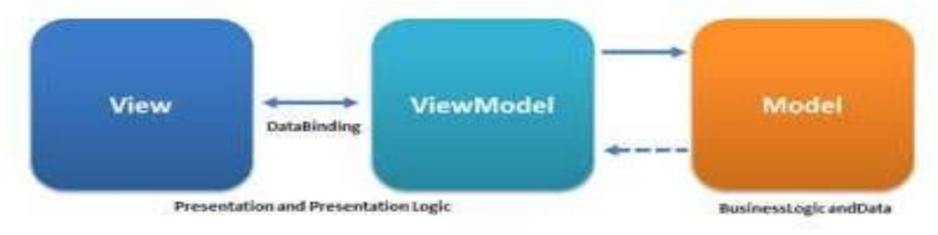
Model-View-ViewModel (MVVM) .

Why should you, as a developer, even care about the Model-View-ViewModel pattern? There are a number of benefits. Before you go on, ask yourself:

- Do you need to share a project with a designer, and have the flexibility for design work and development work to happen near-simultaneously?
- Do you require thorough unit testing for your solutions?
- Is it important for you to have reusable components, both within and across projects in your organization?
- Would you like more flexibility to change your user interface without having to refactor other logic in the code base?

Every good developer wants and tries to create the most sophisticated applications to delight their users. Most of the times, developers achieve this on the first release of the application. However, with new feature addition, fixing the bug without putting a lot of consideration into the structure of the application code becomes difficult due to code complexity. For this, there is a need for



- **Models** hold application data. They're usually structs or simple classes.
- **Views** display visual elements and controls on the screen. **View models** transform model information into values that can be displayed on a view. They're usually classes, so they can be passed around as references.

In other words, we create model and write all code, commands, events (Business logic) in ViewModel once and use that Model and ViewModel in all the applications. We just need to make separate and different Views for each and every applications.

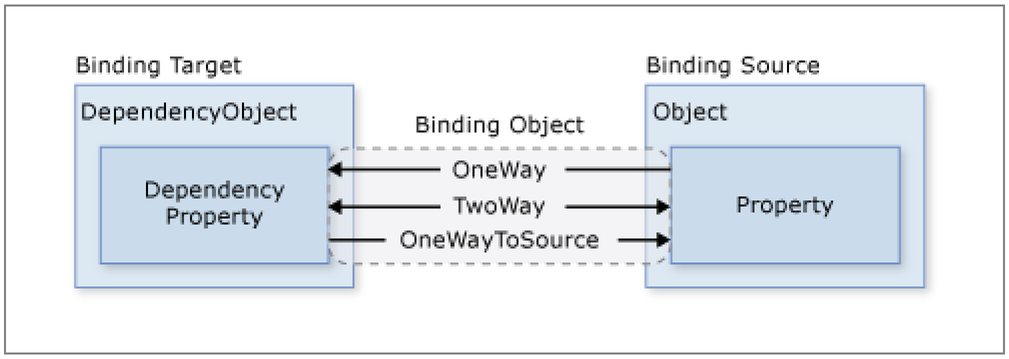
So as we see the most important thing should be in my application is **binding** concept:

First of all we should talk about **Binding**:

provides a simple and consistent way for apps to present and interact with data.

Here are some of the common binding properties:

- **ElementName**: The data source is another element. The source element must have a name specified, usually with the x:Name attribute.
- **Source**: Provide a reference to a valid object source. A binding configured with the Source property takes precedence over any applied data context.
- **RelativeSource**: The source is a property on a related object.
- **Path**: Provide navigation instructions for finding the data source property. The simplest path is merely the name of a property:



Data Binding / Object Binding

Data Binding can be achieved by several ways.

1-Element in XAML with another element in XAML.

2-Propperty in C# with element in XAML.

1- Element in XAML with another element in XAML.

```
<TextBox x:Name="txtName" />
<TextBlock Text="{Binding ElementName=txtName, Path=Text.Length}" />
```

whenever you enter something on the TextBox during runtime, the TextBlock will show the length of the string.

2- Propperty in C# with element in XAML.

How to: Make Data Available for Binding in XAML:

you can make the object available for binding in xaml, We have several ways:

1- Make it as resource:

```
<Window.Resources>
 <src:Person x:Key="myDataSource" PersonName="Joe"/>

  </Window.Resources>



<TextBox.Text>

        <Binding Source="{StaticResource myDataSource}" Path="PersonName"

  UpdateSourceTrigger="PropertyChanged"/>

      </TextBox.Text>
```

OR

```
<TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}"/>
```

2-By using **ObjectDataProvider** :

```
<ObjectDataProvider x:Key="myDataSource" ObjectType="{x:Type src:Person}">

     <ObjectDataProvider.ConstructorParameters>

       <system:String>Joe</system:String>

     </ObjectDataProvider.ConstructorParameters>

   </ObjectDataProvider>
```

```
<TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=Name}"/>
```

We can bind by different ways

```
<!-- Path to sub properties-->
<TextBlock text='{Binding Path=UserName.Length}' />

<!-- Path to attached property -->
<TextBlock text='{Binding Path=(Canvas.Top)}' />

<!-- Path to Indexer property-->
<TextBlock text='{Binding Path='{Brushes[2]}' />
```

Difference between DataContext and ItemsSource

1. DataContext expects an object type where ItemsSource expects IEnumerable type objects.

2. DataContext is a dependency property is exposed by FrameworkElement base class,where as ItemsSource is defined by the ItemsControl class. All the descendants of FrameworkElement can utilize the DataContext property and set an object to its value. But we can only set a type of IEnumerable(or instance of class that derives from).

3. DataContext does not generate template, it only used to hold common data for other controls to bind. In terms of ItemsSource property, it is mainly used to generate template regardless of you set it in XAML or in the code behind.

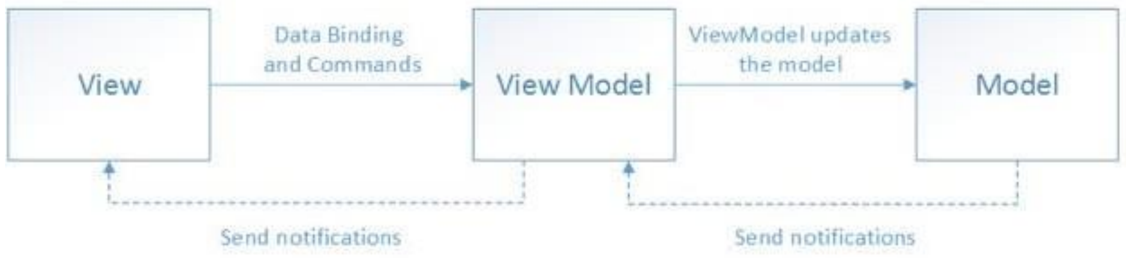
4. DataContext is mainly used to hold common data that other child want to share. Thus it can be inherited by other child elements without problem. But for ItemsSource, it not used to share data in the visual tree. It is only valid for the element that defined. There is still one thing to be noted is that the child element can override the DataContext of the perent DataContext no mater directly or indirectly.

DataTemplate

The following example shows how to create a DataTemplate inline. The DataTemplate specifies that each data item appears as three TextBlock elements within a StackPanel. In this example, the data object is a class called Task**. Note that each TextBlock element in this template is bound to a property of the** Task **class.**

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}">
   <ListBox.ItemTemplate>
    <DataTemplate>
       <StackPanel>
         <TextBlock Text="{Binding Path=TaskName}" />
         <TextBlock Text="{Binding Path=Description}"/>
         <TextBlock Text="{Binding Path=Priority}"/>
       </StackPanel>
     </DataTemplate>
   </ListBox.ItemTemplate>
 </ListBox>
```

There are some things that facilitate DataBinding



1-Commands

2-Notifications

3-Value converter

4- ObservableCollection

- Command:

To bind a command of a button you need to bind a property that is an implementation of an ICommand. An **ICommand** is composed by:

- event EventHandler CanExecuteChanged;

- bool CanExecute(object parameter);

- void Execute(object parameter);

CanExecuteChanged is invoked when changes occur that can change whether or not the command can be executed.

CanExecute will determine whether the command can be executed or not. If it returns false the button will be disabled on the interface.

Execute runs the command logic.

Four Main Concepts

- **The** *command* **is the action to be executed.**
- **The** *command source* **is the object which invokes the command.**
- **The** *command target* **is the object that the command is being executed on.**
- **The** *command binding* **is the object which maps the command logic to the command.**

```
class Updater : ICommand

{   public bool CanExecute(object parameter)

    { return true; }

    public event EventHandler CanExecuteChanged

    {

       add { CommandManager.RequerySuggested += value; }

        remove{CommandManager.RequerySuggested-=value;}

    }

    public void Execute(object parameter)

    { //Your Code }

}
```

expose **Updater** instance in VM.

In view:

```
<Button x:Name="btnUpdate"

                Command="{Binding Path= this property in VM}"
                CommandParameter="{ paramerter }"></Button>
```

- INotifyPropertyChanged

- **INotifyPropertyChanged:** Implement this .NET 2.0 interface and create the **PropertyChanged** event. In your property setters, raise the PropertyChanged event whenever the value changed.
- **To support OneWay or TwoWay binding to enable your binding target properties to automatically reflect the dynamic changes of the binding source (for example, to have the preview pane updated automatically when the user edits a form), your class needs to provide the proper property changed notifications.**

To implement INotifyPropertyChanged you need to declare the **PropertyChanged** event

and create the **OnPropertyChanged** method. Then for each property you want change notifications for, you call **OnPropertyChanged** whenever the property is updated.

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace SDKSample
{
  // This class implements INotifyPropertyChanged
  // to support one-way and two-way bindings
  // (such that the UI element updates when the source
  // has been changed dynamically)
  public class Person : INotifyPropertyChanged
  {
      private string name;
      // Declare the event
      public event PropertyChangedEventHandler PropertyChanged;

      public Person(){}
      public Person(string value)
      {
          this.name = value;
      }
      public string PersonName
      {
          get { return name; }
          set
          {
              name = value;
              // Call OnPropertyChanged whenever the property is updated
              OnPropertyChanged();
          }
      }
      // Create the OnPropertyChanged method to raise the event
      // The calling member's name will be used as the parameter.
      protected void OnPropertyChanged([CallerMemberName] string name = null)
      {
          PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
      }
  }
}
```

- Value converter

Value converters are very frequently used with data bindings.

value converter needs to implement the IValueConverter interface, or alternatively, the IMultiValueConverter interface (more about that one later). Both interfaces just requires you to implement two methods: Convert() and ConvertBack(). As the name implies, these methods will be used to convert the value to the destination format and then back again.

Let's implement a simple converter which takes a string as input and then returns a Boolean value, as well as the other way around.

```
public class YesNoToBooleanConverter : IValueConverter
        {
public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo
culture)
                {
                        switch(value.ToString().ToLower())
                        {
                                case "yes":
                                case "oui":
                                        return true;
                                case "no":
                                case "non":
                                        return false;
                        }
                        return false;
                }
public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
                {
                        if(value is bool)
                        {
                              if((bool)value == true)
                                        return "yes";
                               else
                                        return "no";
                        }

                        return "no";
                }
        }
}
```

ObservableCollection

An ObservableCollection is a dynamic collection of objects of a given type. Objects can be added, removed or be updated with an automatic notification of actions. When an object is added to or removed from an observable collection, the UI is automatically updated . This happens because of,

a CollectionChanged event handler to the ObservableCollecion's events.

The ObservableCollection class exists in the **System.Collections.ObjectModel** namespace.

Ex: One observableCollection object is created with a Person type and bound to the ListView.

In view:

```
<ListView ItemsSource =”{Binding persons}”>
```

In VM:

we can dynamically insert, update and remove items from the ListView:

```
persons.Add(new Person() { Name = txtName.Text, Address = txtAddress.Text });
```

NOW,

We already have develop a project using MVVM pattern, but it really gets difficult some time to manage it, mostly due to event handling between modules & somewhat dependency of views & modules on each other.

So we have some tools can facilitate thate:

popular MVVM frameworks:

- MVVM Light
- Prism
- Caliburn Micro

Getting Started With MVVM Light

 The main purpose of the toolkit is to accelerate the creation and development of MVVM applications in Windows Universal, WPF, Silverlight, Xamarin.iOS, Xamarin.Android and Xamarin.Forms.

he MVVM Light Toolkit helps you to **separate your View from your Model** which creates applications that are **cleaner and easier to maintain and extend**. It also creates **testable applications** and allows you to have a much thinner user interface layer (which is more difficult to test automatically).

This toolkit puts a special emphasis on the **designability** of the created application (i.e. the ability to open and**edit the user interface into Blend)**, including the creation of design-time data to enable the Blend users to "see something" when they work with data controls.

1-Adding MVVM Light to an existing project

If you want to add MVVM Light to an existing project, you should rather use the NuGet package manager to download and add MVVM Light!

We will get a ViewModel folder in this folder we will find two files (MainViewModel,ViewModelLocator)

2- IN viewmodelLocator file REPLACE

//using Microsoft.Practices.ServiceLocation; **TO** using CommonServiceLocator;

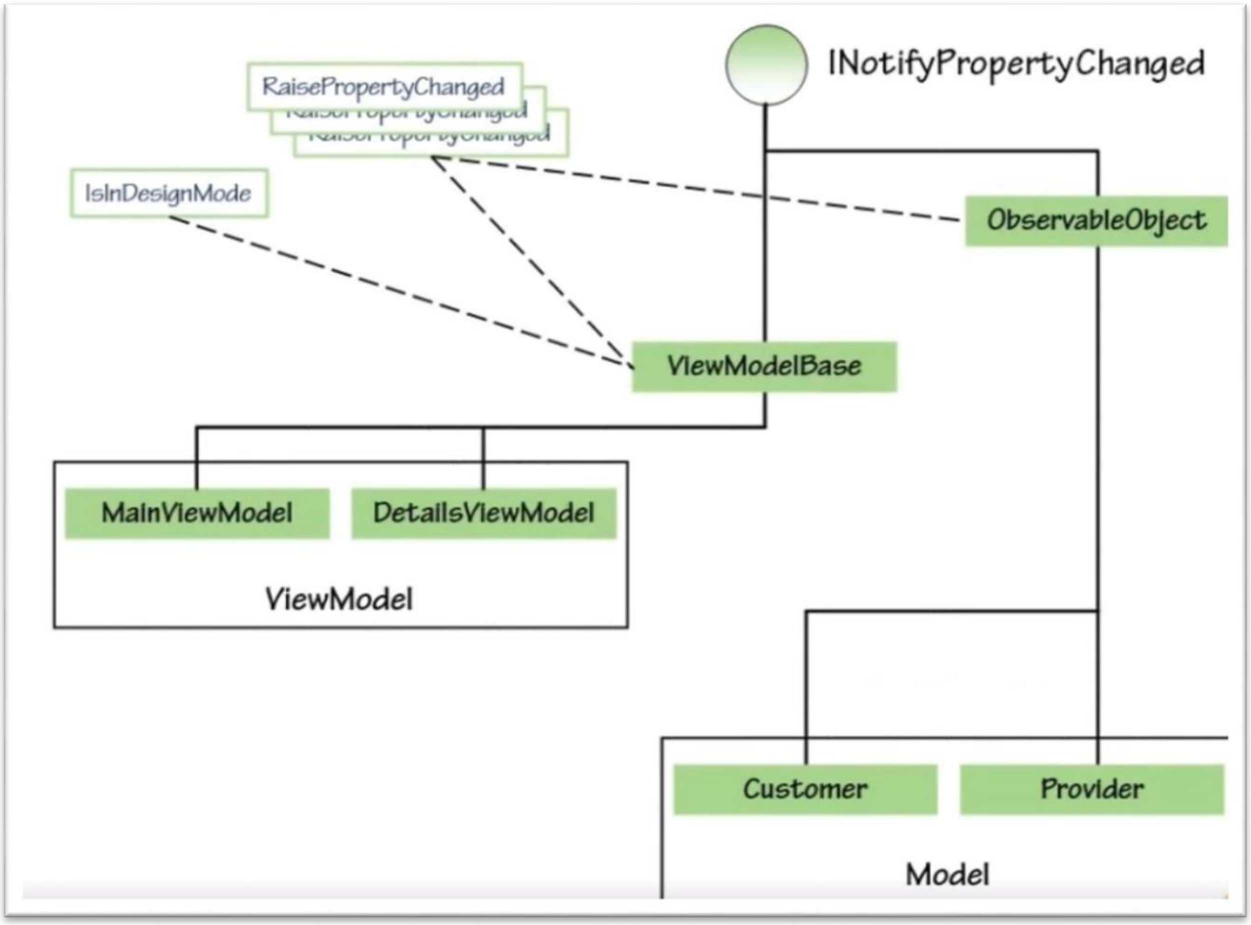
- The main benefit of using a library is that it provides you with most things that you will need to develop an MVVM application out of the box. This for example includes a base class that raises implements the **INotifyPropertyChanged** interface and raises change notifications and an implementation of the **ICommand** interface.

So, MVVM Light give us things that will help us to make MVVM easier

1-ViewModelBase

2- ObservableObject

3-Messanger



Raising the PropertyChanged Event

- Event raise (ObservableObject and ViewModelBase)

```
Ra1sePropertyChanged( "HyProperty" ) j
//   "C1ass1c" way .
// Typically used with a constant for the property's name.

Ra1sePropertyChanged(() => NyProperty) ,
// Supports Intellisense and automatic refactoring.
//   Very, very small performance 1mpact .

Set( "HyProperty",  ref  _oyProperty, va1ue);

Set:( ( ) => NyProperty, r‘ef _myProperty, value) j
// Set method takes care of checking if event must be raised.
// Returns true if event was raised.
```

Raising the PropertyChanged Event

- Event raise and broadcasting through Messenger (ViewModelBase)

```
Ra1sePropertyChanged(”NyProperty", oldValue, value, true);

RalsePropertyChanged( ()  =› NyProperty, o1dVa1ue, va1ue, true);

Set( "NyProperty",  ref  _myProperty, va1ue, I:rue) j

Set:(() =› NyPropert:y, ref  _myProperty, value, true),
```

- Sends a PropertyChangedMessage (see module about Messenger)

1-ViewModelBase

Every viewmodel in MVVM Light Toolkit must inherit from ViewModelBase class. The ViewModelBase abstract class is designed to serve as a base class for all ViewModel classes. It provides support for property change notifications. Instead of implementing the INotifyPropertyChanged interface in each individual ViewModel of your application, you can directly inherit the ViewModelBase class.

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using  System.Windows.Input;
using System.Windows;

namespace MvvmLightTest.ViewModel
{
    public class MainViewModel : ViewModelBase
    {
        int _exampleValue;

        public int ExampleValue
        {
            get
            {
                return _exampleValue;
            }
            set
            {
                if (_exampleValue == value)
                    return;
                _exampleValue = value;
                RaisePropertyChanged("ExampleValue");
            }
        }

    }
}
}
```

when explicitly specifying the changed property, there are a couple of overloads that you can use.

2- ObservableObject

Every model class in this MVVM Light Toolkit must inherit from ObservableObject. ObservableObject class is inherit from the INofityPropertyChanged interface. This interface provides PropertyChanged event handler that notifiy clients that a property value has changed. ObservableObject use that event in RaisePropertyChanged method to notify other classes. In addition, ObservableObject class also provides a Set method to set the property and raise the PropertyChanged event automatically. Set method takes property name, reference to the private variable and the new value. Below I have shown an example of Employee class.

```
public class Employee : ObservableObject
{
    private int id;
    private string name;
    private int age;
    private decimal salary;

    public int ID
    {
        get
        {
            return id;
        }
        set
        {
            Set<int>(() => this.ID, ref id, value);
        }
    }
```

```
public string Name
{
    get
    {
        return name;
    }
    set
    {
        Set<string>(() => this.Name, ref name, value);
    }
}

public int Age
{
    get
    {
        return age;
    }
    set
    {
        Set<int>(() => this.Age, ref age, value);
    }
}

public decimal Salary
{
    get
    {
        return salary;
    }
```

```
        set
        {
            Set<decimal>(() => this.Salary, ref salary, value);
        }
    }

    public static ObservableCollection<Employee> GetSampleEmployees()
    {
        ObservableCollection<Employee> employees = new
ObservableCollection<Employee>();
        for (int i = 0; i < 30; ++i)
        {
            employees.Add(new Employee
                {
                    ID = i + 1,
                    Name = "Name " + (i + 1).ToString(),
                    Age = 20 + i,
                    Salary = 20000 + (i * 10)
                });
        }

        return employees;
    }
```

four properties name ID, Name, Age, and Salary in above model class. All properties setter used the Set method of ObservableObject. Set method assign a new value to the property and then call the RaisePropertyChanged method. The use of RaisePropertyChanged method is must as we have to update the UI when any property changed.

=====================RelayCommand====================

RelayCommand's purpose is to implement the ICommand interface that Button controls needs and to just pass the calls onto some other function which generally sits right next to them in the ViewModel.

The constructor takes two arguments; the first is an Action which will be executed if ICommand.Execute is called (e.g. the user clicks on the button), the second one is

a Func<bool> which determines if the action can be executed (defaults to true, called canExecute in the following paragraph).

```
RelayCommand(which we need to excute,canexcute or not)
```

```
public MainViewModel()

{

  MyCommand = new RelayCommand(

    ExecuteMyCommand,

    () => _canExecuteMyCommand); }

private void ExecuteMyCommand()

{// Do something }
```

OR

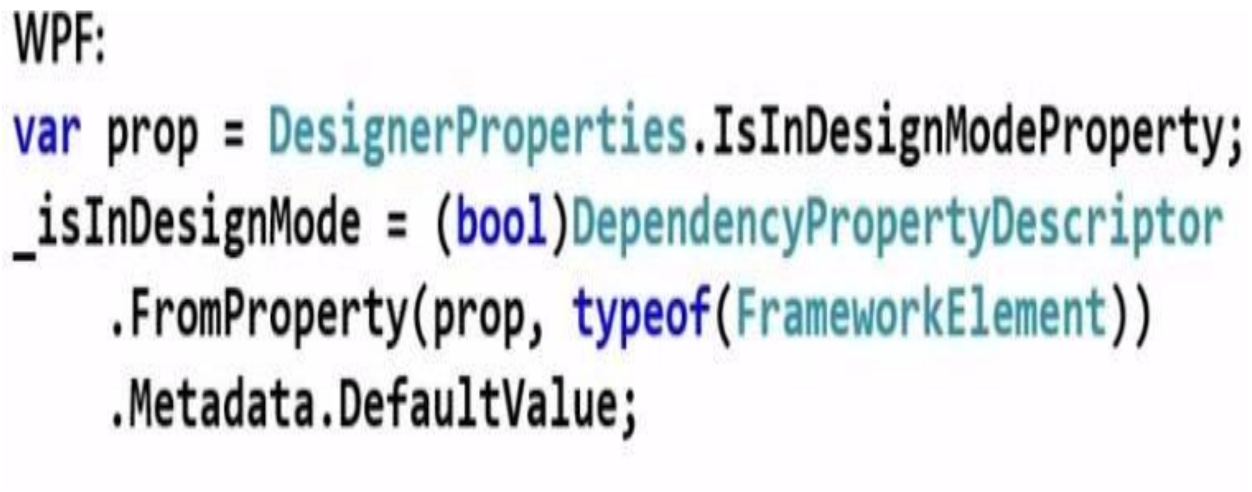
```
public ICommand MyCommand => new RelayCommand(

    () =>
    {
        //execute action
        Message = "clicked Button";
    },
    () =>
    {
        //return true if button should be enabled or not
        return true;
    }
);
```

If canExecute returns false, the Button will be disabled for the user

Before the action is really executed, canExecute will be checked again

Design time output

Before MVVMLight:



```
WPF:
var prop = DesignerProperties.IsInDesignModeProperty;
_isInDesignMode = (bool)DependencyPropertyDescriptor
    .FromProperty(prop, typeof(FrameworkElement))
    .Metadata.DefaultValue;
```

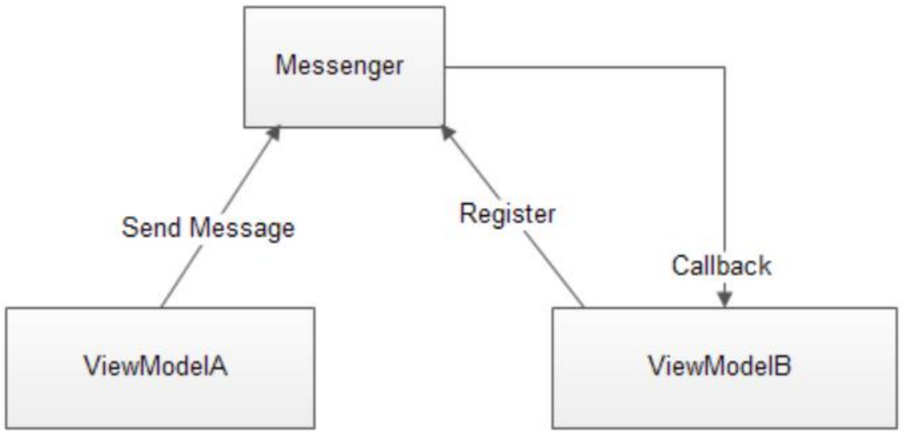
After:

```
if (IsInDesignMode)
{
   Title = "Hello MVVM Light (Design Mode)";
}
else
{
   Title = "Hello MVVM Light";
}
```

MVVMLightMessenger

MVVM Light Messenger is a class that allows exchange messages between objects. Messenger class is mainly used for sending messages between viewmodels. Messenger class decreases coupling between viewmodels. Every viewmodel can communicate with another viewmodel without any association between them.

Messenger is an implementation of Mediator pattern in MVVM Light toolkit. You can know more about Mediator pattern here.



Messenger classes in MVVM Light Toolkit

IMessenger

Interface which all the Messenger classes must inherit. Some methods are Register, Send, and Unregister.

Messenger

Messenger class is the implementation of IMessenger interface. This class is used for send and receive messages between objects.

Messenger Class Important Properties and Methods

1. Default (static) : IMessenger

Provides default instance of Messenger, allowing registering and sending messages in a static manner.

Reset (static) : void

Delete the default instance of Messenger. When you again use Default property of Messenger, it creates new Messenger class.

Register<TMessage>(object recipient, object token, bool receiveDerivedMessagesToo, Action<TMessage> action) : void

This method registers a recipient for a type of message TMessage. TMessage can be anything like int, string or custom class. MVVM Light toolkit also provides some classes which can be used for sending the message like NotificationMessage, PropertyChangedMessage<T> etc. I described these classes later in this section. ?

All four parameters usage are:

object recipient : ?

The object which will receive the message. You can use "this" for registering current object for receiving messages or you can specify other objects.

object token :

Token is a keyword for sending the message to those objects who register with that particular keyword. For example, if two recipients register with the "ViewModelA" token and "ViewModelB" token and message is send using "ViewModelA" token then only that object got the message that register with "ViewModelA" token.

bool receiveDerivedMessagesToo :

If we set this flag to "true", then the recipient also gets message types which inherit from <TMessage>. For example, if bike and car both implement the interface IAutomobile, then registering with the type <IAutomobile> and set the receiveDerivedMessagesToo flag to true, allows recipient to get bike and car messages too.

Action<TMessage> action :

The action delegate will be executed when message of <TMessage> is sent from the Send message.

Unregister<TMessage>(object recipient, object token, Action<TMessage> action): void

This method un-register a recipient for a type of message TMessage.

Send<TMessage>(TMessage message, object token) : void

This method sends messages to registered recipients. Only those recipients will receive messages that register for that particular <TMessage>.?

All two parameters usage are:

a. TMessage message :

Message parameter is an instance of type TMessage.

object token :

Token as explained earlier is a keyword for sending the messages to only those recipients that register for that particular keyword.?

Cleanup : void

This method scans the recipients list for the dead recipients. All recipients are stores as WeakReferences, they can be claimed during the garbage collection process. Cleanup method removes those recipients that are collected by garbage collector.

Messages Classes in MVVM Light Toolkit

1. MessageBase

Base class for all messages types used by the Messenger for sending messages. There are only two properties exists in this class Sender and Target. You can only set both properties in the constructor. Both properties are of type object. Setting both properties are not necessary. These exist only for sending indications to recipients.

NotificationMessage

Use for sending "string" message notification to recipient(s).

NotificationMessage<T>

Use for sending "string" message notification and a generic value <T> to recipient(s).

NotificationMessageAction

Use for sending "string" message notification with a callback Action. When the recipient processed the message, it can execute the callback Action.

NotificationMessageAction<TCallbackParameter>

Use for sending "string" message notification with a callback Action with one <TCallbackParamter> parameter. When the recipient processed with message, it can execute the callback Action with one parameter.

PropertyChangedMessage<T>

Use for sending the property name along with its old value and new value.

DialogMessage [deprecated]

Use for requesting to display the messagebox. This class also provides a callback Action. This callback is used for notify the sender about the user's choice in the message box.

GenericMessage<T>

Use for sending generic value to a recipient. Recipient can access the generic value using Content property.

You can create your own custom messages classes by inherit from the MessageBase class.