MVVM

Model View ViewModel

(Microsofts variation of Model-View-Presenter)

Model View ViewModel (MVVM)

- **Is a Microsoft specialization of the Presentation Model Pattern (MVP).**
- MVVM is targeted WPF and Xamarin.
- MVVM was designed to make use of specific functions in WPF (mostly data binding) to better facilitate the separation of View layer from the Model layer by removing virtually all “code behind” from the View layer.
- Instead of requiring Interactive Designers to write View code, they can use XAML (Expression Blend) and create bindings to the ViewModel, which is written and maintained by application developers.

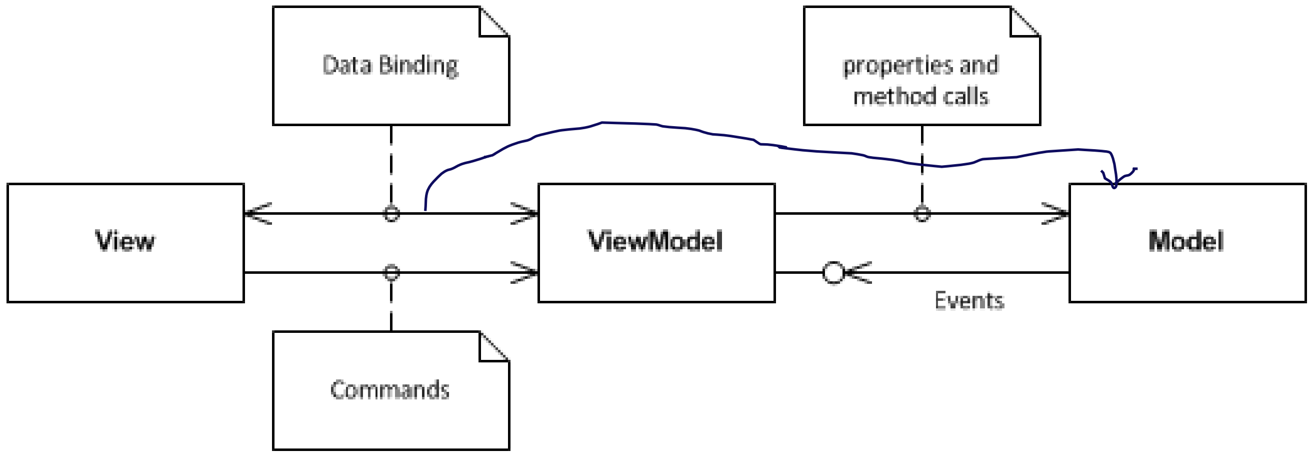
The spirit of MVVM

- Is building UIs that utilize platform enhancements in WPF to provide good separation between UI and business logic in order to make those UIs easier to maintain by developers and designers.

  – John Gossman (inventor of MVVM):
    - *Model/View/ViewModel is tailored for modern UI development platforms where the View is the responsibility of a designer rather than a classic developer. The designer is generally a more graphical, artistic focused person, and does less classic coding than a traditional developer.*

  – Martin Fowler (comment on PresentationModel):
    - *It's useful for allowing you **to test without the UI**, support for some form of **multiple view** and a **separation of concerns** which may make it easier to develop the user interface.*

MVVM Structure



- The View is connected to the ViewModel through data binding and sends commands to the View-Model.
  - **The ViewModel is unaware of the View**.
- The ViewModel may interact with Model through properties, method calls and may receive events from the model.
  - **The Model is unaware of the View-Model**.

Elements of the MVVM pattern: **Model**

As in the classic MVC pattern, the model refers to either:

- An object model that represents the real state content (an object-oriented approach), or
- The data access layer that represents that content (a data-centric approach).

Elements of the MVVM pattern: **View**

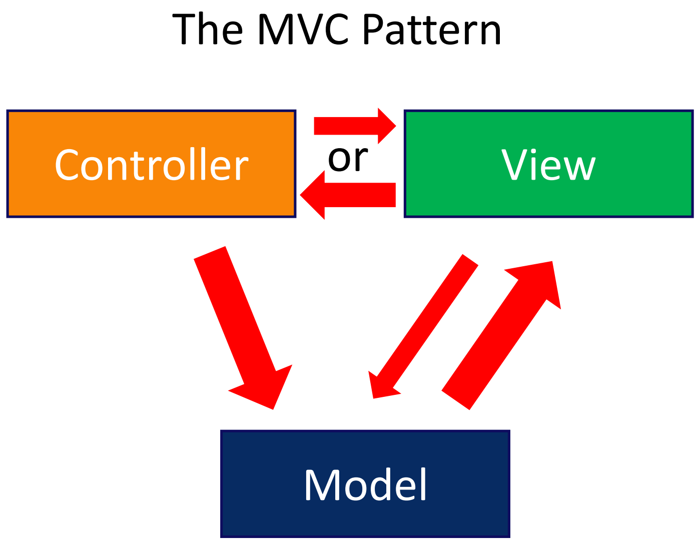
- As in the classic MVC pattern, the view refers to all elements displayed by the GUI such as windows, buttons, graphics, and other controls.
- A View may represent the hole window - or it may only represent a part of a window - typical a **user control** or DataTemplate.

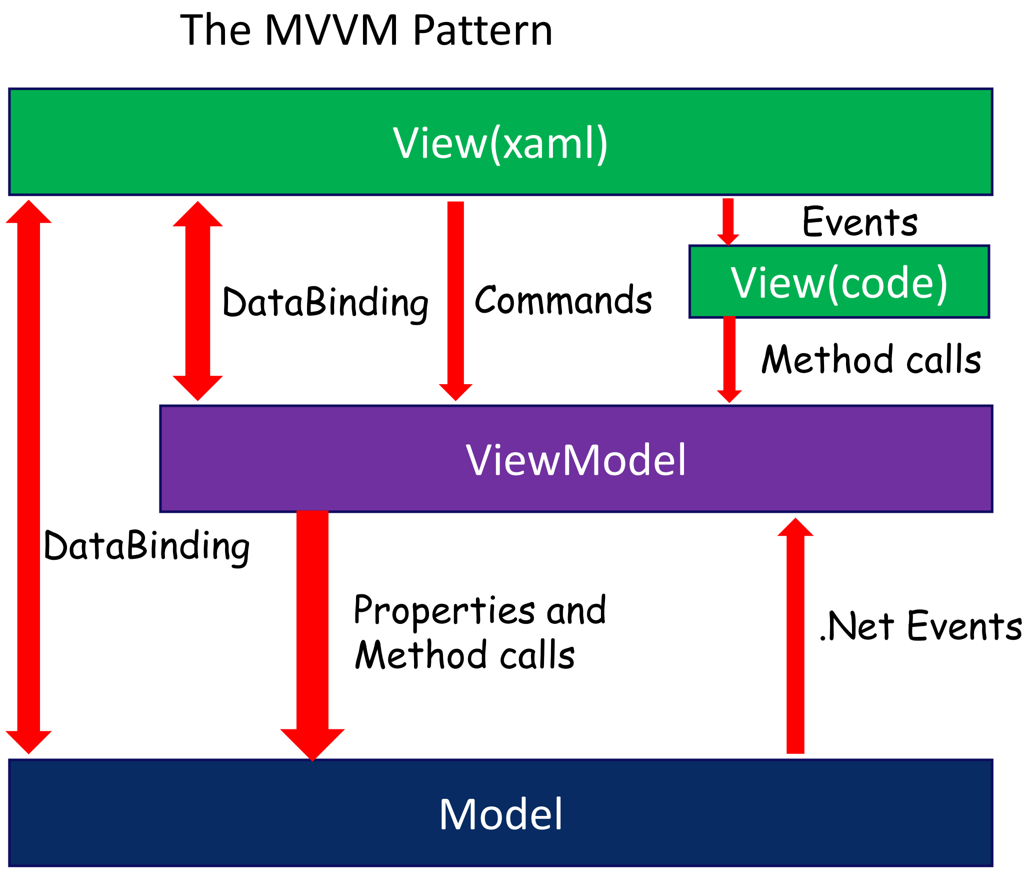
Elements of the MVVM pattern: **ViewModel**

- The ViewModel is a “Model of the View”
- Meaning it is an abstraction of the View that also serves in data binding between the View and the Model
- It could be seen as a specialized aspect of what would be a Presenter (in the MVP pattern) that acts as a data binder/converter that changes Model information into View information and passes commands from the View into the Model
- The ViewModel exposes public:
  - **properties**,
  - **Commands**

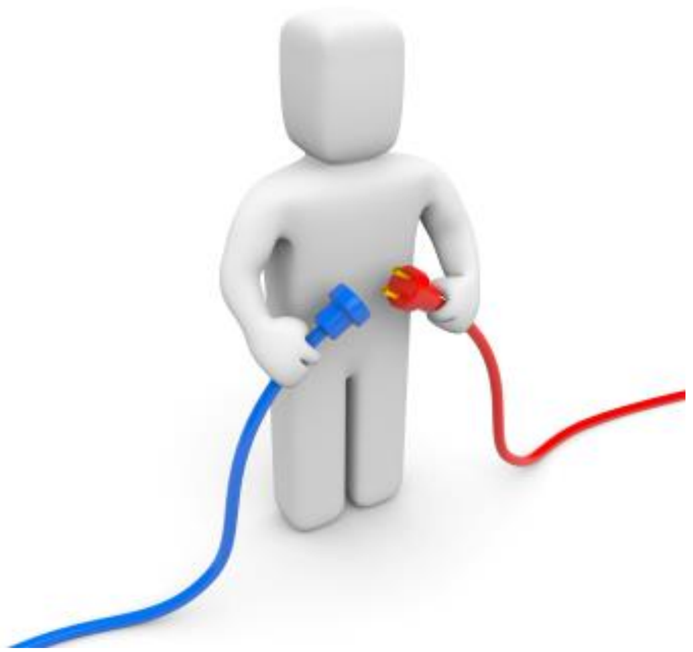
  For the View to bind to.

From MVC to MVVM

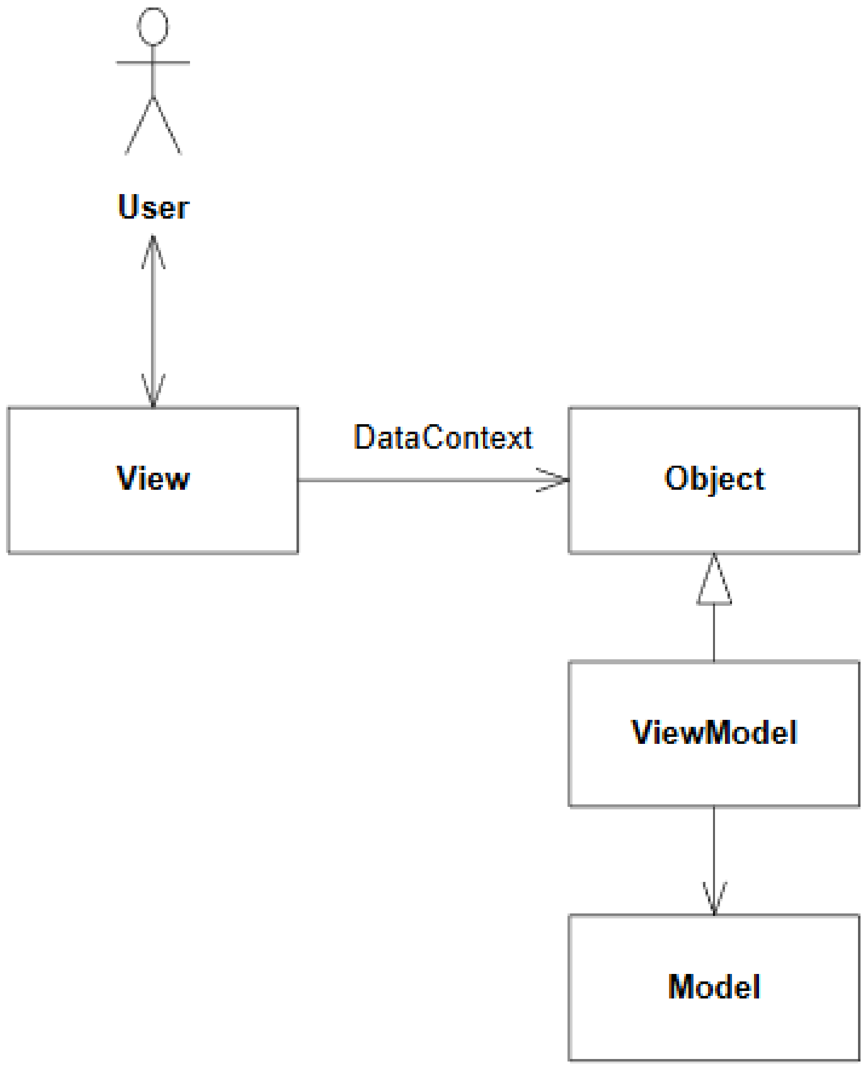




Connecting Views and ViewModels



View – ViewModel relation



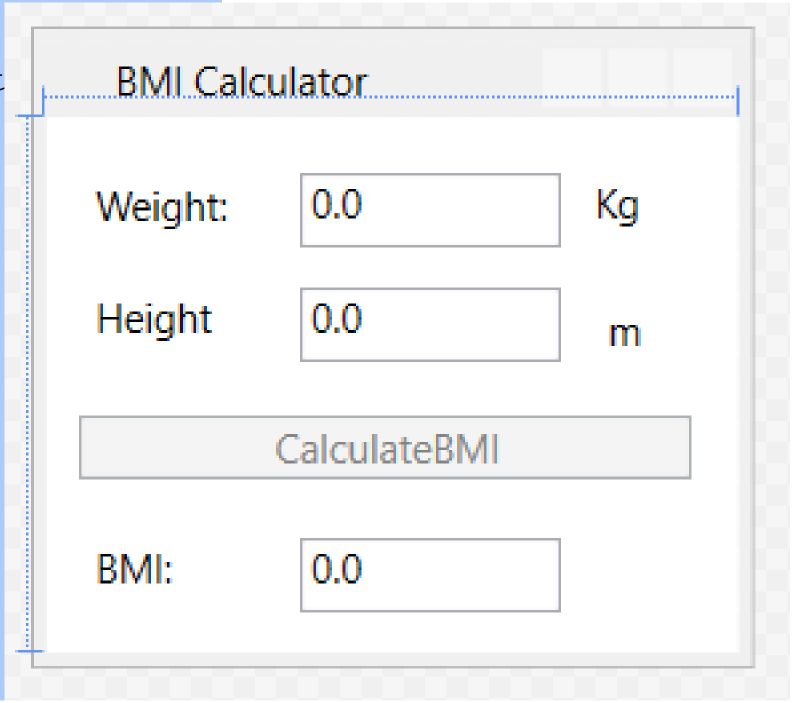
The communication between a View and its ViewModel is mainly done by use of Data Bindings where the View's DataContext property holds the reference to the ViewModel.

The relation from View to ViewModel is specified in XAML and may be obtained from a Locator or some kind of dependency injection.

Binding the View to the ViewModel

The DataContext of the View is set to the ViewModel to which the View shall bind.

```
<Window x:Class="BMICalculator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibilit
        xmlns:local="clr-namespace:BMICalculator"
        xmlns:viewModel="clr-namespace:BMICalculator.ViewModel"
        mc:Ignorable="d"
        Title="BMI Calculator" Height="195.098" Width="220.098">
    <Window.DataContext>
        <viewModel:BMIViewModel/>
    </Window.DataContext>
```



Binding the View to the ViewModel

```
class BMIViewModel : INotifyPropertyChanged
{
    private double _bmi;

    public double Height
    {
        get => bmiModel.Height;
        set
        {
            if (value != bmiModel.Height)
            {
                bmiModel.Height = value;
                OnPropertyChanged();
            }
        }
    }

    ....
```

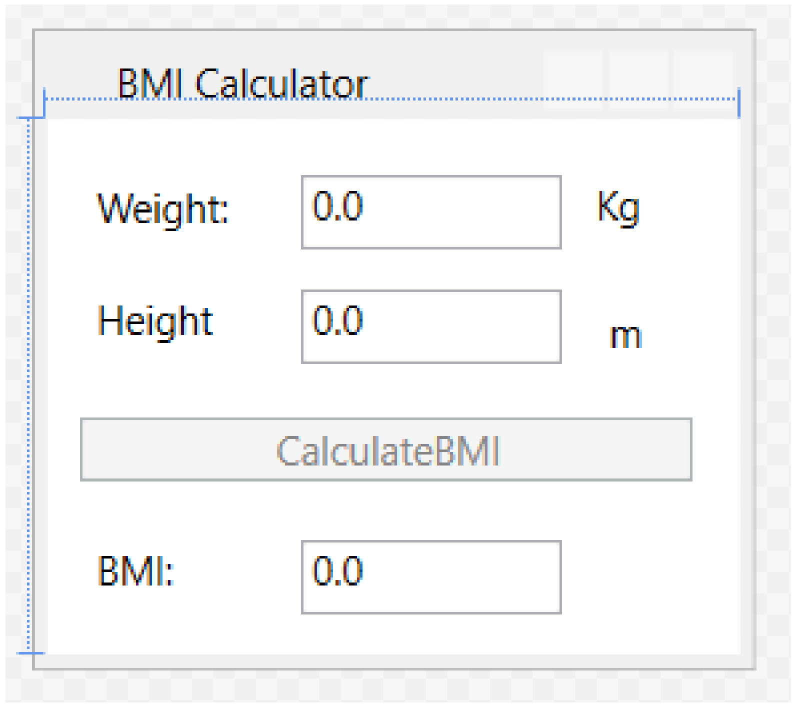
The ViewModel exposes properties, that the View can bind to.

It also implements the INotifyPropertyChanged interface.

Or use a MVVM framework base class like PRISM's BindableBase.

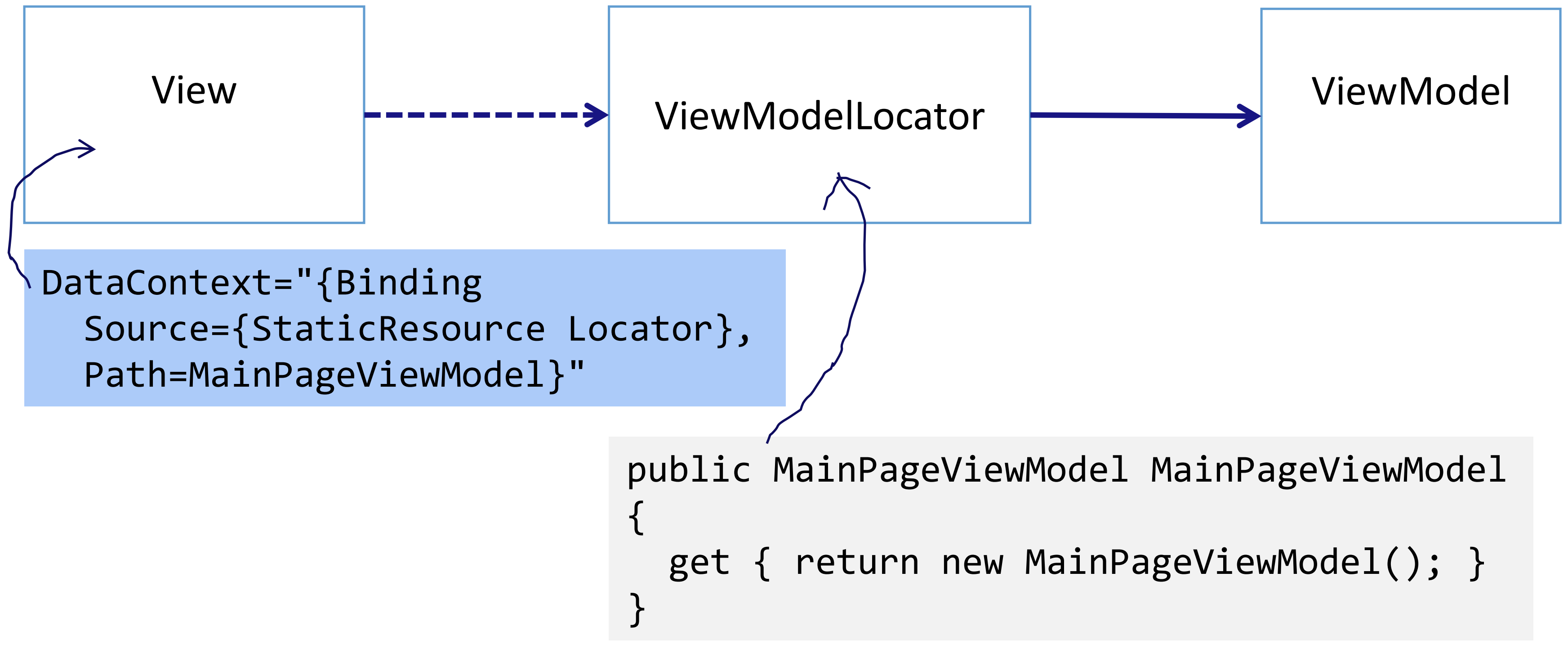
Binding the View to the ViewModel

```
<TextBox
    HorizontalAlignment="Left"
    Height="23"
    Margin="78,17,0,0"
    TextWrapping="Wrap"
    Text="{Binding Path=Height, StringFormat=F1,
Mode=TwoWay}"
    VerticalAlignment="Top"
    Width="80"/>
```



ViewModel Locator

ViewModel Locator Structure



```
DataContext="{Binding
   Source={StaticResource Locator},
   Path=MainPageViewModel}"
```

```
public MainPageViewModel MainPageViewModel
{
   get { return new MainPageViewModel(); }
}
```

- There are several variations of the ViewModelLocator pattern E.g.:
  - You can instantiate it as a resource in App.xaml
  - You can make it static
  - **You can use a MVVM framework implementation like** **`prism:ViewModelLocator.AutoWireViewModel="True"`**

View First: Using a ViewModel Locator

```
public class ViewModelLocator {
   public BMIViewModel BmiViewModel
   {  get{return new BMIViewModel(new BMIModel());}}
}
```

App.xaml:

```
<Application x:Class="UsingAViewModelLocator.App"
    xmlns:local="clr-namespace:UsingAViewModelLocator"
    StartupUri="MainWindow.xaml">
  <Application.Resources>
    <local:ViewModelLocator x:Key="ViewModelLocator" />
  </Application.Resources>
</Application>
```

The View:

```
<Window x:Class="UsingAViewModelLocator.MainWindow"
        Title="BMI Calculator" Height="350" Width="525"
        DataContext="{Binding
                     Source={StaticResource ViewModelLocator},
                     Path=BmiViewModel}" >
```

ViewModel Locators

- The ViewModel Locator shown here is very simple
- For big applications with many Views and ViewModels you may automate the Locator:
  - You can turn the Locator into a Dictionary and have all the ViewModels register themselves to the Locator during application startup
  - You can implement a naming convention in the Locator so when given a View with the name DemoView the Locator will return the ViewModel with the name DemoViewModel (by use of reflection)
  - **This is how the ViewModelLocator in Prism works**.

ViewModel First: Using a UserControl as View

```
<Window x:Class="UserControlAsView.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:vm="clr-namespace:BMICalculator"
    xmlns:local="clr-namespace:UserControlAsView"
    Title="User Control as View" Height="350" Width="525">
  <Window.Resources>
    <DataTemplate DataType="{x:Type vm:BMIViewModel}">
      <local:BmiView />
    </DataTemplate>
  </Window.Resources>
  <Grid>
    <ContentControl Content="{Binding Path=BmiViewModel}" />
   </Grid>
</Window>
```

```
<UserControl x:Class="UserControlAsView.BmiView" …>
```

```
public class MainViewModel
{
  public BMIViewModel BmiViewModel
  {
    get { return new BMIViewModel(new BMIModel()); }
```

```
public class BMIViewModel : INotifyPropert
    {
```

```
public class BMIModel
    {
```

Commands from View to ViewModel

- Binding the presentation layer directly to the properties of a view model works like a charm, but binding user input to the methods of the view model doesn't work at all
  - The built in WPF command routing can't reach the ViewModel!
- To overcome this limitation of WPF, different developers has come up with similar solutions:
  - Josh Smith's **RelayCommand**
  - Prism's **DelegateCommand**
- Both take advantage of the fact that the command properties of some WPF controls allow the use of command types other than RoutedCommand
- The commands just need to implement the ICommand interface
- The suggested way to use the RelayCommands or DelegateCommands is to add command properties to the view model, map the commands to methods in the view model, and then bind the command properties of the view model to the command properties of the WPF controls

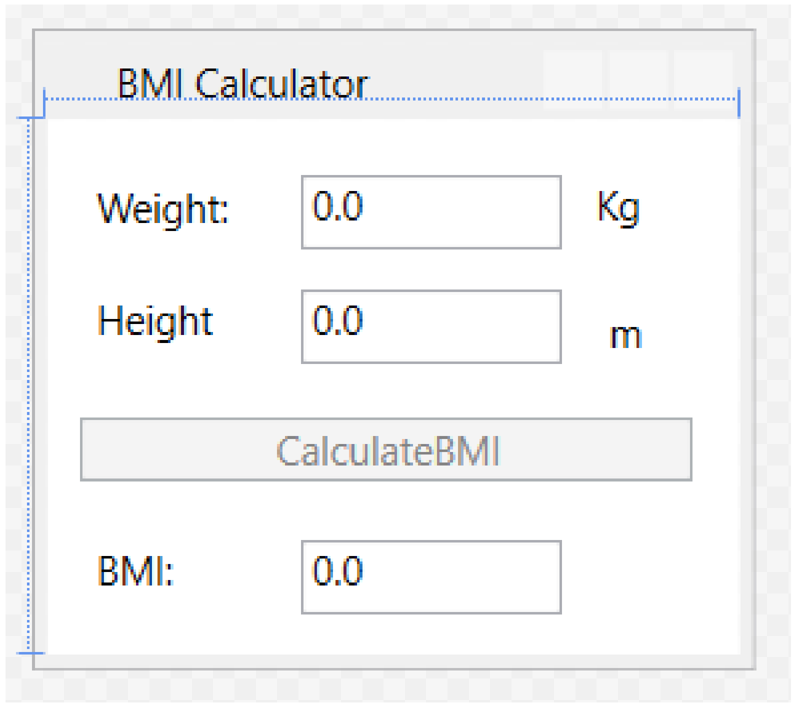
Binding to Commands

```
private ICommand _calcBMICommand;
public ICommand CalcBMICommand
{
    get
    {
        return _calcBMICommand ?? (_calcBMICommand =
            new RelayCommand(CalcBMI, CalcBMICanExecute));
    }
}

private void CalcBMI()
{
    BMI = bmiModel.CalculateBMI();
    OnPropertyChanged("BMI");
}

private bool CalcBMICanExecute()
{
    bool paramsAreValid = (Weight != 0.0 && Height != 0.0);
    return paramsAreValid;
}
```

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Binding to Commands

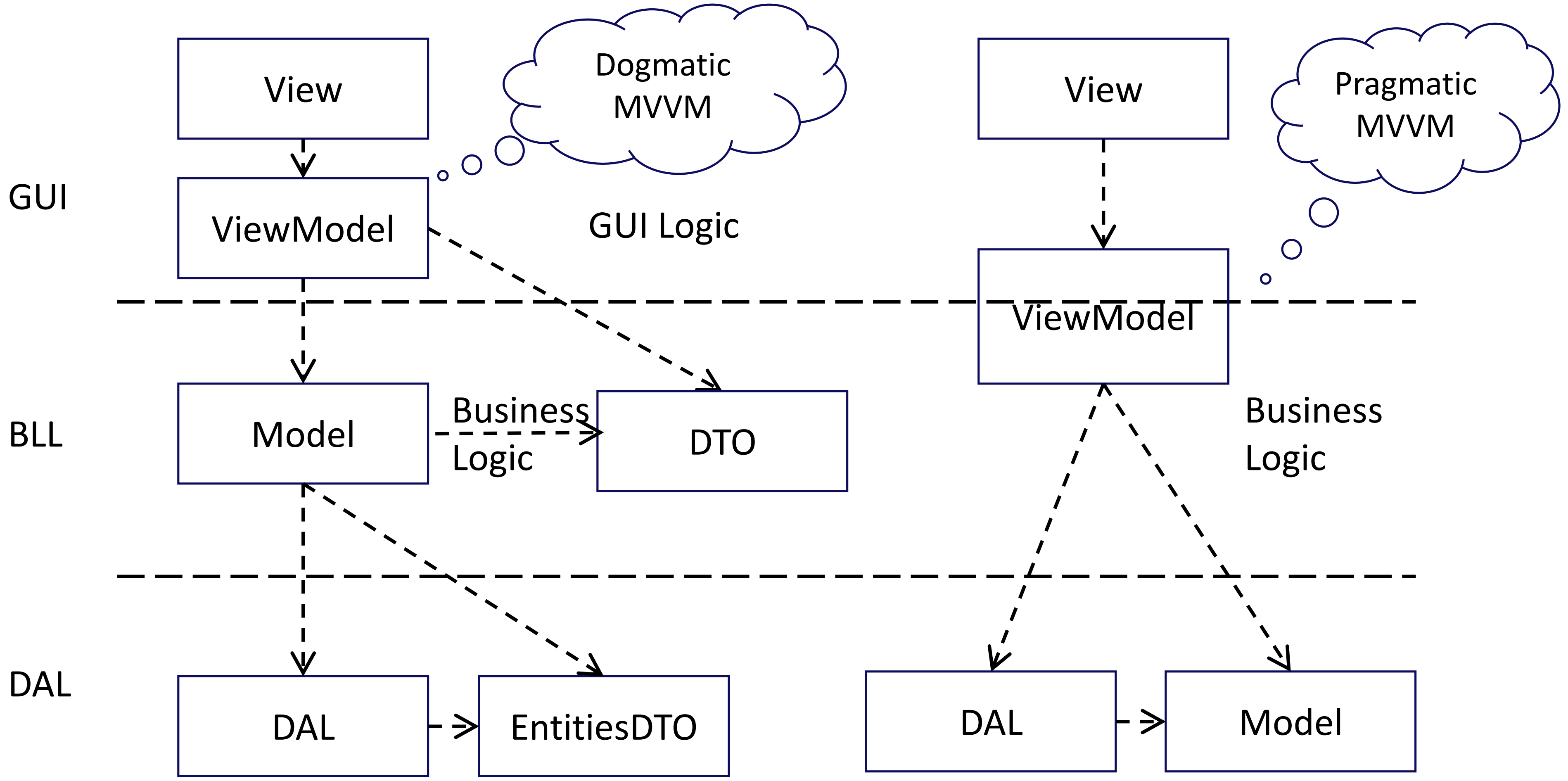
```
<Button Content="CalculateBMI"
  HorizontalAlignment="Left"
  Margin="10,91,0,0"
  VerticalAlignment="Top"
  Width="188"
  Command="{Binding CalcBMICommand, Mode=OneTime}"/>
```



MVVM and N-layer Architectures

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MVVM and N-layer Architecture



Open Source MVVM Frameworks

- MS Patterns&Practices, PRISM – installeres via Nuget ★★
- Rob Eisenberg, "Caliburn". http://caliburnmicro.com/ ★★
- MvvmCross https://github.com/MvvmCross/MvvmCross ★★
- Laurent Bugnion, "MVVM Light Toolkit“ http://www.mvvmlight.net/ ★
- Tony Sneed, Simple MVVM Toolkit https://blog.tonysneed.com/2016/05/25/simple-mvvm-toolkit-it-lives/

References & Links