FIT3077 Assignment 3 Design Rationale

Written by Team Ouroboros

MVP Architectural Pattern

The chosen architectural pattern still follows the MVP (Model-View-Presenter) pattern. As a recap, the MVP design is similar to the MVC and MVVM patterns, in the sense that MVP focuses on clearly defining and dividing responsibility of classes [1], as well as supports extensibility of the system.

As a breakdown of MVP layers, Model is the data layer which manages the business model classes, while View is the user interface (UI) layer which displays data as instructed by the Presenter, which is the logic layer that acts as a bridge between Model and View [2]. In terms of objects used in the Android app codebase, the Model refers to classes in the 'model' package, which is a collection of containers for data storage. The View components are classes that extend the Fragment class, which serves as the layout and UI, and the Presenter are classes that extend respective Adapter subclasses.

As the MVP architecture isolates functionalities of these components, the Presenter is the only layer which handles changes in Model and View, such that it intercepts the actions from the UI layer, updates the data and tells the UI layer to update itself [1].

This architectural pattern is chosen due to its ability to modularize components, supporting Single Responsibility Principle (SRP), as well as providing ease of extending the system with new features without the need of changing the structure and relationships of existing modules, which is the essence of Open/Closed Principle OCP.

In addition, the MVP architecture enforces one-to-one mapping of Presenter to View, which means one Presenter class handles all the listening and updating for one View [2]. This is the main mechanism that prevents god classes and supports extensibility, as new features usually add new views to the system. Overall, the MVP architecture strictly adheres to SRP and OCP, which creates a structured and modularized design without being overly complicated to implement into OO systems.

Advantages of MVP

The main advantage of MVP is isolation of components between Model, View, and Presenter, as well as limiting dependency of classes within the same layer. This is supported by the fourth component of MVP: the View Interface layer; this layer is responsible for loose coupling between View and Model [3]. Therefore, MVP has a comparatively high degree of loose coupling compared to other architectural patterns such as MVC.

For example, in Android, the Fragment base class is the View Interface where each specific Fragment (View) is a subclass that extends Fragment. Then the Adapters (Presenter) connects this interface with the Model to perform updates from data to UI and vice versa. Essentially, this creates an adaptable

design where changes are isolated and new extensions are free to choose a number of views and data when implementing new features [3].

Furthermore, together with one-to-one mapping of Presenter and View for each specific feature, the MVP model allows existing code to be reusable for new extensions without the need of design level refactoring, since components are loosely coupled. Finally, the unique benefit of MVP is the ability to test each component in isolation, since unit testing is considered complex for Android software [3].

Disadvantages of MVP

One of the biggest drawbacks of the MVP pattern is the significant increase in complexity of the system when many new extensions are added [3]. Due to the high degree of loose coupling and modularity of components, one View and its corresponding Presenter are created when one new UI component is implemented.

For example, each card view, layout, and other UI elements in the Android app do not share the same View (Fragment) or Presenter (Adapter) with another UI element; if a new layout or page is implemented, one new Fragment and Adapter will be created. Hence, although the Fragment and Adapter codes are reusable for similar features, there will be a large number of different classes to organize and manage as the system grows with new features which inevitably implement UI elements.

Package Principles

Refactoring