Model-View-ViewModel

Estimated time for completion: **30 minutes**

Goals:

- Apply the Model-View-ViewModel (MVVM) pattern to your application

Overview:

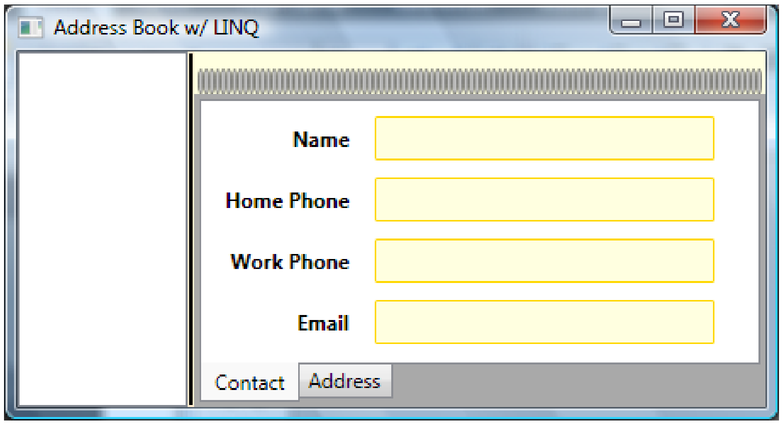
Throughout this lab you will be working on an address book implementation. You will start with a simple UI and then add in the data binding using the MVVM designed pattern to support the display.

Part 1 – Creating an address book client

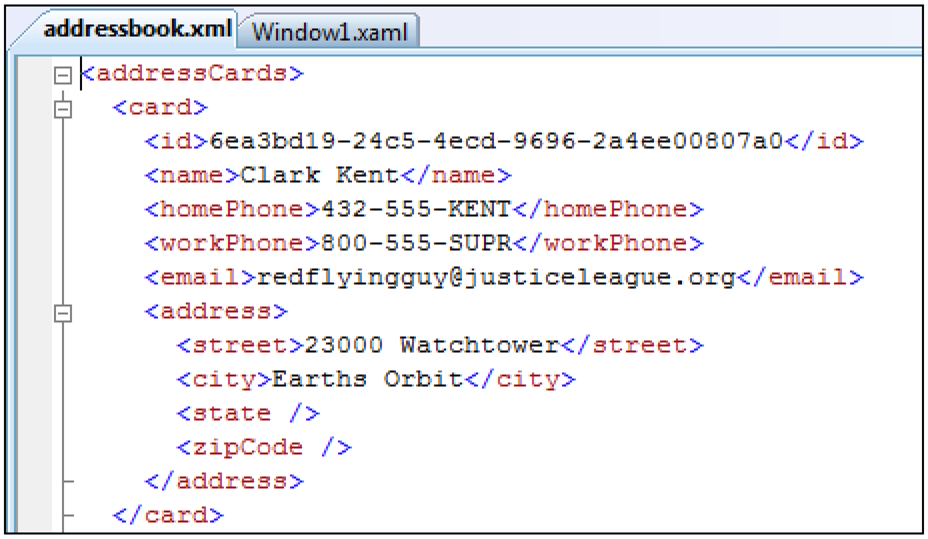
In this part, you'll modify a WPF-based address book application to read an XML data file with XLinq and data bind to a ViewModel. The model itself has already been created

Steps:

1. Open the **AddressBook** starter project located in the **before** folder associated with the lab.
2. Run the application – it should look like:



3. Currently, there is no data source for this application – however one is supplied in the form of the addressbook.xml file in the project. Open the file and examine the XML structure:



```xml
<addressCards>
  <card>
    <id>6ea3bd19-24c5-4ecd-9696-2a4ee00807a0</id>
    <name>Clark Kent</name>
    <homePhone>432-555-KENT</homePhone>
    <workPhone>800-555-SUPR</workPhone>
    <email>redflyingguy@justiceleague.org</email>
    <address>
      <street>23000 Watchtower</street>
      <city>Earths Orbit</city>
      <state />
      <zipCode />
    </address>
  </card>
```

4. Each card has an **id, name, homePhone, workPhone, email and address**.
5. There is already a data model created for you – open the Model folder in the project and examine the files contained in there.
    a. **ContactCardManager** is the data loader/saver. This is essentially the simple Data Access Layer (DAL) which has been created to load and persist the file. In a production application it might be using a database or web service. It is using XLinq (Linq for XML) to read/write the file.
    b. **ContactCard** is the model object for a single contact. It exposes properties for each field present in our data.
    c. **ContactAddress** is the model object for an address.
6. There are some MVVM helpers already preloaded into this project as well. These were taken (with permission) from the **JulMar MVVM library** (http://mvvmhelpers.codeplex.com).
    a. **DelegateCommand** is an implementation of **ICommand** that binds to a delegate Action.
    b. **ViewModel** is a base ViewModel class which implements INotifyPropertyChanged.
7. Our first step will be to create ViewModel objects for the two data elements we want to display. Remember the point of the ViewModel is to have a WPF-binding friendly class. In our case, the model objects are not binding-friendly because they don't implement INotifyPropertyChanged. Assume for this exercise that you do not have the freedom to modify these definitions – either because you do not own them, or because they are shared with another group and unchangeable. To solve this we will create ViewModel wrappers around the data objects which are fully bindable by WPF. This step could be omitted if the data objects were not changed (read-only) and exposed everything we wanted to view through properties.
8. Start by creating a **ViewModels** folder in the project – it's useful to use project folders to group the various sections of our application and we'll use this one to hold all the

ViewModel classes. Right click on the project and select "Add | New Folder" and name it **ViewModels**.

9. In the new **ViewModels** folder, add a source file called **ContactViewModel**. We will use this as our ViewModel wrapper for the **ContactCard** class.

    a. Derive the class from **JulMar.Windows.Mvvm.ViewModel**. This is where we will get our INPC implementation. It has a virtual method called "OnNotifyPropertyChanged" which you can use to raise the PropertyChanged event.

    b. Add a public constructor that takes a **ContactCard** object. Store the instance off into a field in the view model. The lab implementation names the field "_card".

    c. Create a public property for the **Name** property. It should get and set the **_card.Name** property and raise **INotifyPropertyChanged.PropertyChanged**. This is done by calling the "OnNotifyPropertyChanged" method which is defined in the base ViewModel class. When you are finished it should look something like:

```
public class ContactViewModel : ViewModel
{
    private ContactCard _card;

    public ContactViewModel(ContactCard card)
    {
        _card = card;
    }

    public string Name
    {
        get { return _card.Name; }
        set { _card.Name = value; OnPropertyChanged("Name"); }
    }
}
```

10. Do the same steps for the HomePhone, WorkPhone and Email properties.

11. Next, let's add support for the Address. Create a new view model to wrap the **ContactAddress** object. Name it **AddressViewModel**.

12. It should follow the same basic design you've already done for the contact:

    a. Derive it from **ViewModel**.

    b. Constructor which takes a **ContactAddress** as input and stores it off in a field.

    c. Property wrappers for the fields we intend to change or display (Street, City, State, ZipCode).

13. Switch back to your **ContactViewModel**. In the constructor, check the input **ContactCard** object's Address property – if it is null, create a new address and set it onto the ContactCard object:

```
public ContactViewModel(ContactCard card)
{
   _card = card;
   if (_card.Address == null)
      _card.Address = new ContactAddress();
}
```

14. This simply ensures we have a model to work with – just in case the data didn't create an address for the contact.

15. Next, create a new field in the class which will hold this **ContactAddress** in an **AddressViewModel**. Name it "_cardAddress". Assign this field in the constructor by creating a new wrapping view model around the **ContactCard.Address**:

```
AddressViewModel _cardAddress;

public ContactViewModel(ContactCard card)
{
   _card = card;
   if (_card.Address == null)
      _card.Address = new ContactAddress();

   _cardAddress = new AddressViewModel(card.Address);
}
```

16. Add a wrapping property around this field called **Address**. It only needs a getter as we will not allow the entire object to be replaced.

17. Now we've got our basic ViewModel for the data all setup. Our next step is to create a ViewModel which will drive the primary UI. This view model will be responsible for holding the list of contact view models as well as handling UI events.

18. Create a new view model class called "MainViewModel". Derive it from ViewModel like you have before.

19. Add a public property to hold the contacts – since we intend to add/remove contacts, use an **ObservableCollection<T>** to hold the data. The property should have a public getter and a private setter:

```
public class MainViewModel : JulMar.Windows.Mvvm.ViewModel
{
 public ObservableCollection<ContactViewModel> Contacts { get; private set; }
}
```

20. Next, create a public, default constructor. In the constructor, create the above collection and use the **ContactCardManager.Load** method to fill it. Here is the basic code if you need some help:

```
public MainViewModel()
{
   Contacts = new ObservableCollection<ContactViewModel>();
   foreach (var card in ContactCardManager.Load(@"addressbook.xml"))
      Contacts.Add(new ContactViewModel(card));
}
```

21. Open window1.xaml.cs and assign the **DataContext** for the window to a MainViewModel instance.
22. Databind the `ListBox` to the **Contacts** collection. Run the application – a visualization for the contacts has been provided by your designer (lucky you!) It should use that visualization in the displayed list.
23. Next, we want to display the selected item details in the right-side pane. We could use the **ICollectionView** support to track selection (try it, it works). An alternative approach when using MVVM is to track selected in the view model.
24. Add a **SelectedCard** property in the **MainViewModel**. It should be of type **ContactViewModel**. Make sure to raise the property change notification when the setter is used.

```
        private ContactViewModel _selectedCard;
        public ContactViewModel SelectedCard
        {
            get { return _selectedCard; }
            set { _selectedCard = value; OnPropertyChanged("SelectedCard"); }
        }
```

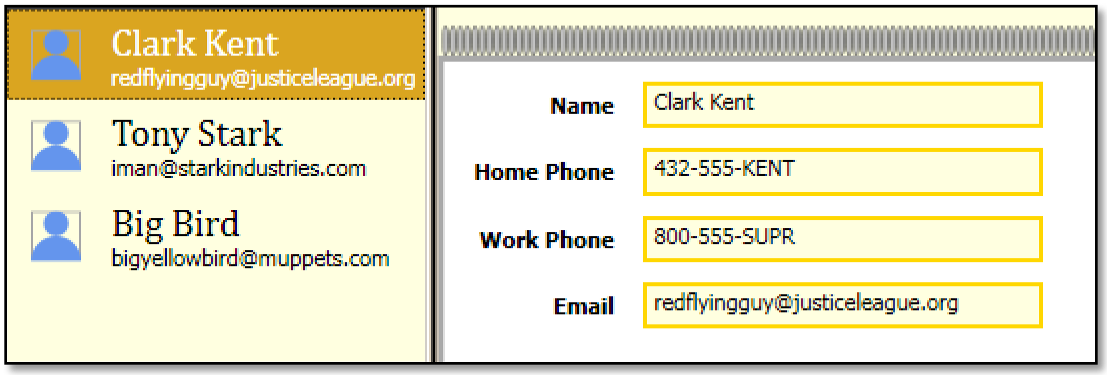
25. Now, bind the ListBox **SelectedItem** property to your new view model property. Finally, set the data context for the TabControl which holds all the details to the **SelectedCard** property.

```
<ListBox MinWidth="100" ItemsSource="{Binding Contacts}"
         Background="LightYellow"
         SelectedItem="{Binding SelectedCard}"
         ItemTemplate="{StaticResource contactCardTemplate}" />

<GridSplitter Grid.Column="1" ResizeBehavior="PreviousAndNext"
         Width="4" Background="Black"
         BorderBrush="BlanchedAlmond" BorderThickness="1" />

<Grid Grid.Column="2">
  ...
  <TabControl Grid.Row="1" Background="White" TabStripPlacement="Bottom"
              Margin="3" DataContext="{Binding SelectedCard}">
```

26. Set the selected item to the first loaded contact in your MainViewModel constructor:

```
 public MainViewModel()
 {
     ...
     if (Contacts.Count > 0)
         SelectedCard = Contacts[0];
}
```

27. Next, we want to databind each of the TextBox elements to the current selection. Go through the TextBox elements and add the appropriate binding statements to link them to the **ContactViewModel**.
28. Run the application – it should now be displaying the property values:



Clark Kent
redflyingguy@justiceleague.org
Tony Stark
iman@starkindustries.com
Big Bird
bigyellowbird@muppets.com

Name: Clark Kent
Home Phone: 432-555-KENT
Work Phone: 800-555-SUPR
Email: redflyingguy@justiceleague.org

29. Make sure to wire up the Address fields as well – you can either set the DataContext on the root element, or fully specify the property path (**Address.Street**, etc.).
30. The next job is to wire up the Add/Remove functionality. We want to put the logic for this into the MainViewModel – by setting commands onto the buttons which activate those features.
31. In the MainViewModel, create two **ICommand** properties, one for Add and one for Remove. In the constructor, use the **Julmar.Windows.Mvvm.DelegatingCommand** wrapper to point these commands to methods in your view model. The remove command requires a selected contact so add a second delegate to that command to return the proper CanExecute state:

```
        public ICommand AddCommand { get; private set; }
        public ICommand RemoveCommand { get; private set; }

        public MainViewModel()
        {
            ...

            AddCommand = new DelegatingCommand(OnAdd);
            RemoveCommand = new DelegatingCommand(OnRemove, OnCanRemove);
        }

        void OnAdd()
        {
        }
        void OnRemove()
        {
        }

        bool OnCanRemove()
        {
        }
```

32. In the Add method, you want to create a new contact and add it to your Contacts collection. Also, for convenience, go ahead and make it the selected contact.

```
        void OnAdd()
        {
            ContactCard newCard = new ContactCard();
            newCard.Id = Guid.NewGuid();

            ContactViewModel vm = new ContactViewModel(newCard);
            Contacts.Add(vm);
            SelectedCard = vm;
        }
```

33. In the Remove method, you want to test the selected item and remove it from the Contacts collection. For convenience, select another card once you are done:

```
        void OnRemove()
        {
            if (SelectedCard != null)
            {
                int index = Contacts.IndexOf(SelectedCard);
                Contacts.Remove(SelectedCard);
                if (Contacts.Count > index)
                    SelectedCard = Contacts[index];
                else if (Contacts.Count > 0)
                    SelectedCard = Contacts[0];
                else
                    SelectedCard = null;
            }
        }
```

34. Finally, implement the OnCanRemove to test for a selected item:

```
        bool OnCanRemove()
        {
            return SelectedCard != null;
        }
```

35. Wire up these commands to the add/remove buttons using data binding. Since they are now properties on the view model you use the data binding syntax to get to them.
36. The last step is to save the file when you are finished. In this case, we need to hook the closing event of the window and write out our file. We could use a solution such as the LifetimeEvents class in the Julmar library, but let's keep it simple – just handle the event in the XAML code behind file and dispose the **MainViewModel**.

```
public partial class Window1
{
    ...
    protected override void OnClosing(CancelEventArgs e)
    {
        if (MessageBox.Show("Would you like to save the changes to the address
book?", "Save Changes", MessageBoxButton.YesNo) == MessageBoxResult.Yes)
            ((IDisposable)DataContext).Dispose();
        base.OnClosing(e);
    }
}
```

37. Then add a **Dispose()** override to the view model which saves the file. You will need to access the underlying model object in the view to save it – in the sample implementation Since the underlying data access layer works with the internal **ContactCard**, you will need to add a property into the ViewModel to retrieve the underlying model object (the _card property) as well.

```
public class MainViewModel
{
    ...
    protected override void Dispose(bool isDisposing)
    {
        if (isDisposing)
        {
            ContactCardManager.Save(@"addressbook.xml",
            Contacts.Select(cvm => cvm.ContactCard));
        }
    }
}
```

38. Run the application and verify that it all works as expected and changes made should be persisted and reloaded when the application is restarted.

Solution

A solution to the entire lab is available at: after\WpfAddressBook.sln