Lecture 8 – MVVM

Mr. Yousif Garabet Arshak
Computer Science Department
University of Zakho
yousif.arshak@uoz.edu.krd

Outlines

- **Instroduction**
- **The MVVM Pattern**
- **View**
- **ViewModel**
- **Model**
- **MVVM Example**

Introduction

- The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI).
- Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve.
- It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an app.

The MVVM Pattern

- There are three core components in the MVVM pattern: the model, the view, and the view model. Each serves a distinct purpose. Figure 2-1 shows the relationships between the three components.

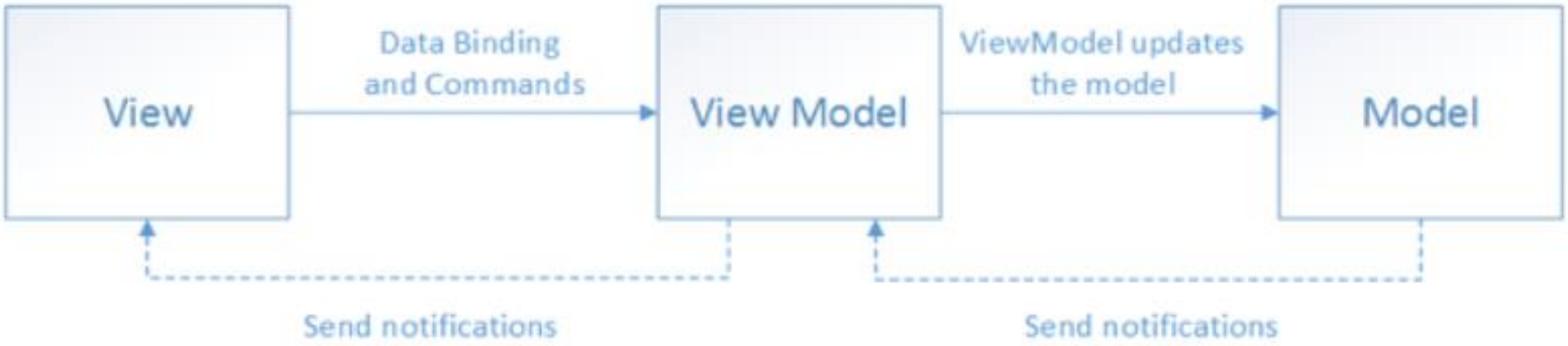


Figure 2-1: The MVVM pattern

The benefits of using the MVVM pattern are as follows:

- If there's an existing model implementation that encapsulates existing business logic, it can be difficult or risky to change it. In this scenario, the view model acts as an adapter for the model classes and enables you to avoid making any major changes to the model code.
- Developers can create unit tests for the view model and the model, without using the view. The unit tests for the view model can exercise exactly the same functionality as used by the view.
- The app UI can be redesigned without touching the code, provided that the view is implemented entirely in XAML. Therefore, a new version of the view should work with the existing view model.
- Designers and developers can work independently and concurrently on their components during the development process. Designers can focus on the view, while developers can work on the view model and model components.

View

- The view is responsible for defining the structure, layout, and appearance of what the user sees on screen. Ideally, each view is defined in XAML, with a limited code-behind that does not contain business logic.
- However, in some cases, the code-behind might contain UI logic that implements visual behavior that is difficult to express in XAML, such as animations.

ViewModel

- The view model implements properties and commands to which the view can data bind to, and notifies the view of any state changes through change notification events.
- The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be displayed.

Tip
Keep the UI responsive with asynchronous operations. Mobile apps should keep the UI thread unblocked to improve the user's perception of performance. Therefore, in the view model, use asynchronous methods for I/O operations and raise events to asynchronously notify views of property changes.

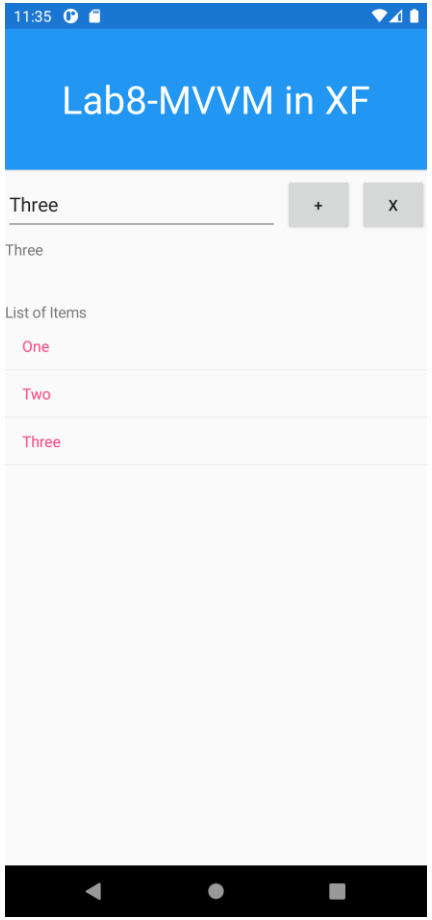
- In order for the view model to participate in two-way data binding with the view, its properties must raise the *PropertyChanged* event. View models satisfy this requirement by implementing the *INotifyPropertyChanged* interface, and raising the *PropertyChanged* event when a property is changed.

- For collections, the view-friendly *ObservableCollection*<T> is provided. This collection implements collection changed notification, relieving the developer from having to implement the *INotifyCollectionChanged* interface on collections.

Model

- Model classes are non-visual classes that encapsulate the app's data. Therefore, the model can be thought of as representing the app's domain model, which usually includes a data model along with business and validation logic.
- Model classes are typically used in conjunction with services or repositories that encapsulate data access and caching.

MVVM Example



Any Questions?