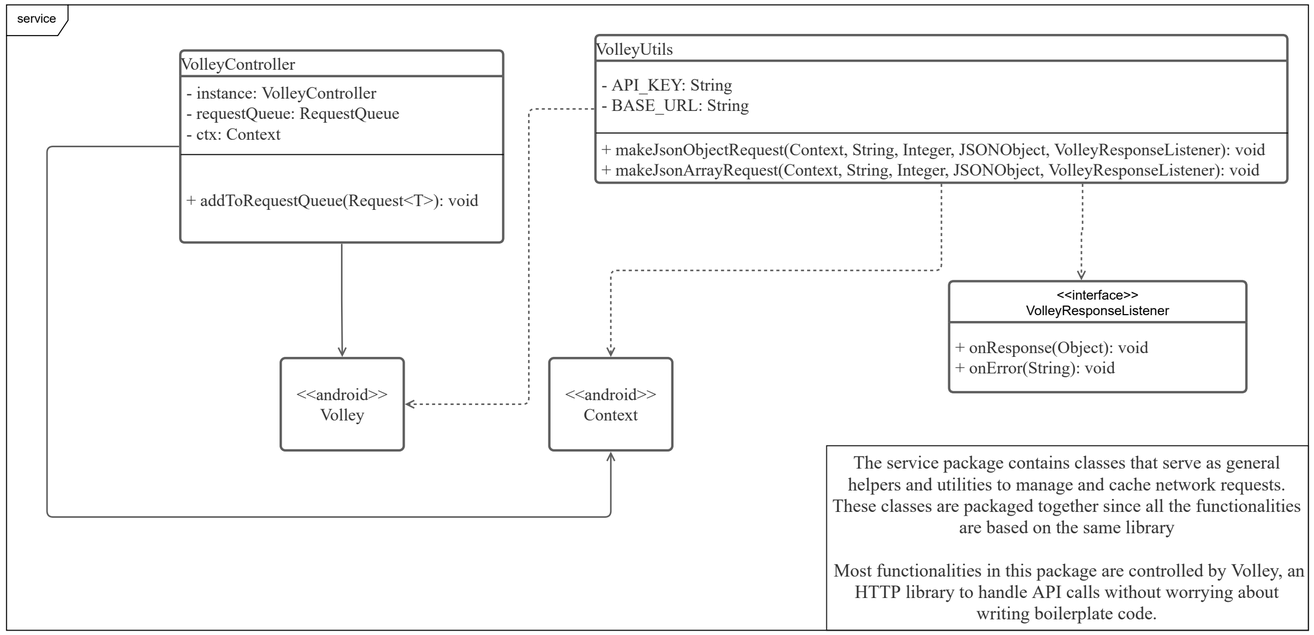


FIT3077 Assignment 3
Team Ouroboros

This first page shows the individual packages for model and service respectively, since all the packages and components do not fit in this page.

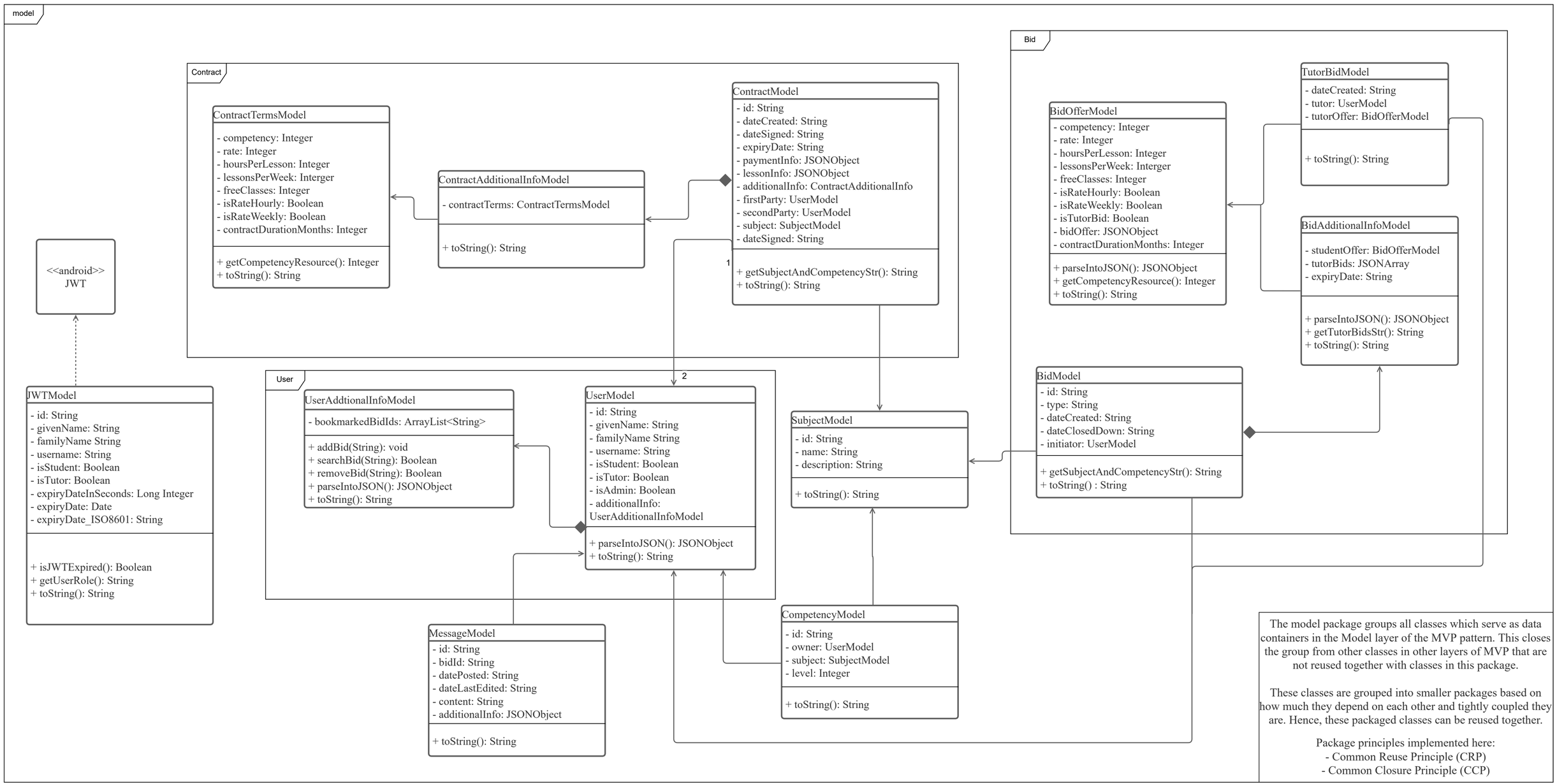
All the diagrams are migrated from LucidChart, hence all shapes still follows the previous style.

General Note:
Only important methods are included for each class (e.g. general getters and setters are not included).

New Package: User

New Model classes:
- CompetencyModel
- UserAdditionalInfo

Modified Model classes:
- BidOfferModel:
  - new methods for dropdown to index contract duration
  - new field: contractDurationMonths
- ContractModel
  - new field: dateSigned
  - modified method: getExpiryDateObj
- ContractTermsModel
  - new field: contractDurationMonths
- UserModel
  - new fields:
    - UserAdditionalInfoModel
    - isAdmin



The model package groups all classes which serve as data containers in the Model layer of the MVP pattern. This closes the group from other classes in other layers of MVP that are not reused together with classes in this package.

These classes are grouped into smaller packages based on how much they depend on each other and tightly coupled they are. Hence, these packaged classes can be reused together.

Package principles implemented here:
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)

The interaction of Model, View, and Presenter for the following features are demonstrated:

- Contract expiration and renewal

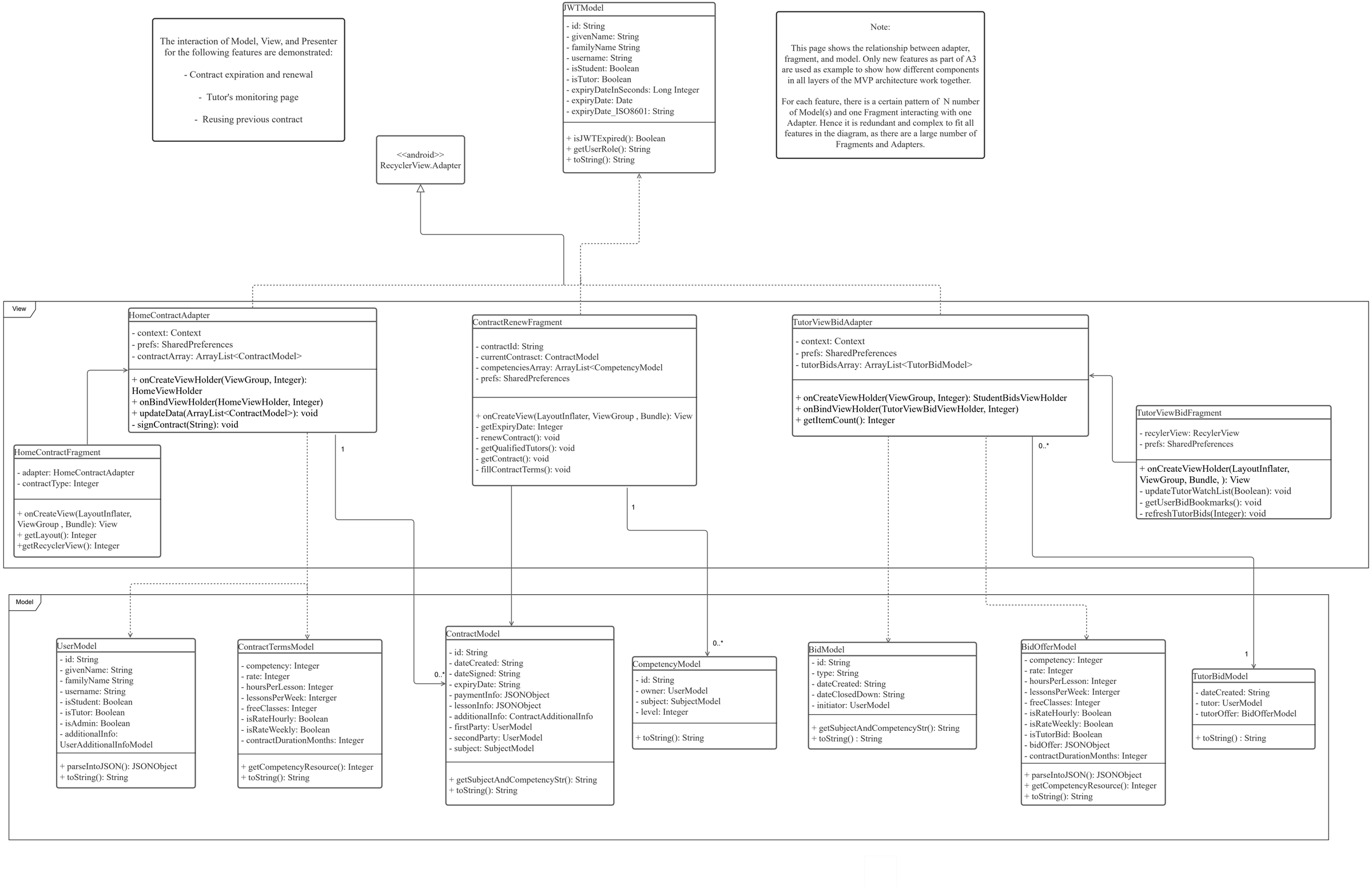
- Tutor's monitoring page

- Reusing previous contract

Note:

This page shows the relationship between adapter, fragment, and model. Only new features as part of A3 are used as example to show how different components in all layers of the MVP architecture work together.

For each feature, there is a certain pattern of N number of Model(s) and one Fragment interacting with one Adapter. Hence it is redundant and complex to fit all features in the diagram, as there are a large number of Fragments and Adapters.



FIT3077 Assignment 3 Design Rationale

Written by Team Ouroboros

MVP Architectural Pattern and Design Principles

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 theoretically complex to implement in 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

The package principles used for this system are Common Reuse Principle (CRP) and Common Closure Principle (CCP). Although these principles seem counterintuitive, each of these package cohesion principles is implemented separately to create a balanced architecture.

For example, each layer of MVP is divided into their own packages while adhering to CCP. The data containers in the Model layer are packaged together to maintain SRP and OCP on a component level. On the other hand, View (fragment) and Presenter (adapter) are packaged together since they are loosely coupled. If there are changes to the UI implementation, the View Interface and Presenter will have to adapt to the changes. However, each View is only conceptually packaged with its corresponding Presenter, as the one-to-one mapping property maintains overall SRP and OCP.

Although the classes in the Model layer are packaged together, there are subgroups of classes that are very tightly coupled. For example, all classes with ‘Bid’ on their names depend on each other, as a slight change in the implementation of a ‘Bid’ class will affect the other bid classes too. This is where CRP is applied, all the tightly coupled classes within the model package are packaged together, such as ‘Contract’,’ User’, and ‘Bid’ packages.

Refactoring

The improvement to existing code and new extensions for the application does not require architectural nor design level refactoring. The initial adoption of the MVP pattern was already established; each component, as well as their corresponding attributes strictly follows the layering structure of Model, View, and Presenter.

Therefore, the extension of new features with their UI elements, and the minor extension to existing data models does not disrupt the functionalities of the existing system due to proper implementation of the MVP pattern. In addition, the solid foundations of SRP and OCP embedded in the initial design following the MVP architecture omits the need for design-level refactoring.

Despite the benefits reaped from the modular and loosely coupled MVP architecture, there are some isolated code-level refactoring performed to increase readability and reduce overall clutter within a class. Most notably data organisation refactoring techniques are applied to some Model components. For example, private fields are self-encapsulated by implementing respective getters and setters to prevent indirect access. Furthermore, some aggregations are appropriately mapped into associations by changing value to reference, which allows keeping track of multiple Model instances.

References

[1] Elvedin, S., (2020). The practical guide: Refactor android application to follow the MVP design pattern. Retrieved from https://www.north-47.com/knowledge-base/the-practical-guide-refactor-android-application-to-follow-the-mvp-design-pattern/

[2] Chugh, A., (2019). Android MVP. Retrieved from https://www.journaldev.com/14886/android-mvp

[3] Sony, N., (2013). MVC v/s MVP - How Common and How Different. Retrieved from https://www.codeproject.com/Tips/31292/MVC-v-s-MVP-How-Common-and-How-Different