI elements to capture key and mouse gestures and use them to activate commands. In our example we will capture **Ctrl-T** gesture if the key event is not handled and routed upwards until it reaches the **Window**. To let the user know what the key is, use **InputGestureText** property of the menu item to display it.

Binding command to key gesture: Shell.xaml

```
<Window.InputBindings>
    <KeyBinding Gesture="Ctrl+T"
        Command="{x:Static c:MyCommands.OpenTextEditor}" />
</Window.InputBindings>
```

Displaying the key gesture in a menu item

```
<MenuItem Header="_Text Editor"
    Command="{x:Static c:MyCommands.OpenTextEditor}"
    InputGestureText="Ctrl+T">
```

1.4 View Model

The concept of a view-model is to separate operations from a view. The view can use data-bindings to use or store data into a view model. Since data is in the view-model, it can also provide commands to the view to process the data. Whenever data in the view-model changes, a **PropertyChanged** event is required to be fired to refresh the UI. The UI uses the **INotifyPropertyChanged** interface to listen to this event. Add a folder to the project named **ViewModels** and add following code to share between all the view models.

Initial base class for view models: ViewModels\BaseViewModel.cs

```
using System;
using System.ComponentModel;
using System.Windows;

namespace SymBank.ViewModels {
    public class BaseViewModel : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;
        public void NotifyPropertyChanged(string propertyName) {
            if (PropertyChanged != null) PropertyChanged(this,
                    new PropertyChangedEventArgs(
                        propertyName));
        }
    }
}
```

We can provide better integration between view and the viewmodel. A view can fetch the view model from **DataContext**. It is then possible for the view to pass itself to the view model so that the view model can access the view.

Provide access to view from viewmodel: BaseViewModel.cs

```
private BaseView _view;
public BaseView View { get { return _view; } set { _view = value; }}
```

Integration to view model from view: BaseView.cs

```
private BaseViewModel _vm;
public BaseViewModel ViewModel { get { return _vm; } }
private void BaseView_Loaded(object sender, RoutedEventArgs e) {
    if (_vm == null) {
        _vm = (BaseViewModel)DataContext;
        if (_vm != null) _vm.View = this;
    }
    ResetFocus();
}
```

Since a view-model now has access to the view we can add in a method that can be used from the view-model to close the view. To allow bindings to be used as well to call the operation, you can expose a command as a property.

Method to close the view from view-model: BaseViewModel.cs

```
public void CloseView() { if (_view != null) _view.Close(); }
```

Exposed command to close views

```
public RelayCommand CloseViewCommand { get; private set; }
public BaseViewModel() {
        CloseViewCommand = new RelayCommand(p => CloseView());
}
```

2 Models & Controllers

2.1 Creating Database Model

Data models can also be persisted to databases. Visual Studio provides technologies to help you generate data models from databases such as Entity Data Model (EDM) and LINQ to SQL classes. In this example we will create a database and then generate a data model using LINQ to SQL. Run *SQL Server Management Studio* and execute *SymBank.sql* script provided by the instructor to create a database with two tables; **Accounts** and **Transactions** and a set of stored procedures. Once the database is ready add a *Data Connection* to the database using the *Server Explorer* window. Then open the **SymBank** project to generate the model. The data model that you generate will automatically support property change notifications.

Data Connection properties

```
Server Name      :   .\SQLEXPRESS
Authentication   :   Windows Authentication
Database         :   SymBank
```

Generate a data model with *LINQ to SQL Classes* template named as **SymBank.dbml** into a **Models** folder. Then drag and drop all tables in the *SymBank* database to the left pane and drag and drop all the stored procedures on the right pane. LINQ to SQL will generate one class for each table in your database to store individual entities and also generate an additional data context class to store and access the entities in the database. You can make modification to properties to make them more accessible. The **Type** column in **Account** model class is currently an **int** value. Add the **enum** type shown below to the **Models** folder and then change the type of property to the enum. Also change **Picture** type from **Binary** to **byte[]** as a converter is available to convert a **byte[]** into an **ImageSource** used by **Image** UI element.

A enumeration for account Type column: Models\AccountType.cs

```
namespace SymBank.Models {
    public enum AccountType {
        Savings,
        Checking,
        Loan
    }
}
```

Even though a view model can be used to do processing, complex processing can still be delegated to a separate class commonly called as a controller. Controllers are used directly by views when implementing the MVC pattern. In a MVVM pattern a controller can be replaced by the view-model or it can still use a controller.

2.2 Controller

You should implement a controller to help you perform data operations on the data model. Controllers provide services to perform tedious or complex operations. Add a **Controllers** folder to **SymBank** project containing the following class.

Account operations controller: Controllers\AccountController.cs

```
public class AccountController {
    public void Add(Account item) {
        var dc = new SymBankDataContext();
        dc.AccountAdd(item.Code, (int)item.Type, item.Name,
            item.ZipCode, MyApp.UserName, DateTime.UtcNow,
            item.Balance, Binary(item.Picture));
    }
    public Account GetAccount(int code) {
        var dc = new SymBankDataContext();
        return dc.Accounts.Single(a => a.Code == code);
    }
    public List<Account> GetAccountList() {
        var dc = new SymBankDataContext();
        return dc.Accounts.ToList();
    }
    public List<Account> GetAccountsForName(string name) {
        name = name.ToLower();
        var dc = new SymBankDataContext();
        var query = from account in dc.Accounts
            where account.Name.ToLower().Contains(name)
            orderby account.Name select account;
        return query.ToList();
    }
    public int Debit(int source, decimal amount) {
        int? transactionCode = null; var dc = new SymBankDataContext();
        dc.AccountDebit(source, amount, MyApp.UserName,
            DateTime.Now, ref transactionCode);
        return (int)transactionCode;
    }
    public int Credit(int source, decimal amount) {
        int? transactionCode = null; var dc = new SymBankDataContext();
        dc.AccountCredit(source, amount, MyApp.UserName,
            DateTime.Now, ref transactionCode);
        return (int)transactionCode;
    }
    public int Transfer(int source, int target, decimal amount) {
        int? transactionCode = null; var dc = new SymBankDataContext();
        dc.AccountTransfer(source, target, amount, MyApp.UserName,
            DateTime.Now, ref transactionCode);
        return (int)transactionCode;
    }
}
```

2.3 ObservableCollection

The previous controller returns multiple accounts as a **List<Account>** collection. This is perfectly fine if the collection is only for display purposes and you are going to add, replace or remove items from it while it is binded to a UI element. A list collection will not fire any notifications when it is changed as it does not implement the interface the UI is listening to which is **INotifyCollectionChanged<T>**. If you want to refresh the UI when the collection is changed use **ObservableCollection<T>** instead. Extend a custom entity collection class from this class as shown below.

Custom entity collection class: Models\AccountList.cs

```
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace SymBank.Models {
    public class AccountList : ObservableCollection<Account> {
        public AccountList() { }
        public AccountList(IEnumerable<Account> items) {
            foreach (var item in items) Add(item);
        }
    }
}
```

You can now modify the controller to return an instance of your custom collection that does implement **INotifyPropertyCollectionChanged<T>** interface. Do not replace existing methods as they can be used if a non-observable collection is required. Add in the following new methods to return observable collections.

Using custom entity collection: Controllers\AccountController.cs

```
public AccountList GetObservableAccountList() {
    var dc = new SymBankDataContext();
    return new AccountList(dc.Accounts);
}

public AccountList GetObservableAccountsForName(string name) {
    name = name.ToLower();
    var dc = new SymBankDataContext();
    var query = from account in dc.Accounts
            where account.Name.ToLower().Contains(name)
            orderby account.Name
            select account;
    return new AccountList(query);
}
```

3 MVVM Pattern

3.1 Views

To demonstrate view-models add views into the **Views** folder; **AddAccountView** and **SearchAccountsView**. First add user controls for each view and then change it into a view by changing the base class in the XAML document and also code-behind class to **BaseView**. Attach an event handler to the **Loaded** event. Update event handlers in the **Shell** window to open up the two views.

Basic XAML content for AddAccountView: Views\AddAccountView.xaml

```
<v:BaseView x:Class="SymBank.Views.AddAccountView"
    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:v="clr-namespace:SymBank.Views"
    mc:Ignorable="d" d:DesignHeight="400" d:DesignWidth="500"
    Header="New Account" Region="MainRegion"
    Loaded="BaseView_Loaded">
</v:BaseView>
```

Basic XAML content for SearchAccountsView: Views\SearchAccountsView.xaml

```
<v:BaseView x:Class="SymBank.Views.SearchAccountsView"
    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:v="clr-namespace:SymBank.Views"
    mc:Ignorable="d" d:DesignHeight="400" d:DesignWidth="500"
    Header="Account Search" Region="SideRegion"
    Loaded="BaseView_Loaded">
</v:BaseView>
```

Opening views directly from event handlers: Shell.xaml.cs

```
private void mnuAddNewAccount_Click(object sender, RoutedEventArgs e) {
    new AddAccountView().Show();
}

private void mnuSearchForAccounts_Click(object sender, RoutedEventArgs e) {
    new SearchAccountsView().Show();
}
```

The views uses three custom styles; **LabelStyle1**, **InputStyle1** and **ButtonStyle1**. Add the following styles to application resources.

Custom style resources to be used in views: Themes\Default.xaml

```
<Style x:Key="LabelStyle1" TargetType="Control">
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="12" />
    <Setter Property="Margin" Value="0,8,0,0" />
</Style>
<Style x:Key="InputStyle1" TargetType="Control">
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="12" />
    <Setter Property="Margin" Value="4,0,0,0" />
</Style>
<Style x:Key="ButtonStyle1" TargetType="Button">
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="12" />
    <Setter Property="Margin" Value="4" />
    <Setter Property="Width" Value="96" />
    <Setter Property="Height" Value="24" />
</Style>
```

The following is the basic structure of AddAccountView. A **ScrollViewer** decorator will be used to provide scrolling in case the size of the region cannot show the entire view consisting of a **Grid** with two rows and two columns. If you use the visual designer to implement the view, make sure that you do not use **Auto** for the Grid first until there is something in the cell or otherwise the cell will be collapsed so you will not be able to visually drop anything into the cell. Also open the **Document Outline** window from **View** menu to make it easier for you to select overlapping or nested panels. It is also annoying that additional unwanted properties are added to each UI element you drop into the designer window. Right click on the element and there should be an option to **Reset** the properties.

Basic structure of the view with 2 rows and 2 columns: AddAccountView.xaml

```
<ScrollViewer HorizontalScrollBarVisibility="Auto"
    VerticalScrollBarVisibility="Auto">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
    </Grid>
</ScrollViewer>
```

StackPanel in the first column

```
<StackPanel Grid.Column="0" Margin="8">
    <Label Content="_Code" Style="{StaticResource LabelStyle1}"
        Target="{Binding ElementName=txtCode}"/>
    <TextBox x:Name="txtCode" Style="{StaticResource InputStyle1}"
        Width="100" MaxLength="8" />
    <Label Content="_Name" Style="{StaticResource LabelStyle1}"
        Target="{Binding ElementName=txtName}"/>
    <TextBox x:Name="txtName" Style="{StaticResource InputStyle1}"
        Width="200" MaxLength="30" />
    <Label Content="_Type" Style="{StaticResource LabelStyle1}"
        Target="{Binding ElementName=cbxType}"/>
    <ComboBox x:Name="cbxType" Style="{StaticResource InputStyle1}"
        Width="200" />
    <Label Content="_Zip Code" Style="{StaticResource LabelStyle1}"
        Target="{Binding ElementName=txtZipCode}"/>
    <TextBox x:Name="txtZipCode" Style="{StaticResource InputStyle1}"
        Width="100" MaxLength="5" />
    <Label Content="_Balance" Style="{StaticResource LabelStyle1}"
        Target="{Binding ElementName=txtBalance}"/>
    <TextBox x:Name="txtBalance" Style="{StaticResource InputStyle1}"
        Width="100" MaxLength="12" />
</StackPanel>
```

StackPanel in the second row

```
<StackPanel Grid.Row="1" Grid.ColumnSpan="2"
    HorizontalAlignment="Left" Orientation="Horizontal" Margin="8">
    <Button Style="{StaticResource ButtonStyle1}" Content="Add" />
    <Button Style="{StaticResource ButtonStyle1}" Content="Clear" />
    <Button Style="{StaticResource ButtonStyle1}" Content="Close" />
</StackPanel>
```

StackPanel in the second column

```
<StackPanel Grid.Column="1" Margin="8">
    <Border x:Name="bdrPicture"
        Background="LightGray"
        BorderBrush="Gray" BorderThickness="2"
        Padding="8" Margin="4">
        <Image x:Name="imgPicture" Height="120" Width="120" />
    </Border>
    <Button Style="{StaticResource ButtonStyle1}"
        Content="Load Picture" />
</StackPanel>
```

Initial content in code-behind class

```
public override void ResetFocus() { txtCode.Focus(); }
```

DataTemplate in SearchAccountsView: Views\SearchAccountsView.xaml

```
<v:BaseView.Resources>
    <DataTemplate x:Key="searchResultsTemplate">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Image Grid.Column="0"
                Source="{Binding Picture}"
                Width="80" Height="80" Margin="4" />
            <StackPanel Grid.Column="1" Margin="4">
                <Label Content="{Binding Code}"/>
                <Label Content="{Binding Name}"/>
            </StackPanel>
        </Grid>
    </DataTemplate>
</v:BaseView.Resources>
```

Contents of the view

```
<DockPanel Margin="4">
    <ToolBar DockPanel.Dock="Top">
        <TextBox x:Name="txtName"
            Style="{StaticResource InputStyle1}"
            MaxLength="8" MinWidth="160" />
        <Button x:Name="btnSearch">
            <Image Stretch="None"
                Source="{StaticResource SearchImage}" />
        </Button>
        <Button x:Name="btnClose">
            <Image Stretch="None"
                Source="{StaticResource CancelImage}" />
        </Button>
    </ToolBar>
    <ListBox x:Name="lsbResults" Margin="0,4,0,0"
        ItemTemplate="{StaticResource searchResultsTemplate}">
        <ListBox.ItemsPanel>
                <ItemsPanelTemplate>
                    <WrapPanel Orientation="Vertical" />
            </ItemsPanelTemplate>
        </ListBox.ItemsPanel>
    </ListBox>
</DockPanel>
```

Initial content in code-behind class

```
public override void ResetFocus() { txtName.Focus(); }
```

3.2 View Models

The role of the view model is to help provide content and operations to view through data-binding. Expose content and commands through properties. If the content can be changed during the view, ensure that **PropertyChanged** event is invoked so that the UI will be refreshed for controls binded to the property. Following is the view model for **AddAccountView**.

View-model for AddAccountView: ViewModels\AddAccountViewModel.cs

```
namespace SymBank.ViewModels {
    public class AddAccountViewModel : BaseViewModel {
        private Account _account;
        private List<AccountType> _accountTypes;
        private AccountController _accountController;

        public Account Account {
            get { return _account; }
            set { _account = value;
                NotifyPropertyChanged("Account"); }
        }

        public List<AccountType> AccountTypes {
            get { return _accountTypes; }
        }

        public void Add() {
            try {
                _accountController.Add(_account);
                Shell.Status = string.Format(
                    "Account {0} added.", _account.Code);
                CloseView();
            }
            catch (Exception ex) {
                Shell.Failure("Error adding account.\n" + ex.Message);
            }
        }

        public void Clear() {
            _account.Name = "New account";
            _account.Type = AccountType.Savings;
            _account.ZipCode = "11111";
            _account.Balance = 100m;
        }

        public void LoadPicture(string path) {
            try { _account.Picture = File.ReadAllBytes(path); }
            catch (Exception ex) {
                Shell.Failure("Error loading picture.\n" + ex.Message);
            }
        }
```

```
        public void LoadPicture() {
            var filename = Shell.GetOpenFileName("Load Picture",
                "Image Files|*.bmp;*.jpg;*.png", string.Empty);
            if (filename != null) LoadPicture(filename);
        }

        public RelayCommand AddCommand { get; private set; }
        public RelayCommand ClearCommand { get; private set; }
        public RelayCommand LoadPictureCommand { get; private set; }


        public AddAccountViewModel() {
            AddCommand = new RelayCommand(p => Add());
            ClearCommand = new RelayCommand(p => Clear());
            LoadPictureCommand = new RelayCommand(p => LoadPicture());
            _account = new Account(); Clear();
            _accountController = new AccountController();
            _accountTypes = new List<AccountType> {
                AccountType.Savings,
                AccountType.Checking,
                AccountType.Loan
            };
        }
    }
}
```

You can expose a data model or a view model for binding using the **Source** property on a **Binding**. However this will be too troublesome to set the binding source on each binding expression. The simplest way is to assign it to the **DataContext** property and it will become the default binding source for all nested elements. You can register the prefix for **SymBank.ViewModels** namespace and instantiate view-model as shown for **AddAccountView**.

Instantiating view model in DataContext

```
<v:BaseView x:Class="SymBank.Views.AddAccountView"
    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:v="clr-namespace:SymBank.Views"
    xmlns:vm="clr-namespace:SymBank.ViewModels"
    mc:Ignorable="d" d:DesignHeight="400" d:DesignWidth="500"
    Header="New Account" Region="MainRegion">
    <v:BaseView.DataContext>
        <vm:AddAccountViewModel />
    </v:BaseView.DataContext>
```

Can also expose as view resource

```
<vm:AddAccountViewModel x:Key="vm" />
```

3.3 Data-Binding

You can use a **Binding** markup extension to bind properties of elements in the view with any properties in the view model. Use the **Source** binding property to assign an object that you wish to bind to. If you are not binding to the entire object, then use **Path** binding property to specify the property path that leads to the actual property that you wish to bind to. Use **Mode** binding property to specify **TwoWay** binding to ensure any changes to the UI element property will update the binding source as well. A **OneWay** binding mode would only display data from the binding source but does not update it.

Binding Path, Mode and Source

```
<TextBox x:Name="txtCode" Style="{StaticResource InputStyle1}"
    Text="{Binding Path=Account.Code, Mode=TwoWay, Source={StaticResource vm}}"
    Width="100" MaxLength="8" />
```

The problem with using binding source is that you have to set it for each UI element. This is alright if each element has a different binding source but will be a bit tedious if all elements are binding to the same source. However if **Source** is not set, it will bind to the **DataContext** property of the element instead. The difference is **DataContext** value can be passed down from ancestor elements. So if you assign your view-model to the **DataContext** of the view, all elements will get it unless you assign a different one for that element or its parent. As long as you set **DataContext** you can remove the binding source. You can shorten a binding further by removing the binding mode. The default binding mode, which is **Default** is *OneWay* for all display controls and *TwoWay* for input controls. Since we are using a **TextBox**, which is an input control, it will be *TwoWay* by default. You can also remove the **Path** word if the first binding property is actually the property path.

Simplified binding expression

```
<TextBox x:Name="txtCode"
    Style="{StaticResource InputStyle1}"
    Width="100" MaxLength="8"
    Text="{Account.Code}" />
```

You can now bind the rest of the controls. Note that you have to perform two bindings on the **ComboBox**. You need to bind **ItemsSource** to specify the list of items that you wish to display and bind **SelectedItem** to retrieve and store the item selected. If the property type is different you can always use a converter or add a property that will return the value as a different property type in your view model.

Binding an ItemsControl and using custom converters

```
<ComboBox x:Name="cbxType"
    Style="{StaticResource InputStyle1}"
    ItemsSource="{Binding AccountTypes}"
    SelectedItem="{Binding Account.Type}"
    Width="200" />
```

To allow the user to activate operations from view, you can bind commands in your view-model to **Command** property of the buttons in the view. When the user click on the button, the command will be executed. The button will be automatically disabled if the command cannot be executed.

Binding to commands in view-model

```
<Image x:Name="imgPicture"
        Source="{Binding Path=Account.Picture}"
        Height="120" Width="120" />
        <Button Style="{StaticResource ButtonStyle1}"
            Command="{Binding LoadPictureCommand}"
            Content="Load Picture" />
                    :

<Button Style="{StaticResource ButtonStyle1}" Content="Add"
    Command="{Binding AddCommand}" />
<Button Style="{StaticResource ButtonStyle1}" Content="Clear"
    Command="{Binding ClearCommand}" />
<Button Style="{StaticResource ButtonStyle1}" Content="Close"
    Command="{Binding CloseViewCommand}" />
```

You have now completed the view and can start using it to add one or more accounts. Use the same technique for **SearchAccountsView**. This view is much simpler as you just need to store a single text string and provide a read-only collection of **Account** objects as the result for the search. There is only one command for searching as the command for closing the view is already provided by the base class.

View-model for SearchAccountsView: SearchAccountsViewModel.cs

```
namespace SymBank.ViewModels {
    public class SearchAccountsViewModel : BaseViewModel {
        private AccountController _accountController;
        private AccountList _results;
        private string _name;

        public string Name {
            get { return _name; }
            set {
                _name = value;
                NotifyPropertyChanged("Name");
            }
        }

        public AccountList Results {
            get { return _results; }
            private set {
                _results = value;
                NotifyPropertyChanged("Results");
            }
        }
```

```
        public void Search() {
            try {
                Results = _accountController.
                    GetObservableAccountsForName(_name);
                Shell.Status = string.Format(
                    "Search completed. {0} items found.",
                    _results.Count);
            }
            catch (Exception ex) {
                Shell.Failure("Error searching for accounts.\n" + ex.Message);
            }
        }

        public RelayCommand SearchCommand { get; private set; }

        public SearchAccountsViewModel() {
            SearchCommand = new RelayCommand(p => Search());
            _accountController = new AccountController();
            _name = string.Empty;
        }
    }
}
```

Instancing view-model as DataContext: SearchAccountsView.xaml

```
<v:BaseView.DataContext>
    <vm:SearchAccountsViewModel />
</v:BaseView.DataContext>
```

Binding to the view model

```
<TextBox x:Name="txtName" Text="{Binding Name}" />
<Button Command="{Binding SearchCommand}">...</Button>
<Button Command="{Binding CloseViewCommand}"></Button>
                    :
<ListBox x:Name="lsbResults" Margin="0,4,0,0"
            ItemsSource="{Binding Results}">
```

3.4 Supporting Validation

The data-model that we generated from the database does not generate code to help validate properties of each class. This is not a big problem as stored procedures that we use to update the database can validate data before performing their operation. However it will be faster and a better user experience to validate and report errors from UI. Since all the existing data-model properties does not validate do not bind to those properties directly. Instead expose properties in the view-model to retrieve and store data into the data-model but with added validation.

Properties that perform validation: AddAccountViewModel.cs

```
public string AccountName {
    get {   return _account.Name; }
    set {   if (string.IsNullOrEmpty(value))
                throw new Exception("Account name cannot be blank");
            _account.Name = value;
            NotifyPropertyChanged("AccountName");
    }
}

public decimal OpeningBalance {
    get {   return _account.Balance; }
    set {   if (value < 100) throw new Exception(
                "Minimum opening balance must be 100");
            _account.Balance = value;
            NotifyPropertyChanged("OpeningBalance");
    }
}

private Regex _zipcodePattern = new Regex(@"^\d{5}$");

public string AccountZipCode {
    get {   return _account.ZipCode; }
    set {   if (!_zipcodePattern.IsMatch(value))
                throw new Exception("Invalid zipcode.");
            _account.ZipCode = value;
            NotifyPropertyChanged("AccountZipCode");
    }
}
```

Always update through validated properties

```
public void Clear() {
    AccountName = "New account";
    _account.Type = AccountType.Savings;
    AccountZipCode = "11111";
    OpeningBalance = 100m;
}
```

We are using exceptions to indicate validation failures. However by default WPF will ignore exceptions that occur on bindings. You will need to enable exceptions to be used as validation errors using **ValidatesOnExceptions** binding property in your binding expression as shown below.

Binding to validated properties: AddAccountView.xaml

```
<TextBox x:Name="txtName"
    Style="{StaticResource InputStyle1}" Width="200" MaxLength="30"
    Text="{Binding AccountName, ValidatesOnExceptions=True}" />
```

If you test now, you can see that when you try to move away from a **TextBox** that has validation errors, a red border appears on it. Unlike other UI elements validation is only done by default on any **TextBox** in **LostFocus**. If you wish to validation on changes, set **UpdateSourceTrigger** binding property to **PropertyChanged**.

Validate imediately on text changed

```
<TextBox x:Name="txtName"
    Style="{StaticResource InputStyle1}" Width="200" MaxLength="30"
    Text="{Binding Path=AccountName, ValidatesOnExceptions=True,
            UpdateSourceTrigger=PropertyChanged}" />
```

Even though the validation is now done, WPF only highlights the error but does not stop user from still continuing the operation even though the errors are not corrected. We have to detect errors in our view or view-model manually by providing a validation error handler. You can use it to track how many errors are not corrected using a simple integer variable that can be used for disabling commands.

Disabling commands on errors: AddAccountsViewModel.cs

```
private int _addErrors;

public AddAccountViewModel() {
    AddCommand = new RelayCommand(p => Add(), p => _addErrors == 0);
                    :
```

Handler to detect errors and refresh commands

```
public void OnAddAccountValidationError(
    object sender, ValidationErrorEventArgs e) {
    switch (e.Action) {
            case ValidationErrorEventAction.Added: ++_addErrors; break;
            case ValidationErrorEventAction.Removed: --_addErrors; break;
    }
    AddCommand.NotifyCanExecuteChanged();
}
```

When a validation error occurs, a notification event can be routed upwards from the element that failed validation. At any level, use **AddErrorHandler** from **Validation** class to assign the error event handler. In the following we will detect validation error events at **BaseView** level and attach the handler from the view-model. If you have multiple forms within a single view, you will need to attach multiple handlers to each section of the view where your form is.

Attaching error handler

```
public AddAccountView() {
    InitializeComponent();
    var vm = (AddAccountViewModel)DataContext;
    Validation.AddErrorHandler(this, vm.OnAddAccountValidationError);
}
```

Even though we can now detect validation errors, by default no validation error event is actually sent when validation fails. You need to enable **NotifyOnValidationError** binding property to send error notification events if validation fails.

Enabling validation error notifications: AddAccountView.xaml

```
<TextBox x:Name="txtName"
        Style="{StaticResource InputStyle1}" Width="200" MaxLength="30"
        Text="{Binding Path=AccountName, ValidatesOnExceptions=True,
            UpdateSourceTrigger=PropertyChanged,
            NotifyOnValidationError=True}" />
```

You have now implemented validation in your view but there is still a problem. When validation fails, only a red border is drawn around the control but it does not show the actual error to the user. The red border is generated by a default **ControlTemplate**. You can customize how to display validation errors for each UI element by assigning a custom template to the **Validation.ErrorTemplate** attached property.

Custom error template: Themes\Default.xaml

```
<ControlTemplate x:Key="MyErrorTemplate" TargetType="Control">
    <Grid>
        <AdornedElementPlaceholder Name="element" />
        <Border BorderBrush="Red" BorderThickness="1" />
        <Image HorizontalAlignment="Right" VerticalAlignment="Top"
            Width="16" Height="16" Margin="0,-8,-8,0"
            Source="{StaticResource ErrorImage}"
        ToolTip="{Binding ElementName=element,
            Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"/>
    </Grid>
</ControlTemplate>
```

You can then assign custom error template by using the **Validation.ErrorTemplate** attached property. If you find it tedious to attach it to every UI element that you wish to validate, you can use the Style to assign it instead.

Assigning attach properties in style: Themes\SymBank.xaml

```
<Style x:Key="InputStyle1" TargetType="Control">
    <Setter Property="Validation.ErrorTemplate"
            Value="{StaticResource MyErrorTemplate}" />
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="12" />
    <Setter Property="Margin" Value="4,0,0,0" />
</Style>
```