Microsoft

Xamarin University

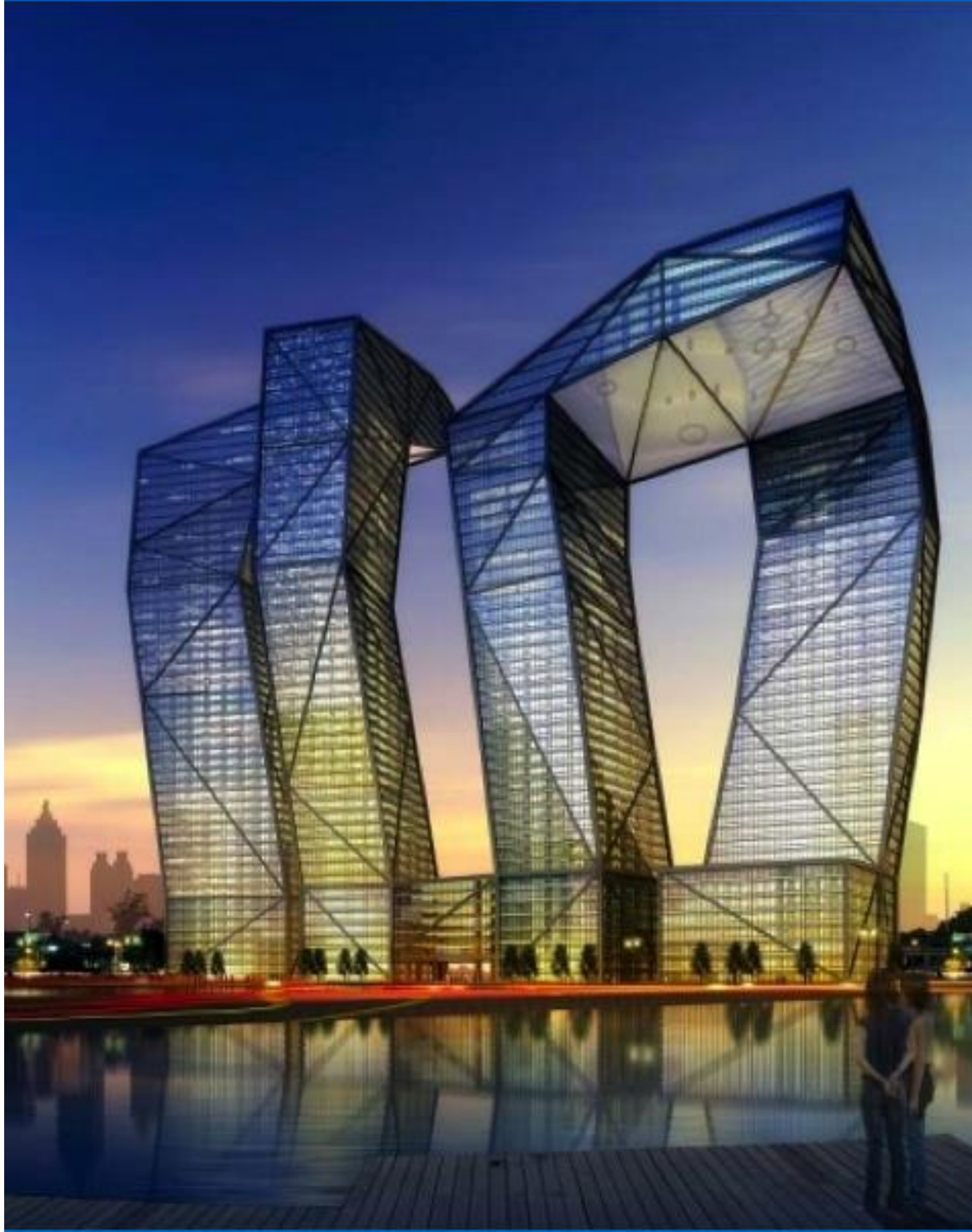
XAM301

# Mobile Application Architecture

Download materials from university.xamarin.com

# Objectives

1. Apply Model-View-Controller to a Xamarin.iOS app
2. Apply Model-View-Presenter to a Xamarin.Android app
3. Apply Model-View-ViewModel to a Xamarin.Forms app



# Separate presentation and domain

To maximize code sharing, *separate the presentation and domain layers*, this is referred to as the **Separated Presentation Pattern**

Ensure that any code that manipulates presentation *only* manipulates presentation, pushing all domain and data source logic into **clearly separated** areas of the program.

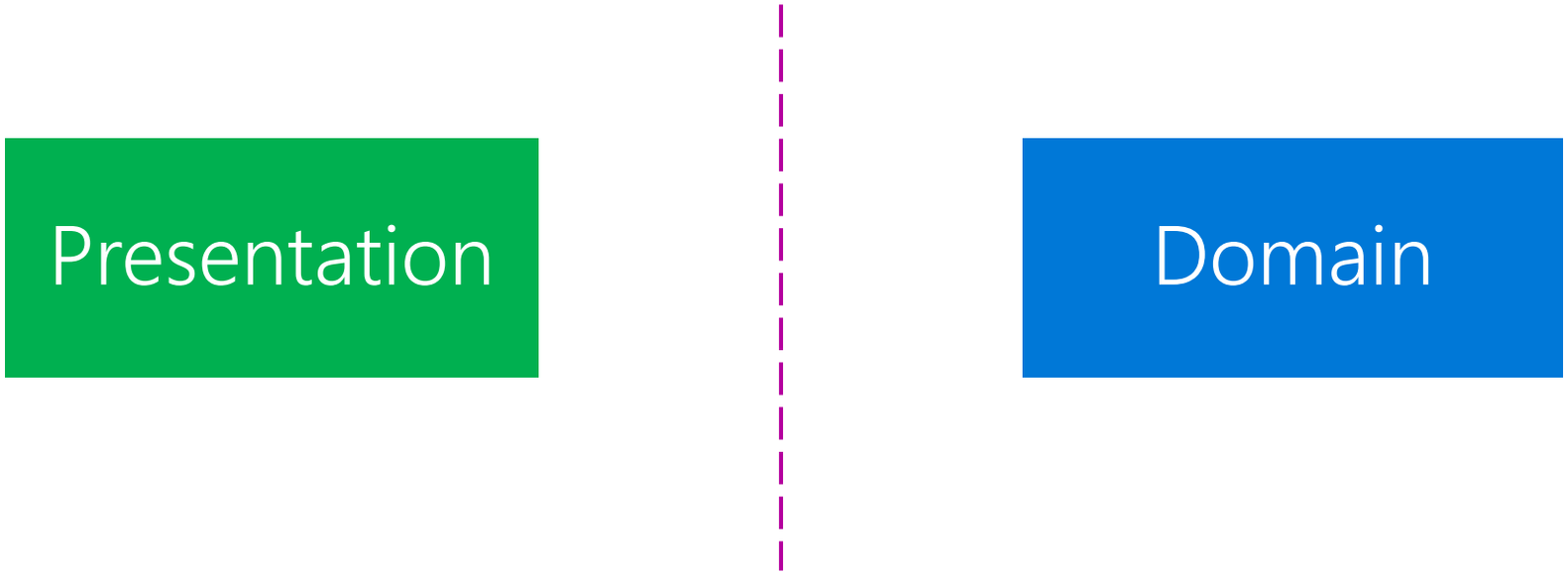
http://martinfowler.com/eaaDev/SeparatedPresentation.html

# Separated presentation patterns

Separated presentation patterns are patterns and rules used to help developers separate code into logical layers – specifically to separate UI from business code

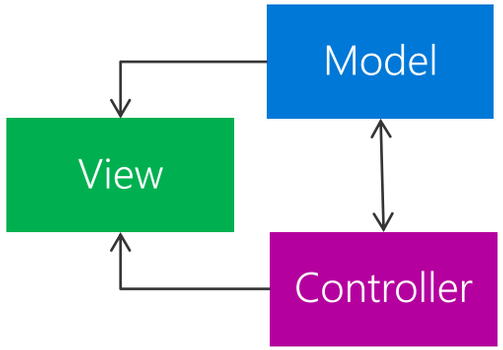
# Separated presentation patterns

Separated presentation patterns are patterns and rules used to help developers separate code into logical layers – specifically to separate UI from business code

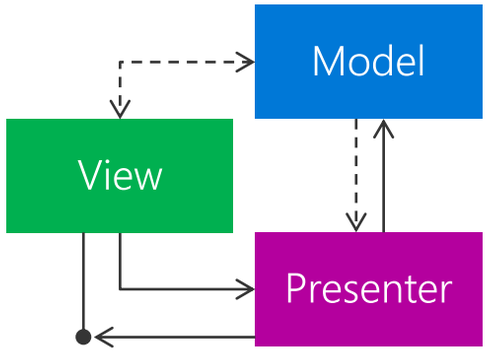


# Common separation presentation patterns

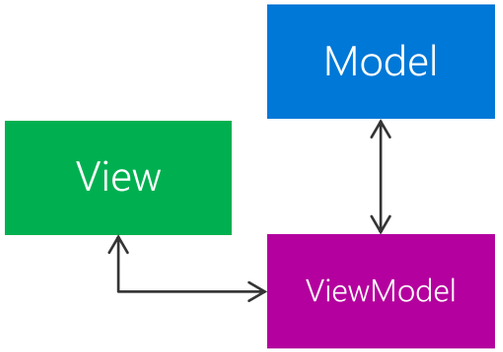
Choosing a presentation pattern for your application depends on the APIs for the development platform and the developer's personal preference




Model-View-Controller (MVC)




Model-View-Presenter (MVP)




Model-View-ViewModel (MVVM)

# Why standardize on a style?

There are several key benefits to selecting a well-known architectural style


Improvements can be shared across apps

Easier for developers to onboard to projects

Often can share components between apps

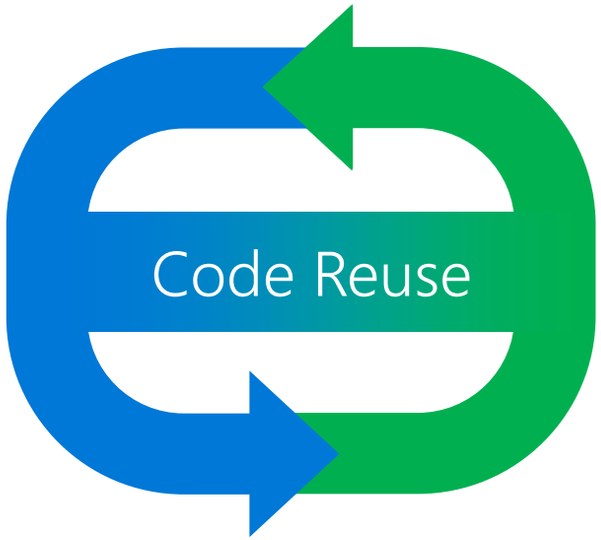
# Use the Single Responsibility Principle

Every object should be responsible for a **single piece of functionality**, and should only have **one reason to change**

# Benefits of separating concerns

Separating the presentation code from the domain/logic can help developers maintain, and scale and test their code





# Apply Model-View-Controller to a Xamarin.iOS app

# Tasks

1. Architect an iOS app using MVC

# MVC frameworks

Many popular development frameworks are designed to use the Model-View-Controller pattern to separate business logic from UI

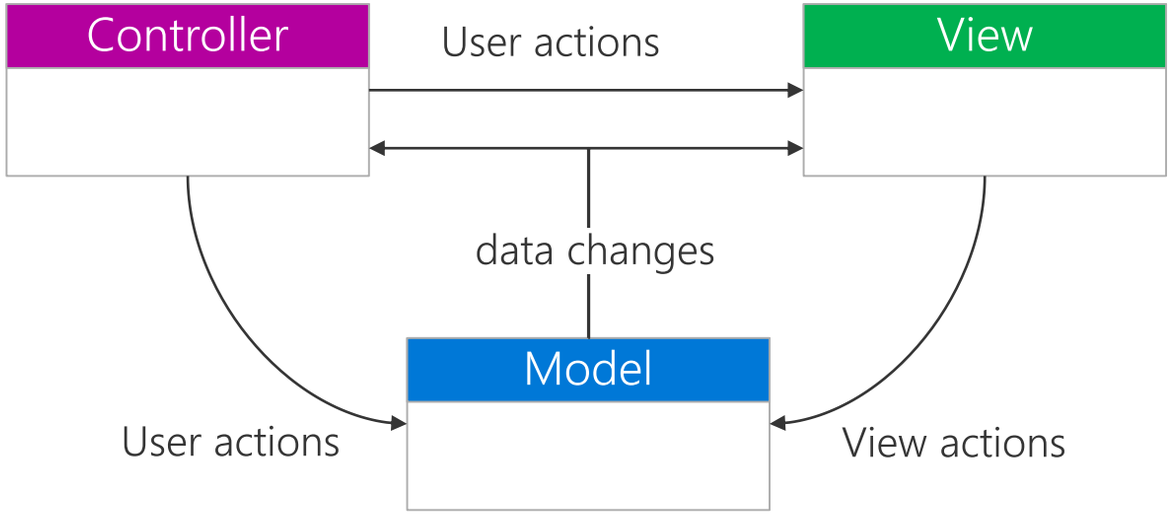

ASP.NET MVC


Ruby on Rails

Xamarin.iOS

# Model-View-Controller (MVC)

MVC was one of the first formalized structural styles for building UI-based architectures where the presentation was separated from the logic and data driving it



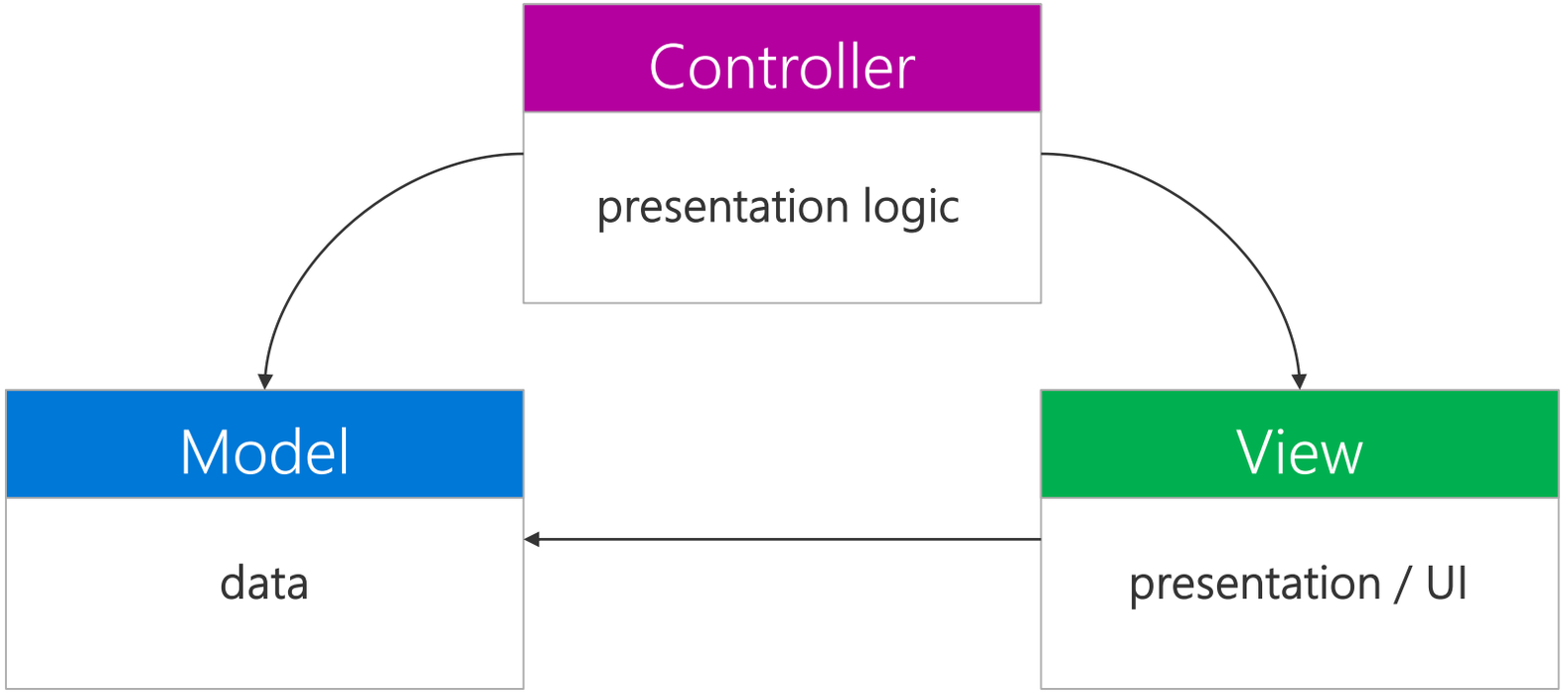

http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

"Probably the widest quoted pattern in UI development is Model View Controller (MVC) - it's also the most misquoted. I've lost count of the times I've seen something described as MVC which turned out to be nothing like it."

https://www.martinfowler.com/eaaDev/uiArchs.html

# Model-View-Controller (MVC)

MVC uses three layers to separate data, presentation, and interaction logic



# MVC types

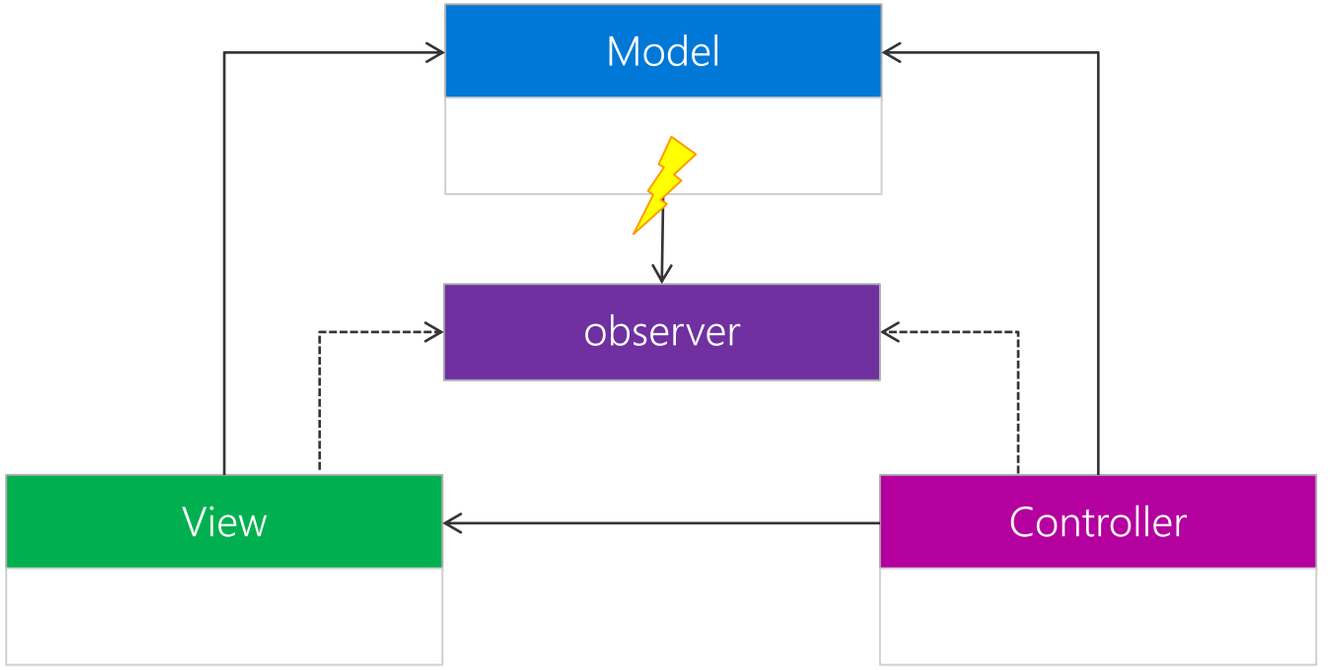
The Model-View-Controller pattern is typically implemented in one of two ways depending on the requirements of the domain logic

Active model

Passive model

# MVC with an active model

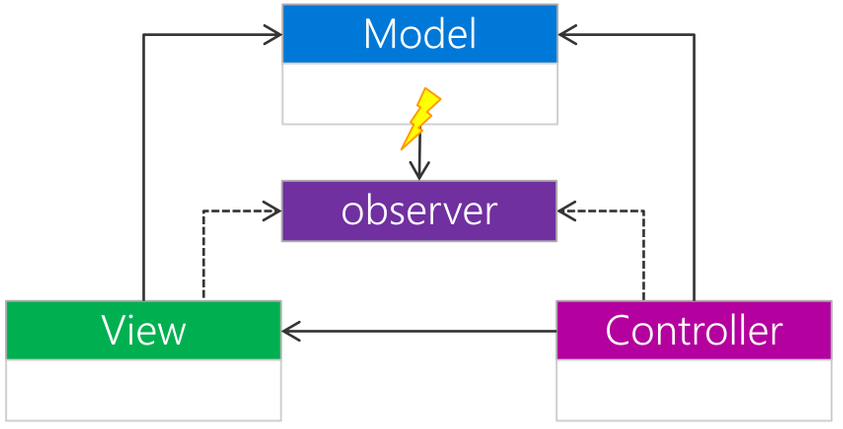
The MVC **active model** style is where the model changes independently and notifies the view/controller using the *observer* pattern



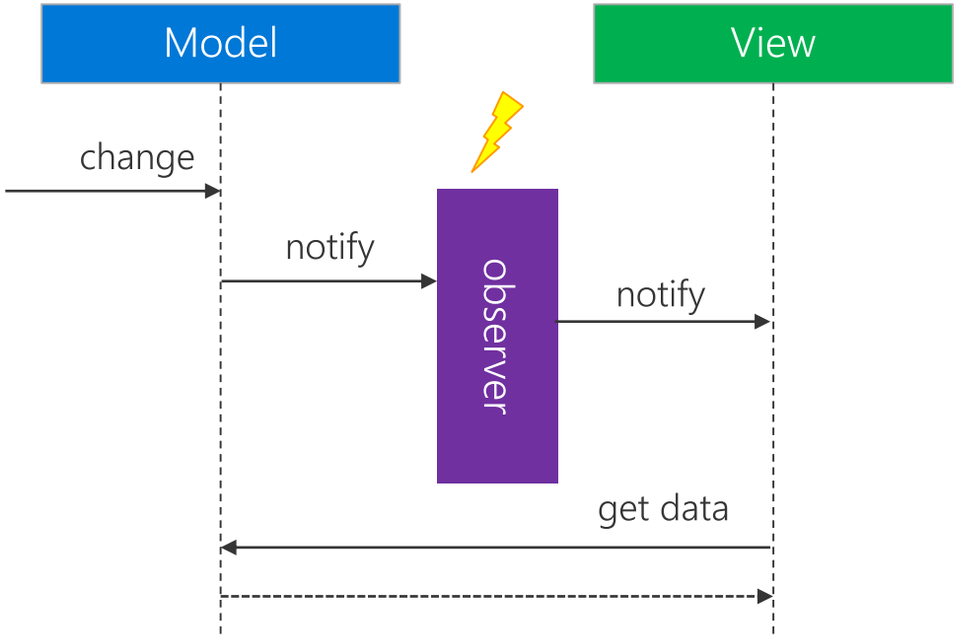

Model-View-Controller
(active model)

# MVC with an active model

The MVC **active model** style is where the model changes independently and notifies the view/controller using the *observer* pattern

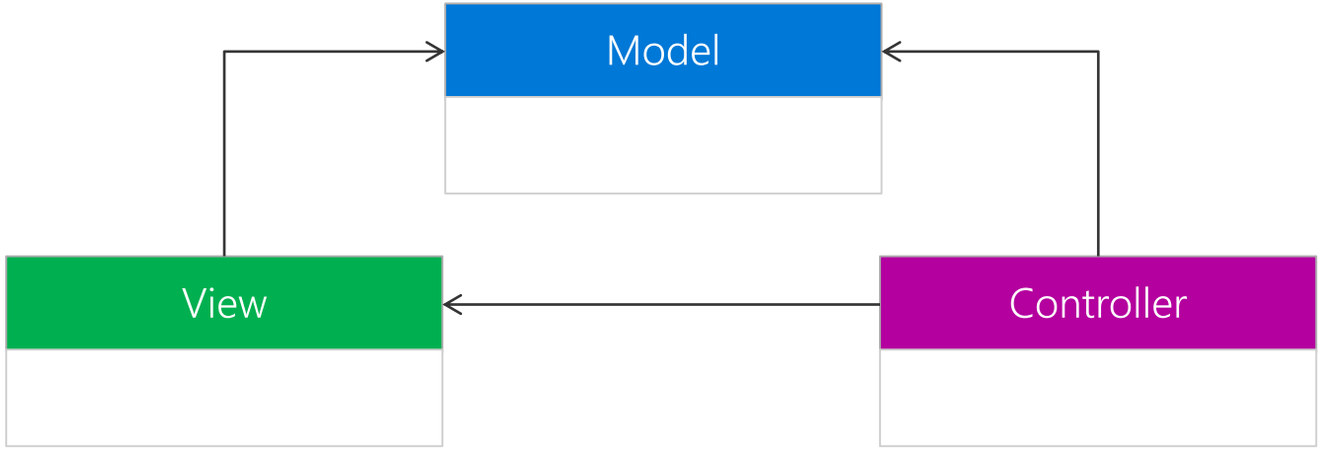



Model-View-Controller
(active model)



# MVC with a passive model

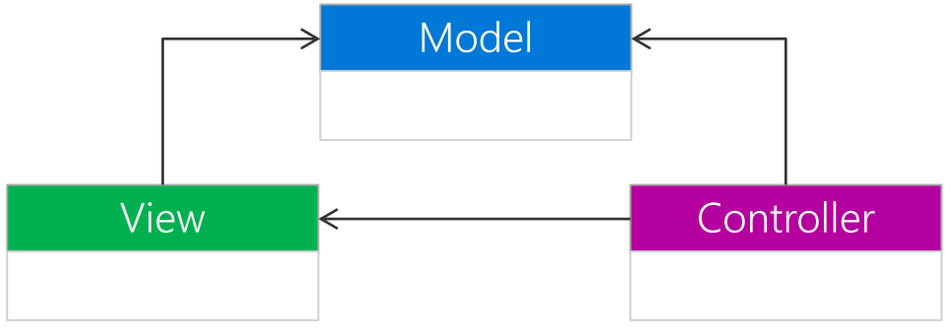
The MVC **passive model** style is where the model is manipulated only by the controller and never changes on it's own



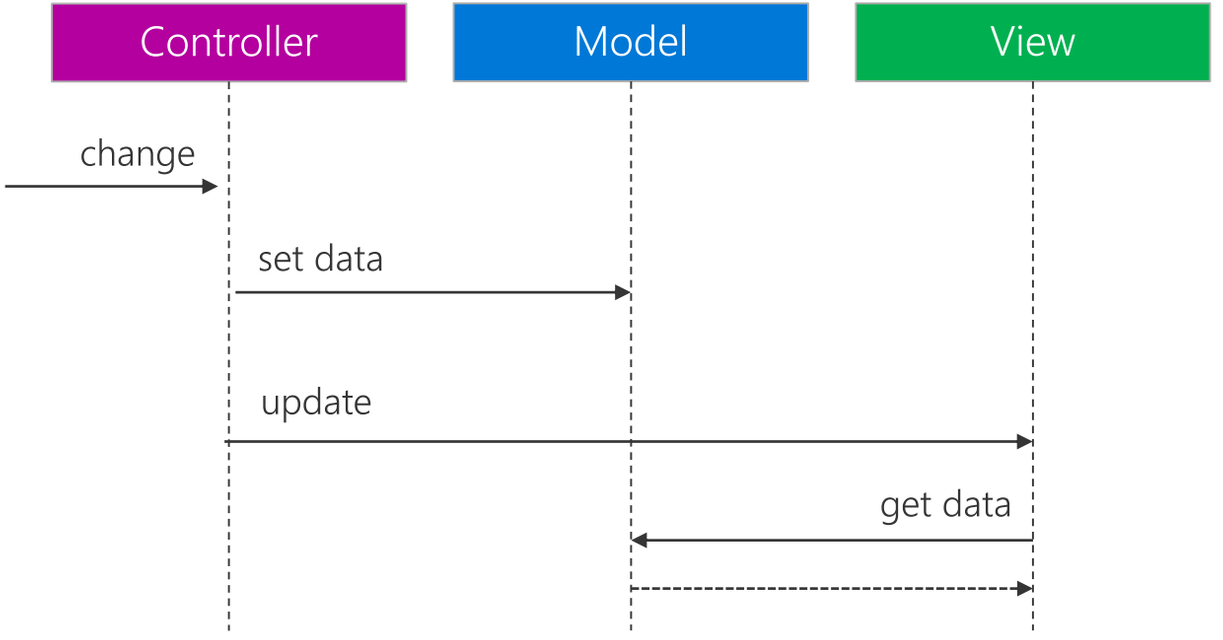

Model-View-Controller
(passive model)

# MVC with a passive model

The MVC **passive model** style is where the model is manipulated only by the controller and never changes on it's own




Model-View-Controller
(passive model)



# What is the Model?

The model represents the application data and domain logic

```
public class Employee
{
   public int Id { get; }
   public string Name { get; set; }
   public string Title { get; set; }
   public int Supervisor { get; set; }
   public DateTime HireDate { get; set; }
}
```

Data (often just simple DTOs)

```
public class PayrollManager
{
   public void GiveBonus(
         Employee emp,
         double amount)
   {
      ...
   }
```

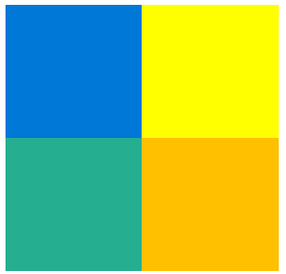
Logic (domain specific)

# What is the View?

View presents the information to the user in a **platform-specific** fashion - visual elements should be managed here

Click here

Controls



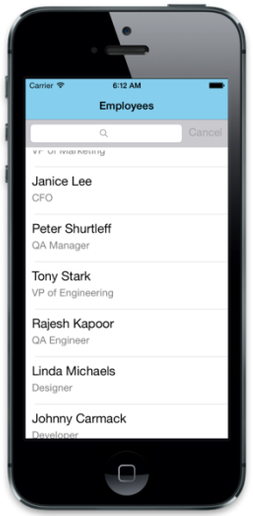
Colors / Themes

*Hello World!*

Fonts

# What is the Controller?

Controller defines the application behavior and maps user actions to the underlying data




UI activity

notify

get

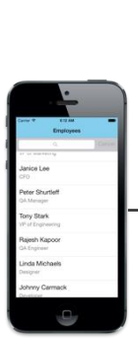
Controller

update

```
public class Employee
{
   public int Id { get; }
   public string Name { get; set; }
   public string Title { get; set; }
   public int Supervisor { get; set; }
   public DateTime HireDate { get; set; }
}
```

# Visual logic vs. Business Logic

Controllers often end up being the dumping ground for code - a common nickname is "Massive View Controller" – never forget that the place for all your domain logic (e.g. business logic) is actually the **model**



"tap"

Controller

determine what to do

call

Model

data

business logic

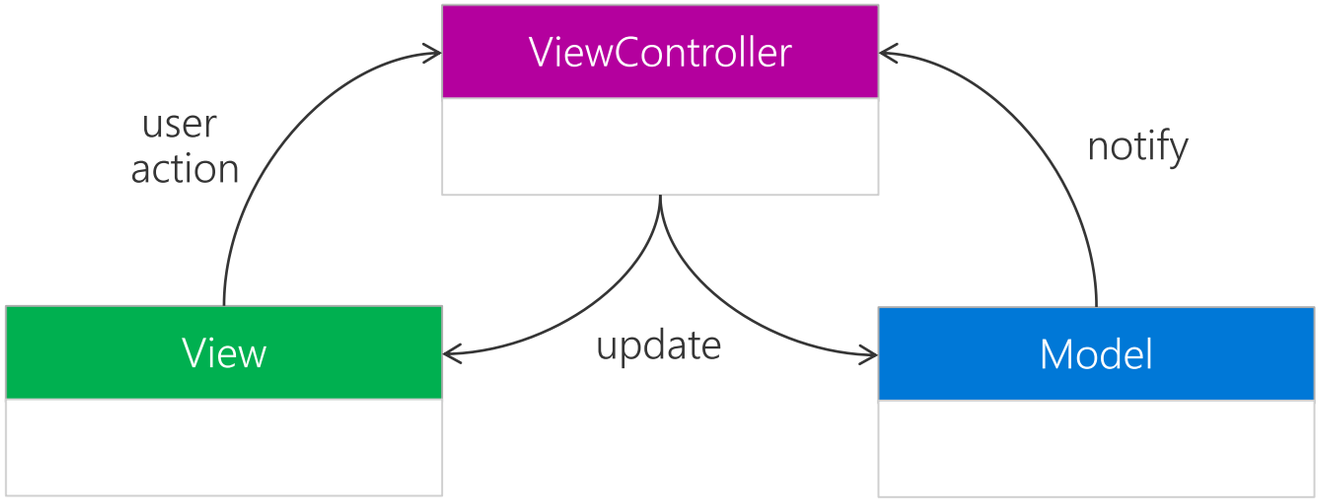
persistence

All of these things *belong* in this area because they are **part of the problem domain**

You should still apply separation principles with classes, namespaces, etc. to organize this efficiently

# MVC in the Apple world

UIKit utilizes MVC to decouple the model and view; each screen has a required parent View Controller where view interactions are handled



# Exercise

Apply MVC in an iOS application

# Pros and Cons of MVC

MVC is a tried and true pattern that supports the separation of concerns – particularly useful on platforms that encourage it

| Pros | Cons |
| --- | --- |
| Clean separation of roles in app which makes architecture more clear and allows for multiple developers to work on codebase | View and Controller tend to be closely coupled making independent changes more difficult to coordinate |
| Well known. Lots of frameworks out there which implement and provide support for the pattern | Increased complexity is not always suitable for smaller applications |
| Model classes and domain is testable and reusable with different platforms and UI technologies | Controller can become a dumping ground of code because it tends to be the easiest place to put logic |
| Can have multiple views of the same model (think Excel showing graph of numeric data) | Pattern can be complex to understand and apply |

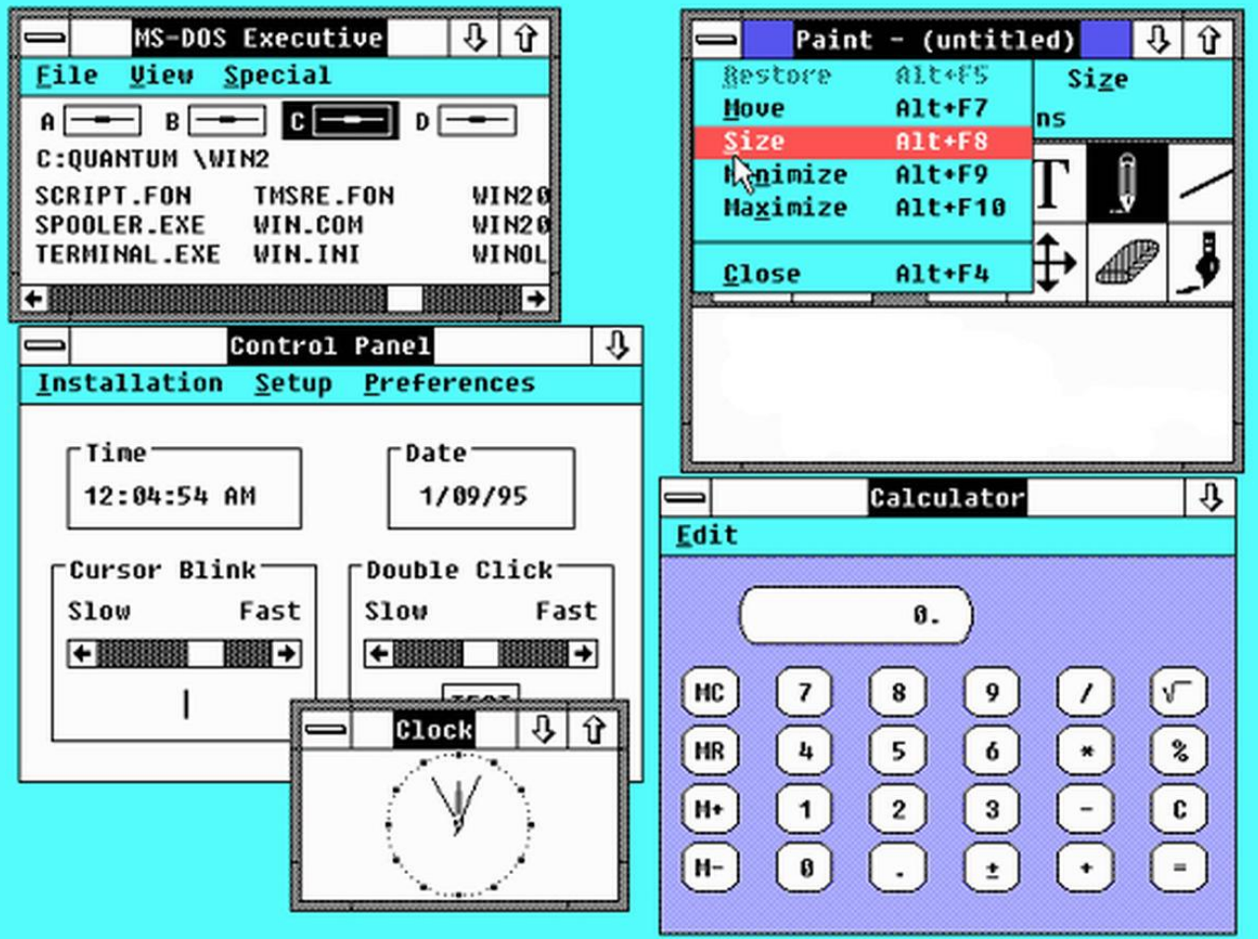
# Apply Model-View-Presenter to a Xamarin.Android app

# Tasks

1. Architect an Android application using MVP

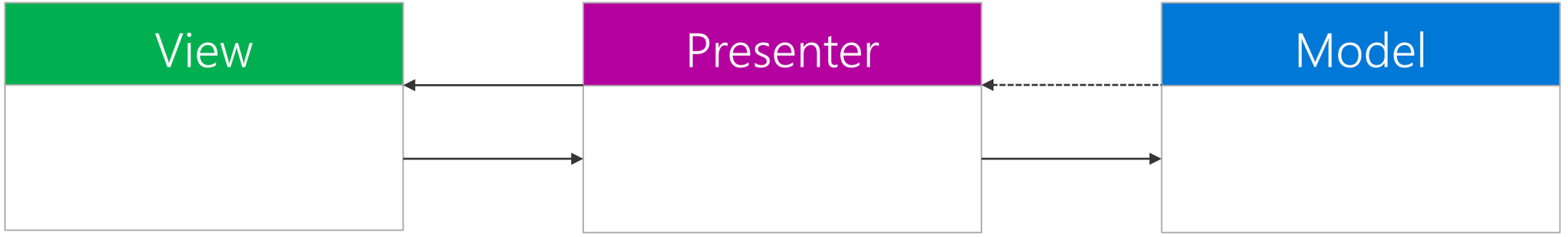
# Welcome to the Future

GUI programming changed completely with the introduction of a real control model that managed visualization and user interaction



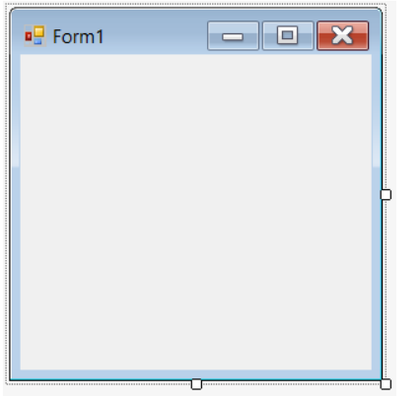
# Model-View-Presenter (MVP)

Model-View-Presenter is a separate presentation pattern where we include a *presenter* that works with views that manage their own behavior (e.g. Controls)



# MVP frameworks

Many popular frameworks are designed to use the Model-View-Presenter pattern to separate business logic from UI



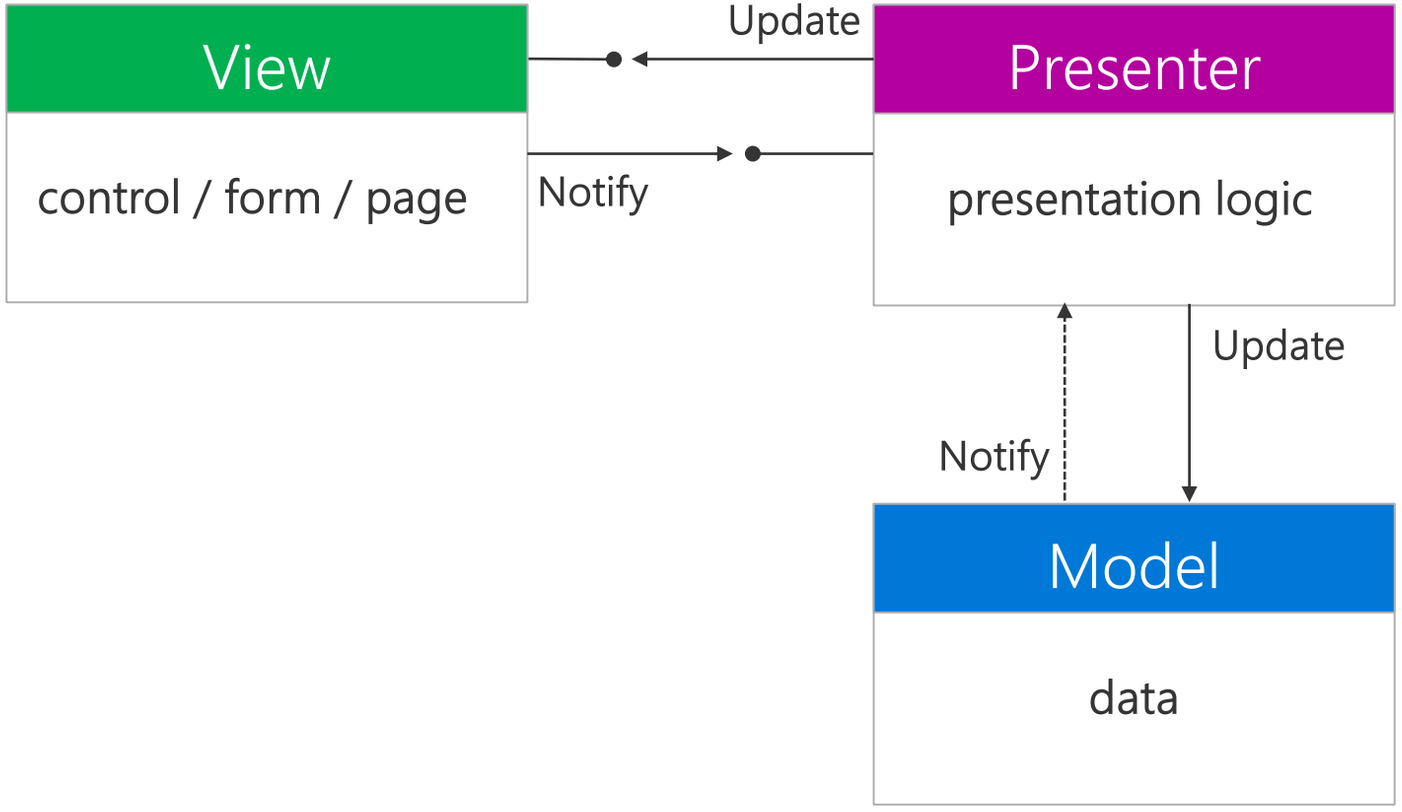

Windows Forms

Xamarin.Forms

Xamarin.Android

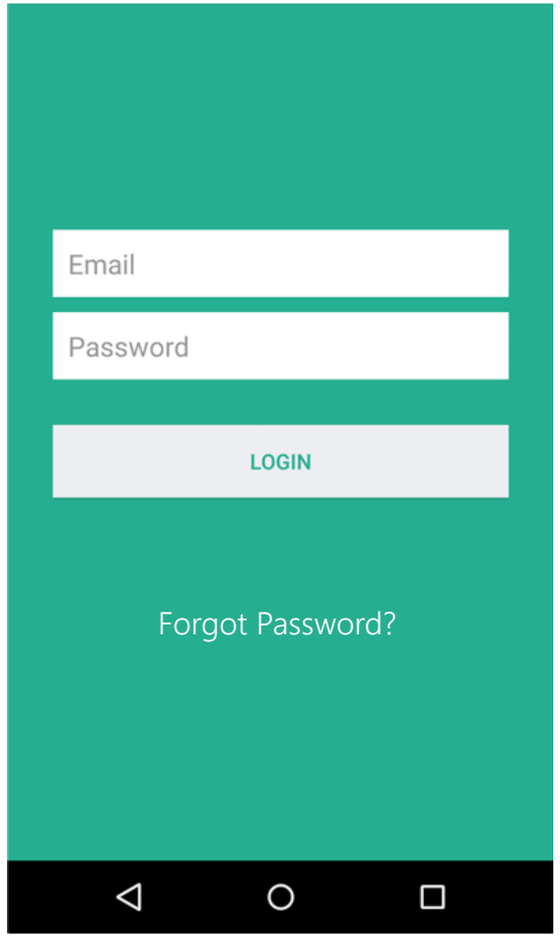
# MVP – Passive View

In **Passive View**, the interaction between the view and model is always done through the presenter; this makes the UI completely testable through the presenter



# Example: decoupling the view and presenter

Can use two related interfaces to keep view/presenter contracts synchronized



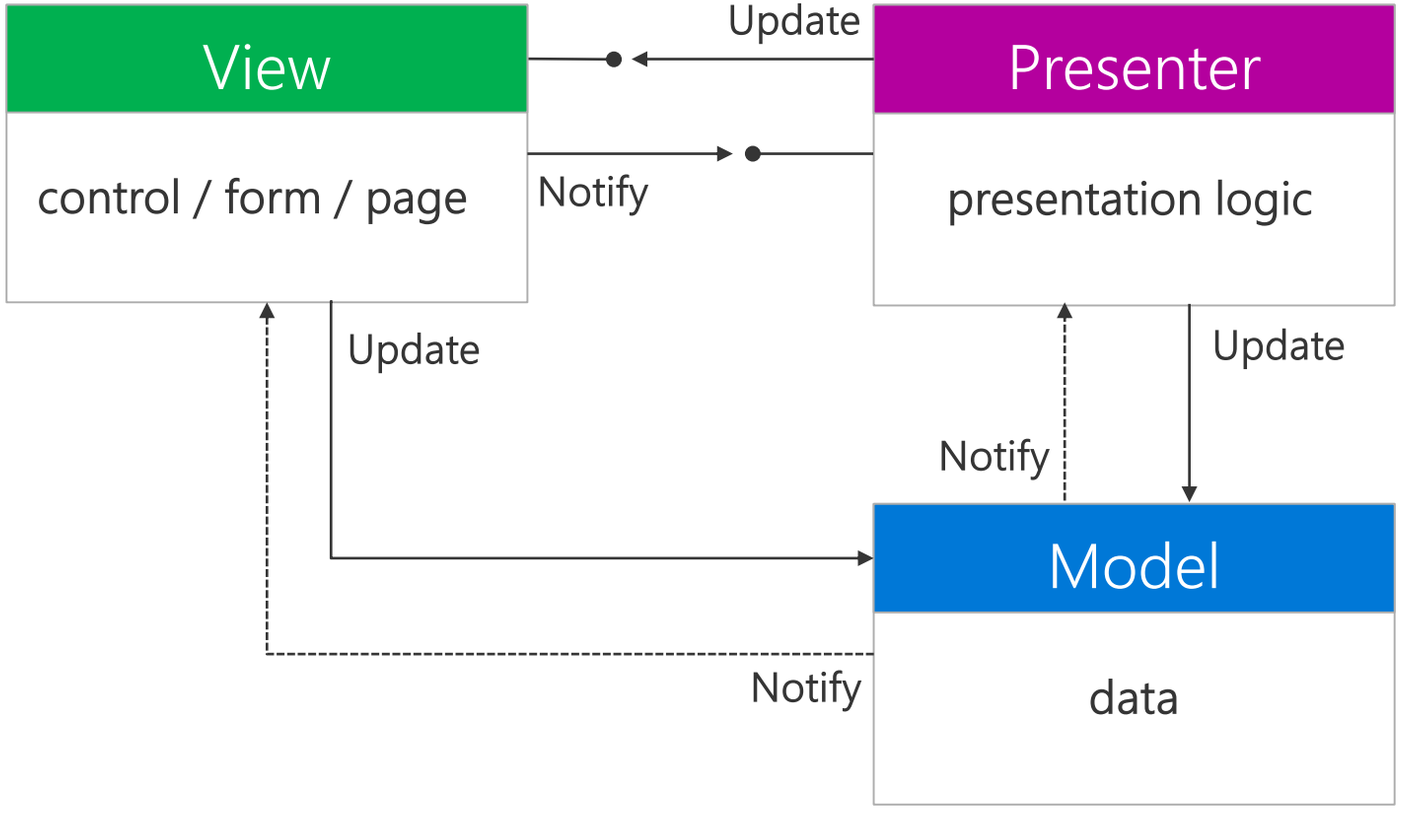

```
public static class LoginPage
{
    public interface IView // implemented by Activity
    {
        void ClearFields();
        void SetEmail(string email);
    }

    public interface IPresenter
    {
        void PerformLogin(IView view, string email,
                          string password);
        void ForgotPassword(IView view, string email);
    }
}
```

The key thing here is our presenter is not tied to the Activity!

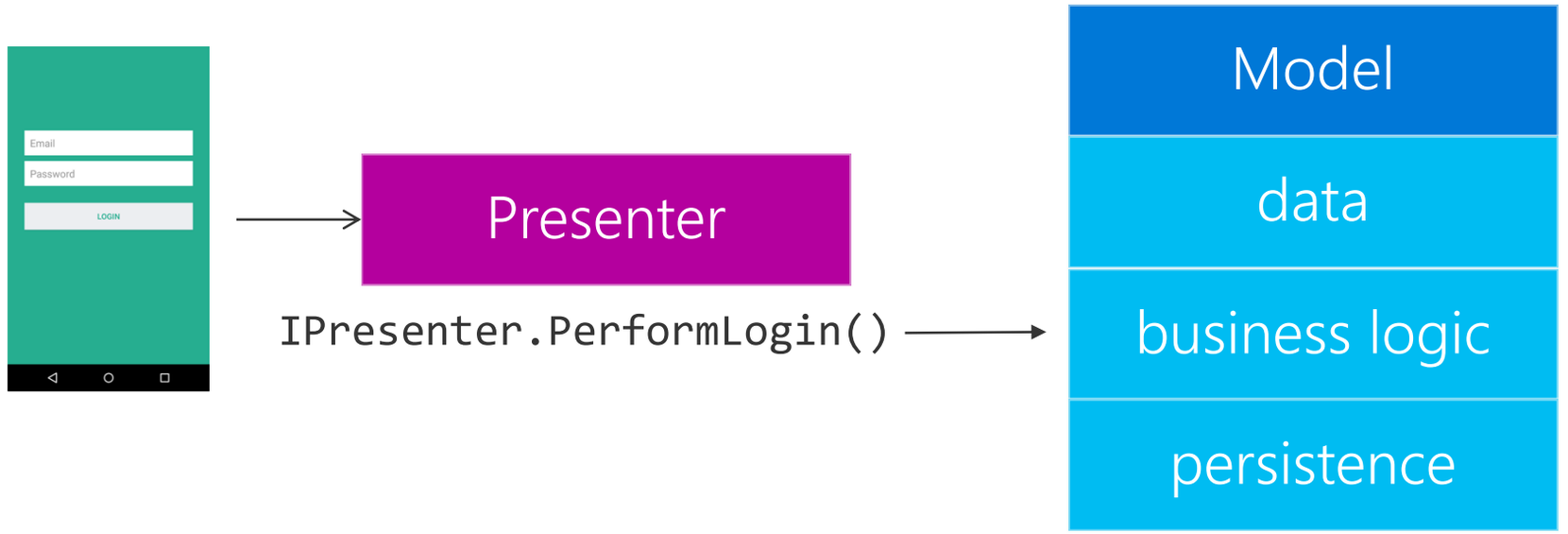
# MVP – Supervising Controller

In **Supervising Controller**, the view interacts directly with the model without the presenter being involved; the presenter updates the model and the model then pushes those changes directly to the view



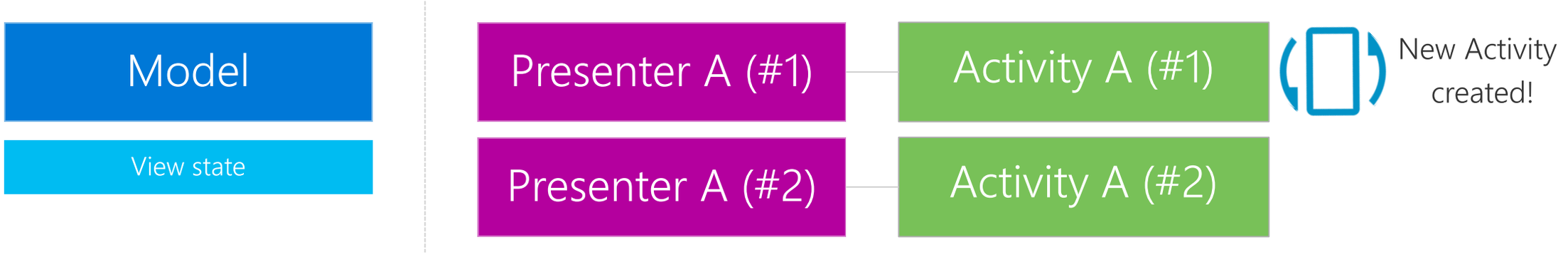
# Visual logic vs. Business Logic

Visual logic and behavior is placed in the Presenter, domain logic is placed in the model layer



# Presenter lifetime and persistence

Presenter should have a 1:1 relationship with the view and should move any necessary view state into the model with an in-memory cache



# Exercise

Apply MVP in an Android application

# Pros and Cons of MVP

MVP is a great technique for most modern UI frameworks that provides testability and a separation of the view and data driving it

| Pros | Cons |
| --- | --- |
| Can help promote better architecture in frameworks that do not encourage separation of view and data (e.g. Android) | Not all decisions are easy to make – should the presenter be persisted? How does it align with the lifecycle of the app? |
| Passive view allows for high testability at the expense of more code in the presenter | App needs to create and connect presenters to views |
| Supervising controller promotes code simplicity over full testability (but is still highly testable) | Model must provide some sort of change notification if it changes independent of the presenter |

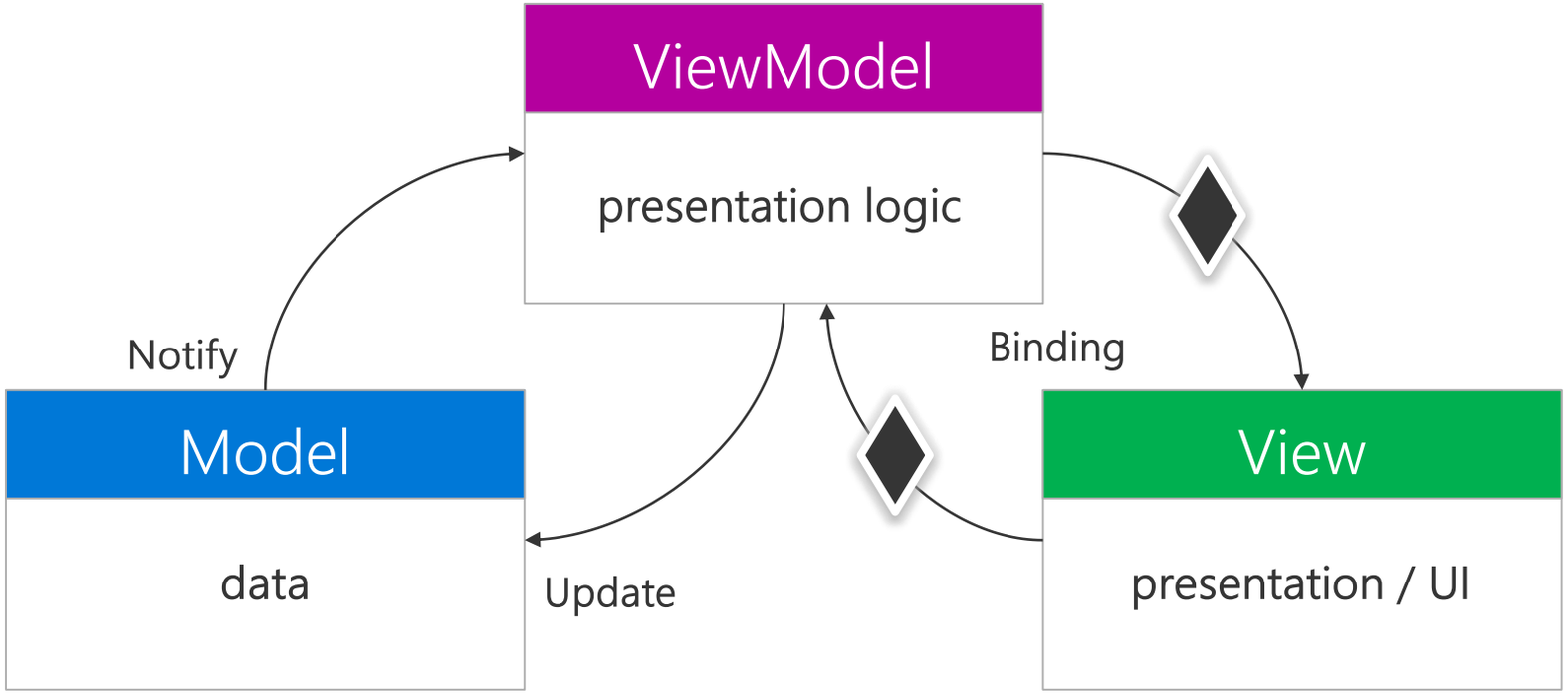
# Apply Model-View-ViewModel to a Xamarin.Forms app

# Tasks

1. Architect a Xamarin.Forms app using MVVM

# Model-View-ViewModel

MVVM is a variation of MVC which uses a data binding infrastructure to connect controllers (named "view models") to views




https://martinfowler.com/eaaDev/PresentationModel.html

# What is the ViewModel?

The ViewModel provides a **view-centric representation** of the data to display

exposes bindable properties and implements property change notification

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    private Employee model;

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

    public EmployeeViewModel(Employee model) {
        model = model;
    }
    ...
}
```

often has a 1:1 relationship with model

# Customize the ViewModel for the View

The ViewModel enables conversion and coercion of methods or model properties to allow the view to more easily display data

```
partial class EmployeeViewModel
{
    ...
    public string DateHiredText {
        get { return model.HireDate.ToString("MMM d, yyyy"); }
    }

    public EmployeeViewModel Supervisor
    {
        get
        {
            return new EmployeeViewModel(Employee.GetById(this.supervisor));
        }
    }
}
```

# Bindable Properties in the ViewModel

The ViewModel provides bindable properties to help the View access related data

```
partial class EmployeeViewModel
{
    ...
    public IEnumerable<string> ActiveProjects {
       get {
           return CompanyProjects.All
             .Where(p => p.Owner == model.Id
                    && p.IsActive)
             .Select(p => p.Name).ToList();
        }
    }
}
```

# Visual State and Logic

The ViewModel provides a place to put inconvenient logic for the UI – for example perform input validation prior to storing it in the model, or perform visual calculations or runtime status values for the UI

```
partial class DownloaderViewModel {
   private int percentComplete;
   public int PercentComplete {
      get { return percentComplete; }
      set {
         if (percentComplete != value) {
            percentComplete = value;
            OnPropertyChanged(nameof(PercentComplete));
         }
      }
   }
}
```

# Connecting a View and ViewModel

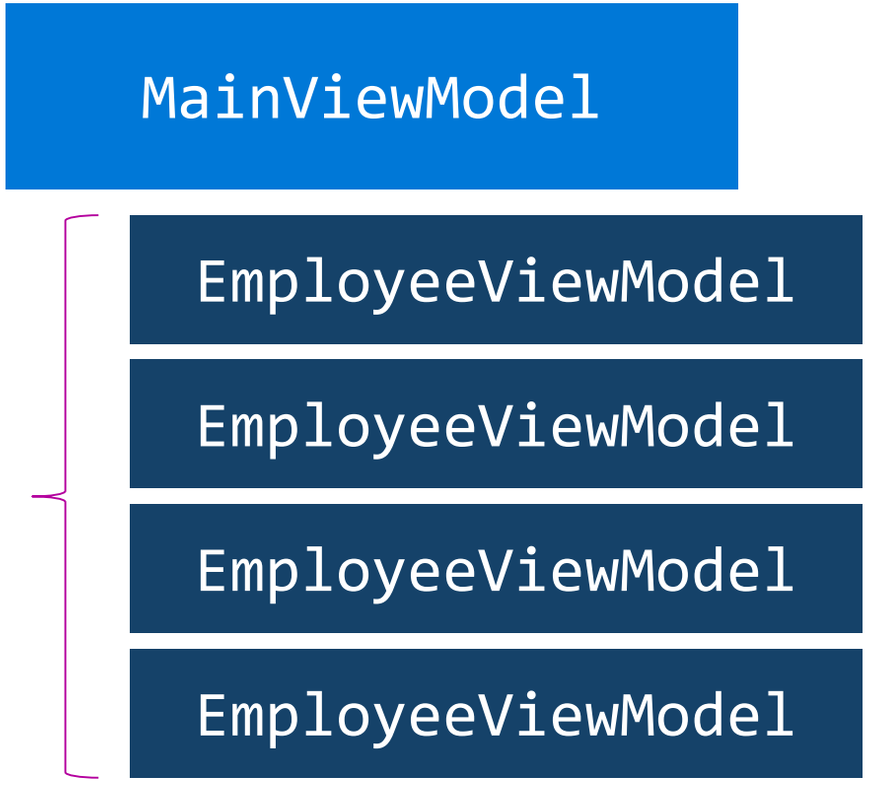
A ViewModel object is most often set as the **`BindingContext`** for the view– the view uses bindings to connect to properties exposed by the ViewModel

```
public partial class MainPage : ContentPage
{
    readonly MainViewModel viewModel = new MainViewModel();
    public MainPage ()
    {
        BindingContext = viewModel;
        InitializeComponent ();
    }
    ...
}
```

# ViewModel to View relationships

Apps often have multiple view models – one for each "data-bindable" entity being displayed

No rules on the mapping between ViewModels and Views, or between ViewModels and Models – it's typically 1:1 but can be adjusted for app requirements




`MainViewModel` might expose collection of `EmployeeViewModel` objects to bind to a `ListView`

# What goes in the ViewModel

ViewModel should not expose visual "things" such as colors and fonts; instead provide data that the view uses to *decide* these things

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


```
partial class EmployeeViewModel
{
    public bool IsSupervisor { get; }
    ...
}
```

Always push decisions like what color something is into the View, can do this with value converters, code behind, or triggers in XAML

# Exercise

Apply MVVM in a Xamarin.Forms application

# Become more proficient with MVVM

Xamarin University has a dedicated class for MVVM (**XAM320**) which provides more advice on how to handle specific problems in MVVM:

- How to deal with Selection
- How to deal with Actions
- More advice on how to structure ViewModels
- How to Unit Test View Models

XAM320

## Model-View-ViewModel in Xamarin.Forms

Apply the Model-View-ViewModel (MVVM) design pattern to your Xamarin.Forms apps to improve code reuse and testability.

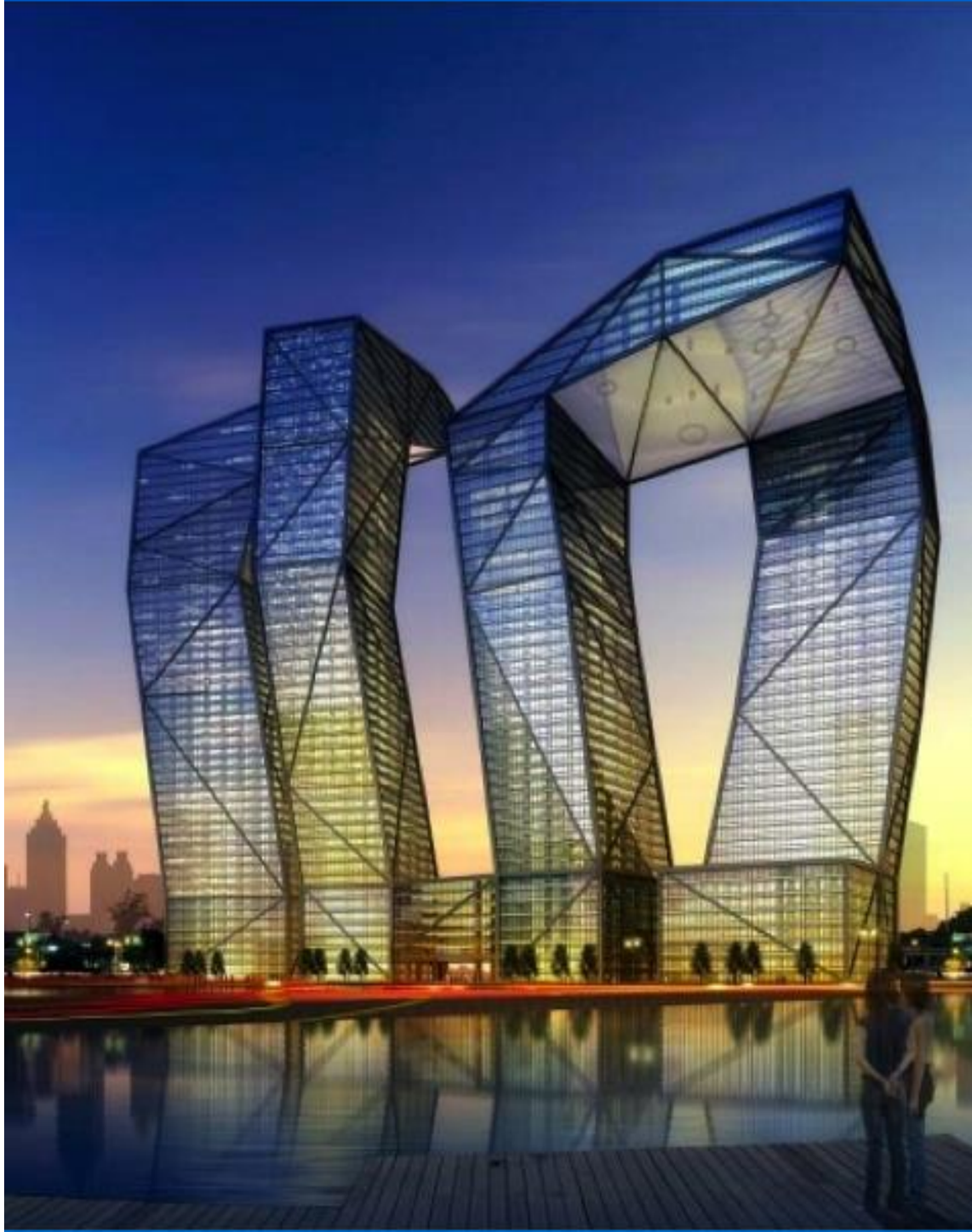
# Choosing an architectural style

All three styles are proven and provide good separation of concerns - choice is often a matter of preference and familiarity with a given pattern

| MVC / MVP | MVVM |
| --- | --- |
| Well-known pattern used in many platforms and frameworks – should lean towards this pattern when using native Android and iOS UIs | Introduced originally for WPF and carried over to all XAML-based frameworks; however becoming more popular for other UI platforms |
| Controller acts as an intermediary between the View and Model and executes business logic on behalf of both of them | ViewModel represents the "model" for the view (or the "view" of the model); can also provide execution logic with the Command pattern |
| Controller must include code to keep the Model and the View synchronized by monitoring both to watch for changes | Uses Bindings and the Observer pattern to keep the View in sync with the Model; can be difficult to debug bindings |

# Summary

1. Architect to maximize code sharing
2. Apply Model-View-Controller to a Xamarin.iOS app
3. Apply Model-View-Presenter to a Xamarin.Android app
4. Apply Model-View-ViewModel to a Xamarin.Forms app



Microsoft

# Thank You!

Please complete the class survey in your profile:

university.xamarin.com/profile