XAM320

Design an MVVM ViewModel in Xamarin.Forms

Download class materials from
university.xamarin.com

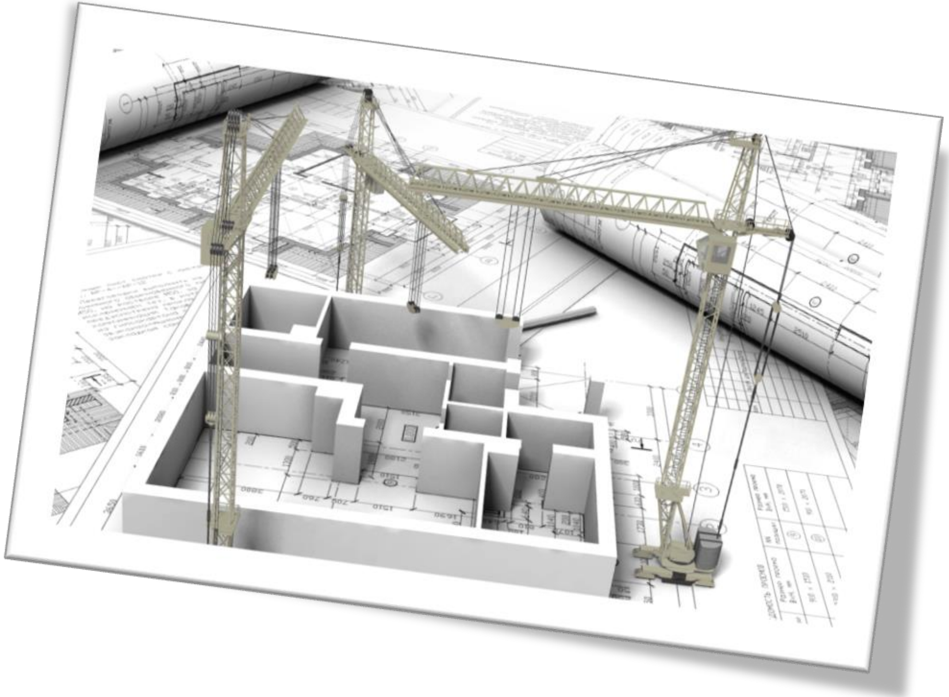
Microsoft

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

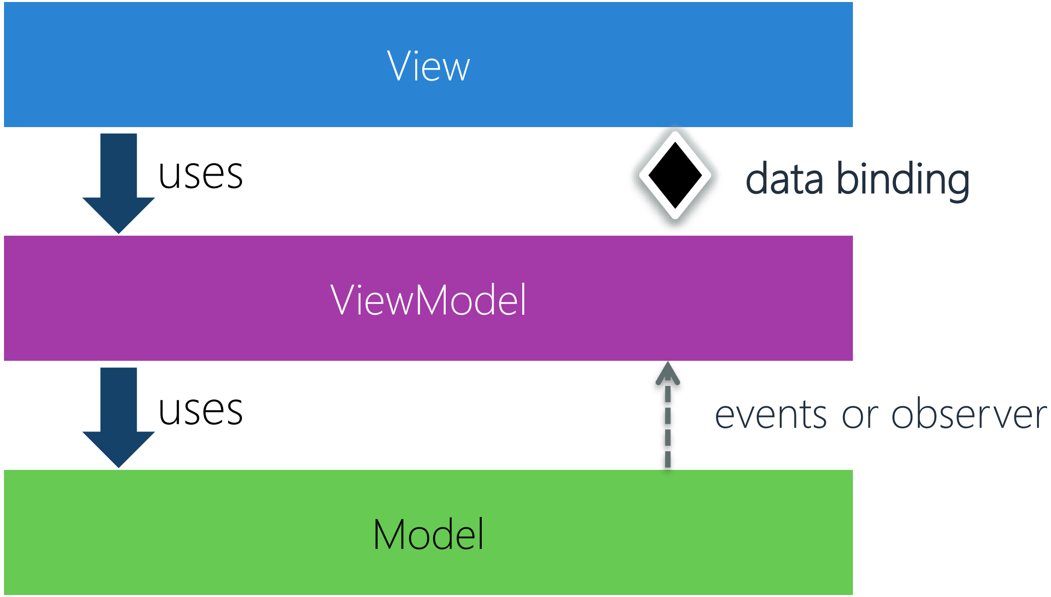
Objectives

1. Define Visual Behavior
2. Use Commands
3. Test MVVM based apps



[Reminder] Model-View-ViewModel

- MVVM is a layered, **separated presentation pattern** where a data binding engine takes the place of the controller / presenter



MVVM Libraries

- You can create your own MVVM support, but there are several popular MVVM libraries available for cross platform development
  - Prism [`pnpmvvm.codeplex.com`]
  - MvvmCross [`github.com/MvvmCross`]
  - MvvmLight [`codeplex.com/MvvmLight`]
  - ReactiveUI [`reactiveui.net`]
  - Caliburn.Micro [`github.com/Caliburn-Micro`]
  - MvvmHelpers [`codeplex.com/MvvmHelpers`]
  - [your favorite goes here] ☺

Define Visual Behavior

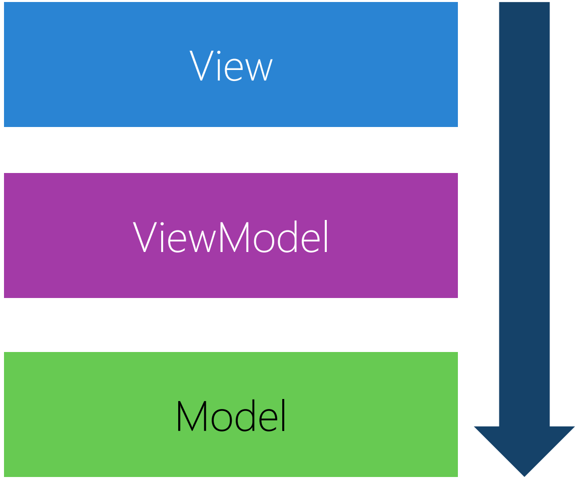
Xamarin University

Tasks

1. Control and activate events with selection
2. Utilize properties to define Visual Behavior
3. Employ Data Triggers

View vs. ViewModel

- ViewModel is intentionally designed to support the View, but should be written to be **UI-agnostic**
  - it should *not* have dependencies on anything in Xamarin.Forms



Each layer should only have direct knowledge about the layer below it

Selection in XAML

- Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
<ListView ItemsSource="{Binding Employees}"
          SelectedItem="{Binding SelectedEmployee, Mode=TwoWay}" />
```

Make sure to mark it *two-way* so ViewModel is notified when selection is altered by the UI

Xamarin University

Dealing with Selection

- Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
public partial class MainViewModel : BaseViewModel
{
   ...
   private EmployeeViewModel selectedEmp;
   public EmployeeViewModel SelectedEmployee {
      get { return selectedEmp; }
      set { selectedEmp = value; RaisePropertyChanged("SelectedEmployee"); }
   }

   public MainViewModel() {
         SelectedEmployee = Employees
    }
}
```

Setter called when selection is changed

Dealing with Selection

- Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
public partial class MainViewModel : BaseViewModel
{
   ...
   private EmployeeViewModel selectedE
   public EmployeeViewModel SelectedEm
      get { return selectedEmp; }
      set { selectedEmp = value; Raise
   }

   public MainViewModel() {
        SelectedEmployee = Employees.FirstOrDefault();
    }
}
```

When UI supports "selection" vs. activation, view model can default or change selection based on runtime decisions, all in a unit-testable way

Working with visual properties

- Assume a business requirement is to change the color of the employee's name in the UI if they are a supervisor

```
partial class EmployeeViewModel
{
    public Color NameColor { get; }
}
```

Avoid this! `Color` is a Xamarin.Forms specific type

... this is better but still not ideal – colors should be determined by the designer role and view code

```
partial class EmployeeViewModel
{
    public string NameColor { get; }
```

What we *really* want to do here is to have our UI change based on state properties such as **bool** or enumerations – we could do this with bindings and value converters

Working with visual properties

- Assume a business requirement is to change the color of the employee's name in the UI if they are a supervisor

```
partial class
{
    public Col
}
```

```
partial class EmployeeViewModel
{
    public bool IsSupervisor {
        get { ... }
        private set {
    }
}
```

r is a
pecific type

Let's expose a boolean property indicating whether the employee has subordinates ...

... this is better but still not ideal – colors should be determined by the designer role and view code

```
del

    public string TitleColor { get; }
}
```

Xamarin University

Working with visual properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsSupervisor}"
                   Value="True">
         <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label
      <Da                              el"
                                     ng IsSupervisor}"

         <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Assign default value – this is used when no trigger is matched

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTri ger TargetType="Label"
                 inding="{Binding IsSupervisor}"
                 ="True">
                 ="TextColor" Value="Blue" />

   </Label.Triggers>
</Label>
```

Can have zero or more *triggers* in the triggers collection exposed by the `Triggers` property

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsSupervisor}"
                   Value="True">
                      y="TextColor" Value="Blue" />
```

DataTrigger is used to change visual properties of an **Element** based on data binding

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsSupervisor}"
                   Value="True">
         <Setter Property="TextColor" Value="Blue" />
```

Binding property identifies the ViewModel property the Data Trigger is watching

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsSupervisor}"
                   Value="True">
         <Setter P   perty="TextColor" Value="Blue" />
```

… and a comparison test for that binding; e.g. when IsSupervisor = true

Visual Behavior through properties

- Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextCo
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding I   pervisor}"
                   Value="True">
        <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Has one or more setters to change properties when the trigger condition is matched

This is completely dynamic and is driven completely through the binding engine – so if the property changes at runtime, the trigger is re-evaluated and applied or removed!

Value Converters

- Value Converters allow for *type mismatch* conversions – e.g. when the data does not match the UI requirements
- This conversion task is often taken up by the VM instead – reducing the need for value converters
- Still useful to have more primitive converters for bindings

```
BooleanToColorConverter
```

```
ArrayToStringConverter
```

```
DoubleToIntegerConverter
```

```
NotBooleanConverter
```

```
IntegerToBooleanConverter
```

MVVM + other patterns

- MVVM is not the only design pattern needed, often need to utilize other patterns to provide necessary features through abstractions

Dependency Injection

Factory and Singleton

Command

Navigation

Alerts + Prompts

Messages

Managing navigation

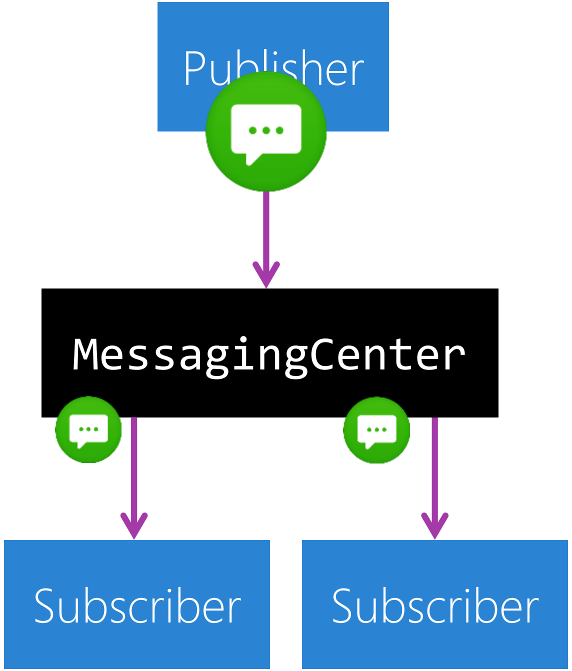
- Screen navigation can be handled in different ways – easiest is just to have an app-specific service that *knows* the screens which the VM uses

```
public enum AppScreen { Main, Detail, Edit, ... }

public class NavigationManager
{
   public Task<bool> GotoScreen(AppScreen screen) {...}
   public Task<bool> GoBack() { ... }
}
```

Enum defines the screens, and the class implements the navigation using the known app structure – master / detail, **`NavigationPage`**, etc.

Loosely-coupled messages

- Another common requirement is communication between unrelated app components in a loosely-coupled fashion
  - VM to VM
  - service to VM
- This is easily solved with the built-in `MessagingCenter`



Publishing a message

- Publisher passes message key and optional parameter

Publisher identifies sending type and parameter type through generic parameters

```
MessagingCenter.Send<MainViewModel, ItemViewModel>(
        this, "Select", selectedItem);
```

Subscribing to a message

❖ Subscribers identify the message by the sender type and message key and provide a delegate callback to run when message is received

```
MessagingCenter.Subscribe<MainViewModel, ItemViewModel> (
    this, "Select",
    (mainVM, selectedItem) => {
       // Action to run when "Select" is received
       // from MainViewModel
    });
```

Combination of the **sender type**, **string message**, and **parameter type** is the key for the message recipient – these must match between publisher and subscriber

Individual Exercise

Driving behavior through properties

Summary

1. Control and activate events with selection
2. Utilize properties to define Visual Behavior
3. Employ Data Triggers

Use Commands

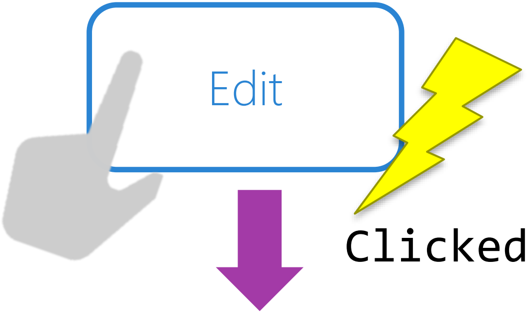
Xamarin University

Tasks

1. Implement the ICommand interface
2. Generalize a command

Event Handling

- UI raises events to notify code about user activity
  - `Clicked`
  - `ItemSelected`
  - …
- The downside is that these events **must be handled** in the code behind file



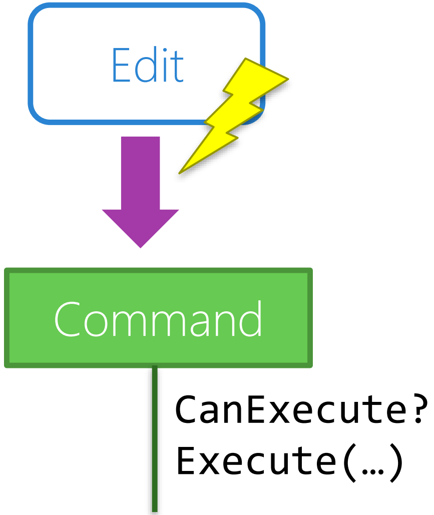
```
public MainPage()
{
    ...
    Button editButton = ...;
    editButton.Clicked += OnClick;
}

void OnClick (object sender, EventArgs e)
{
     ...
}
```

Commands

- Microsoft defined the **ICommand** interface to provide a commanding abstraction for their XAML frameworks

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Can provide an optional parameter (often **null**) for the command to work with for context



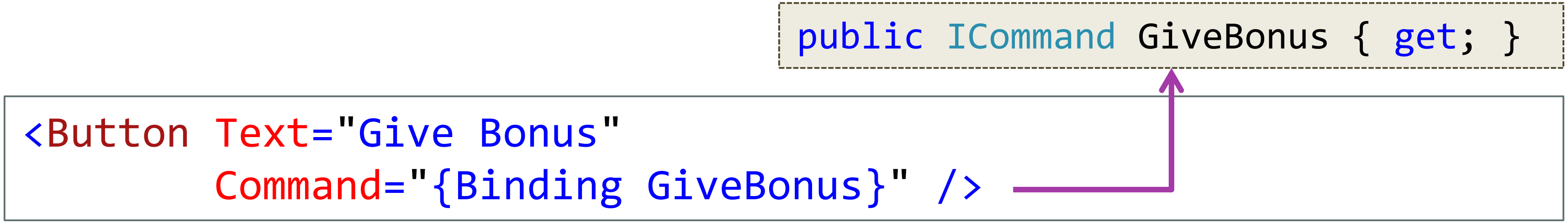
Commands in Xamarin.Forms

- A few Xamarin.Forms controls expose a **`Command`** property for the main action of a control

`Button` `Menu` `ToolbarItem` `TextCell`

Commands in Xamarin.Forms

- A few Xamarin.Forms controls expose a **Command** property for the main action of a control



```
public ICommand GiveBonus { get; }
```

```
<Button Text="Give Bonus"
        Command="{Binding GiveBonus}" />
```

Can data bind a property of type **ICommand** to the **Command** property

Gesture-based commands

- Xamarin.Forms also includes a **`TapGestureRecognizer`** which can provide a command interaction for other controls or visuals

```
<Image Source="IDareYouToTapMe.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Command="{Binding BeBraveCommand}"
            CommandParameter="TheyTookTheDare!" />
    </Image.GestureRecognizers>
</Image>
```

`CommandParameter` property supplies the command's parameter – in this case as a **`string`**

Implementing commands in the VM

- Command should be exposed as a public property from the ViewModel

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public ICommand GiveBonus { get; private set; }
    ...
    public EmployeeViewModel(Employee model) {
        this.model = model;
        GiveBonus = new GiveBonusCommand(this);
    }
    ...
}
```

```
public class GiveBonusCommand : ICommand
```

Implementing ICommand

- `ICommand` has three required members you must implement

`CanExecute` is called to determine whether the command is valid, this can enable / disable the control which is bound to the command

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Implementing ICommand

- ICommand has three required members you must implement

Execute is called to actually run the logic associated with the command when the control is activated – it will only be called if CanExecute returned true

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Implementing ICommand

- **`ICommand`** has three required members you must implement

`CanExecuteChanged` is an event which the binding will subscribe to, the ViewModel should raise this event when the validity of the command changes

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

The binding will then call **`CanExecute`** and enable / disable the UI in response

```
public partial class GiveBonusCommand : ICommand
{
    public event EventHandler CanExecuteChanged = delegate {};

    MainViewModel viewModel;
    public GiveBonusCommand(MainViewModel vm) {
        this.viewModel = vm;
    }

    public bool CanExecute(object parameter) {
        return this.viewModel.SelectedEmployee != null
            && (DateTime.Now - this.viewModel.SelectedEmployee.HireDate)
                  .TotalHours > 8;
    }

    public void Execute(object parameter) {
        this.viewModel.SelectedEmployee.GiveBonus(1000);
    }

    public void RaiseCanExecuteChanged() {
        CanExecuteChanged(this, EventArgs.Empty);
     }
}
```

Command relies heavily on the data in the ViewModel … could we move this logic?

Implementing commands generically

- Can use built-in **Command** and **Command<T>** to forward command to VM

```
public class Command<T> : ICommand
{
    Action<T> _function;
    public void Execute(object parameter) {
        _function.Invoke((T) parameter);
    }

    public bool CanExecute(object parameter)  {...}
    public event EventHandler CanExecuteChanged;
}
```

Initialize with delegates for each of the required methods – then you can define each command with logic in the ViewModel

Using delegate commands

- **Command<T>** and **Command** provides mechanism to centralize the logic for the commands into the VM

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public ICommand GiveBonus { get; private set; }
    public EmployeeViewModel(Employee model) {
        GiveBonus = new Command(OnGiveBonus, OnCanGiveBonus);
    }

    void OnGiveBonus() { ... }
    bool OnCanGiveBonus() { return ... }
}
```

Existing MVVM Libraries

- Easy to roll your own MVVM support, but there are several really good MVVM libraries available for cross platform development which include a lot of additional features
  - Prism [`pnpmvvm.codeplex.com`]
  - MvvmCross [`github.com/MvvmCross`]
  - MvvmLight [`codeplex.com/MvvmLight`]
  - ReactiveUI [`reactiveui.net`]
  - Caliburn.Micro [`github.com/Caliburn-Micro`]
  - MvvmHelpers [`codeplex.com/MvvmHelpers`]
  - [your favorite goes here] ☺

Flash Quiz

Xamarin University

Flash Quiz

① Commands are *not* supported on which control?

a) Button

b) Switch

c) MenuItem

d) Trick question - commands are supported on all of them!

Flash Quiz

① Commands are *not* supported on which control?

a) Button

b) [✓ Switch]

c) MenuItem

d) Trick question - commands are supported on all of them!

Xamarin University

Flash Quiz

② Commands are described through ___________.

a) IDelegateCommand

b) DelegateCommand

c) ICommand

d) Command

Flash Quiz

② Commands are described through ___________.

a) IDelegateCommand

b) DelegateCommand

c) ICommand

d) Command

Group Exercise

Using commands to run behavior

Xamarin University

Summary

1. Implement the ICommand interface
2. Generalize a command

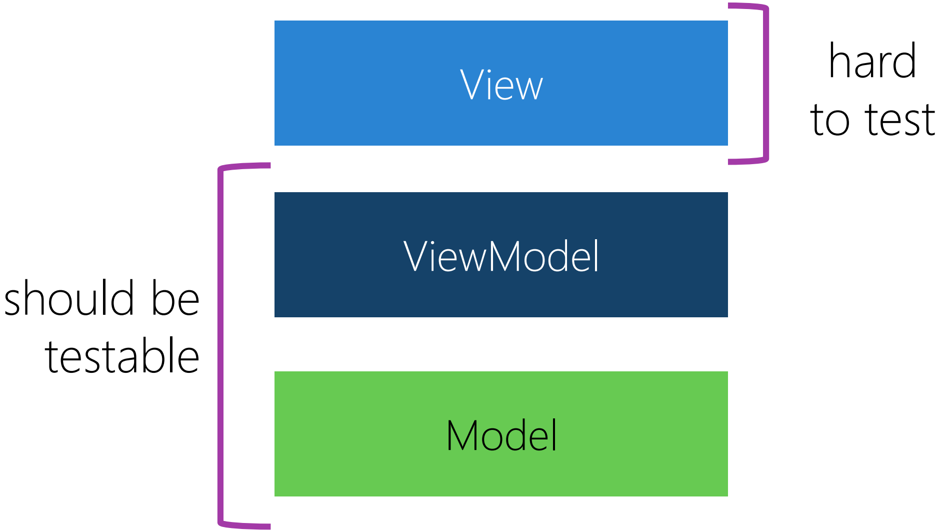
Test MVVM based apps

Tasks

1. UnitTest the ViewModel

Testing Surface

- Unit tests involve testing small, isolated pieces of our application independently; that's very hard to do for tightly coupled GUI applications
- Testable code is code which does not have dependencies on a UI being present



Testing the ViewModel

- ViewModel can be tested independently of the UI / platform
- Allows for testing of business logic *and* visual logic
- Can use well-known unit testing frameworks such as NUnit or MSTest

Testing the ViewModel

set properties and invoke command – just like UI would

```
[TestMethod]
void Employee_GiveBonus_Succeeds()
{
   var data = new Employee(...);
   var vm = new EmployeeViewModel(data);
   vm.GiveBonus.Execute("500");

   Assert.AreEqual(500,
          data.GetNextPaycheckData().Extras);
}
```

... and then test the results to verify it does what you expect

Demonstration

Adding unit tests for View Models

Summary

1. UnitTest the ViewModel

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft