# Bonus Exercise 2: MVC, MVP and MVVM in Kotlin

Student Amirhossein Zeinali Dehaghani (std ID: 1225496) (std ID: 1225496)
Professor Dr. Sabah Mohammed
Lakehead University

## Contents



**Click here to watch the screen recording of application**

# 1 Introduction

In software development, Model-View-Controller (MVC), Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM) are the three architectural patterns which are used frequently.

## 1.1 MVC (Model-View-Controller)

MVC is a design pattern that separates the application into three components. The Model manages the data and business logic, the View is responsible for the user interface, and the Controller handles user input and updates the Model and View.

## 1.2 MVP (Model-View-Presenter)

MVP is a derivative of MVC that emphasizes a more decoupled approach. In MVP, the Presenter replaces the Controller. It handles user interactions by processing inputs from the View, updating the Model, and pushing data back to the View, making the View passive and focused on display logic.

## 1.3 MVVM (Model-View-ViewModel)

MVVM is another architectural pattern designed primarily for data-binding frameworks. It allows for a more reactive approach where the ViewModel holds the presentation logic and transforms data from the Model into a format that the View can easily consume. MVVM eliminates the need for a Controller or Presenter, as the View binds directly to the ViewModel's properties, which updates the UI when the data changes automatically.

# 2 Overview of the Application

The application designed as a simple counter-app that demonstrates the implementation of MVC, MVP, and MVVM architectural patterns. This application mainly intended to highlight the differences and similarities of these architectural styles rather than focus on complex functionality.

# 3 MainActivity Implementation

## 3.1 MainActivity.kt

The `MainActivity` class serves as the entry point for the application, allowing users to access to different architectural patterns implemented in the app. It uses buttons to transition between the MVC, MVP, and MVVM activities.

Key components include:

- **Button Initialization:** In the `onCreate()` method, three buttons are initialized corresponding to each architectural pattern (MVC, MVP, MVVM). Each button is linked to a different activity.
- **Starting Activities:** On clicking each button, an `Intent` is created to start the respective activity:
  - `mvcButton`: Starts `MvcActivity` when clicked.
  - `mvpButton`: Starts `MvpActivity` when clicked.
  - `mvvmButton`: Starts `MvvmActivity` when clicked.
- **User Interaction:** This activity serves as the main interface for users to choose which architectural pattern they would like to explore.

The `MainActivity` acts as a central hub to navigate between the different implementations.

### 3.2 activity_main.xml

The file defines the user interface layout for the `MainActivity` class.

- **LinearLayout:** The root element is a vertical `LinearLayout` that centers its content.
- **ImageView (logo_image):** Displays the application logo (Lakehead logo).
- **TextView (name_label):** This `TextView` shows my name as developer(Amirhossein Zeinali Dehaghani).
- **TextView (id_label):** Another `TextView` displays my student ID.
- **Button (mvc_button):** This button is labeled for the MVC example.
- **Button (mvp_button):** This button is labeled for the MVP example.
- **Button (mvvm_button):** This button is labeled for the MVVM example.

### 3.3 MainActivity output (Screenshot)


Figure 1: Screanshot of MainActivity output

## 4 MVC Implementation

### 4.1 CounterModel.kt

The `CounterModel` class represents the Model in the MVC architecture.

- **increment():** Increments the counter value by one.
- **getCount():** Returns the current value of the counter.

### 4.2 MvcActivity.kt

The `MvcActivity` class acts as the View in the MVC pattern. It is responsible for managing the components of the user interface and user interactions.

Key points include:

- Initialization of UI components such as `TextView` for displaying the counter value and a `button` to increment the counter.
- The class creates an instance of the `CounterModel` to store the counter data and the `CounterController` to handle user actions.
- A click listener is set on the increment button, which calls the `incrementCounter()` method of the controller when clicked.
- The `updateView()` method updates the displayed counter value whenever it is called.

## 4.3 CounterController.kt

The `CounterController` class serves as the Controller in the MVC architecture. It is responsible for managing the interaction between the Model and the View. The **incrementCounter():** method increments the counter in the model and invokes the `updateView()` method of activity (view) to refresh the user interface.

## 4.4 activity_mvc.xml

The file defines the user interface layout for the `MvcActivity` class.

- **LinearLayout:** The root element is a vertical `LinearLayout` that centers its content.
- **TextView (name_label):** This `TextView` displays a welcome message.
- **ImageView (logo_image):** This `ImageView` shows MVC architecture.
- **TextView (counter_label):** Another `TextView` is used to display the current counter value.
- **Button (increment_button):** Finally, there is a `Button` labeled "Increment." This button is clickable and triggers the logic to increase the counter when pressed.

## 4.5 MVC output (Screenshot)

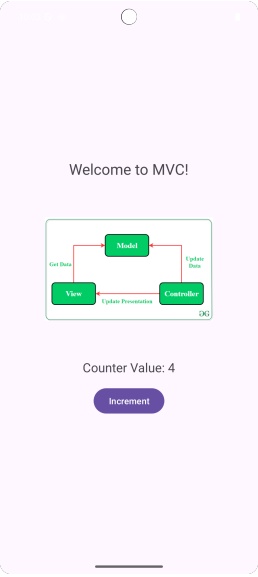


Figure 2: Screanshot of MVC output

# 5 MVP Implementation

## 5.1 CounterModel.kt

The `CounterModel` class encapsulates the counter data and logic. It contains a private variable `count` to hold the counter value, which is initialized to zero.

The class provides two methods:

- **increment():** Increments the counter value by one.
- **getCount():** Returns the current value of the counter.

## 5.2 CounterPresenter.kt

The `CounterPresenter` class is responsible for managing the interaction between the model and the view. It takes a `CounterModel` and a `CounterView` as parameters.

The presenter has two main methods:

- **incrementCounter():** This method calls the `increment()` method of the model and then updates the view by calling `updateView()` with the new counter value.
- **getCounterValue():** Retrieves the current counter value from the model.

## 5.3 CounterView.kt

The `CounterView` interface defines the methods required for the view to update itself.The **updateView** method must be implemented by any view that needs to display the updated counter value.

## 5.4 MvpActivity.kt

The `MvpActivity` class acts as the View in the MVP pattern. Implements the `CounterView` interface and manages the components of the user interface.

Key points include:

- Initialization of the `CounterPresenter` with a new instance of `CounterModel` and the activity as the view.
- Setting a click listener on the increment button that invokes the `incrementCounter()` method in the presenter when clicked.
- The `updateView()` method updates the `TextView` displaying the counter value when called.

## 5.5 activity_mvp.xml

The file defines the user interface layout for the `MvcActivity` class.

- **LinearLayout:** The root element is a vertical `LinearLayout` that centers.
- **TextView (name_label):** This `TextView` displays a welcome message.
- **ImageView (logo_image):** This `ImageView` displays MVC architecture.
- **TextView (mvp_counter_label):** Another `TextView` is used to display the current counter value.
- **Button (mvp_increment_button):** Finally, there is a `Button` labeled "Increment." This button is clickable and triggers the logic to increase the counter when pressed.

### 5.6 MVP output (Screenshot)

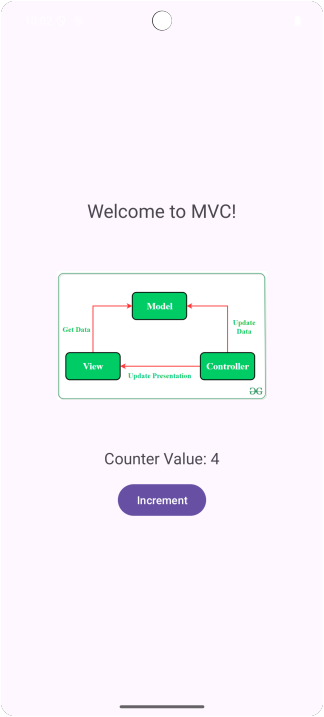


Figure 3: Screanshot of MVP output

## 6 MVVM Implementation

### 6.1 CounterModel.kt

The `CounterModel` class represents the Model in the MVVM architecture. It encapsulates the counter data using a private variable `count`, which is initialized to zero. This class provides two essential methods:

- **increment():** Increments the counter value by one.
- **getCount():** Returns the current value of the counter.

### 6.2 CounterViewModel.kt

The `CounterViewModel` class functions as the ViewModel in the MVVM pattern. It holds the logic for the user interface and interacts with the Model.

Its key features include:

- **_counterValue:** A `MutableLiveData` object that holds the counter value, allowing it to change over time.
- **counterText:** A `LiveData<String>` that formats the counter value as a string for display in the UI.
- **onIncrementButtonClicked():** This method increments the counter value in the model and updates the `LiveData` with the new value.

### 6.3 MvvmActivity.kt

The `MvvmActivity` class acts as the View in the MVVM architecture. It is responsible for managing the user interface and user interactions.

Key aspects include:

- Initialization of data binding using `DataBindingUtil` to connect UI components with the ViewModel.

- Creation of the `CounterViewModel` instance using a `ViewModelProvider` and a custom factory to manage the lifecycle of the ViewModel.
- Binding the ViewModel to the activity layout and setting the lifecycle owner to observe changes in `LiveData`.

### 6.4 activity_mvvm.xml

The file defines the user interface layout for the `MvvmActivity` class.

- The root element is defined as a `layout` to enable data binding.
- The `LinearLayout` arranges its child views in a single column (vertical orientation).
- **TextView (name_label):** Displays a welcome message.
- **ImageView (logo_image):** Shows MVVM architecture.
- **TextView (counter_label):** Binds to the `counterText` property of the ViewModel, reflecting the current counter value.
- **Button (increment_button):** Calls the `onIncrementButtonClicked()` method in the ViewModel when clicked, handling the counter increment logic.

### 6.5 MVVM output (Screenshot)

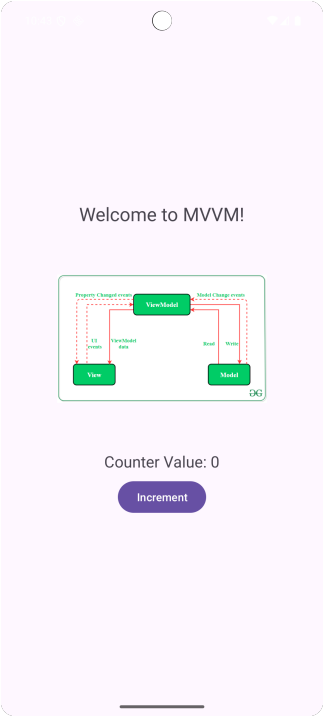


Figure 4: Screanshot of MVVM output

## 7 Conclusion

The application serves as a practical example of these three architectural patterns: MVC, MVP, and MVVM. Each pattern was analyzed through its components, including the Model, View, and Controller or Presenter, demonstrating how they interact to manage user input and update the user interface.

In conclusion, the application illustrates how leveraging the strengths of MVC, MVP, and MVVM can lead to better structured and more efficient software development.