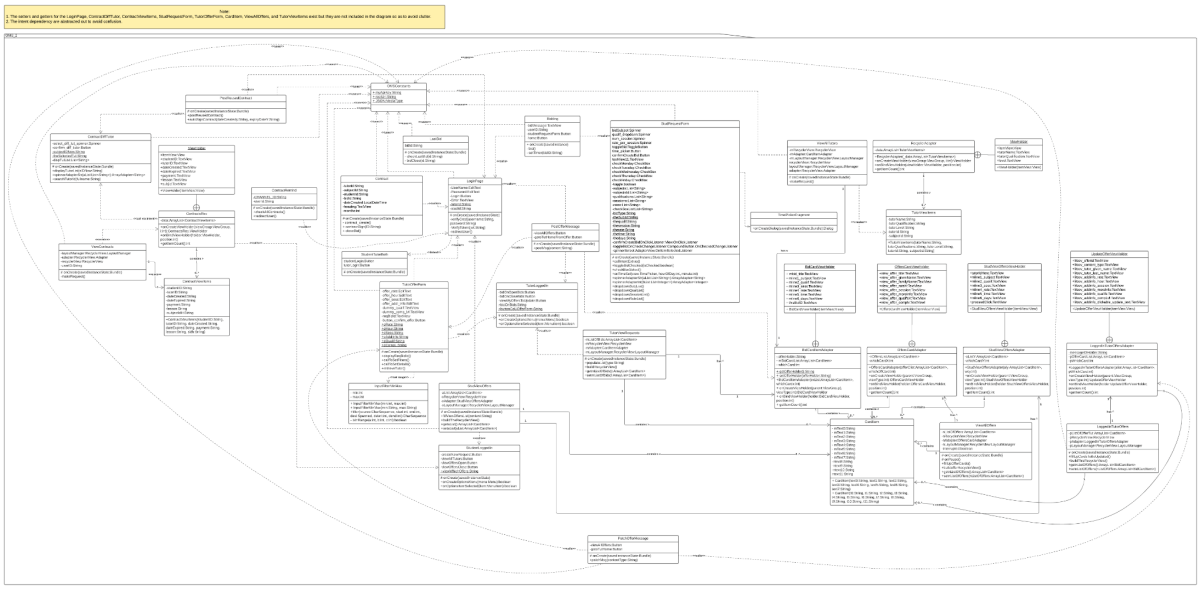
Class Diagram



Class Diagram also added separately for clarity

Design Rationale

While implementing Assignment 3 we added the following new Classes reusing previous design principles.

Design Principle - Single Responsibility Principle

The new Classes added are using the principle again in assignment 3. For example to implement the contract renewal and alert we created separate classes to implement each functionality of these. For viewing all contracts we created 3 classes that are being used to create a Recycler View of the signed contracts.

Examples of this can also be seen in using the details from the previous contracts. There are separate classes for signing a new contract with the same tutor and separate classes for signing with another tutor. We tried our best to follow the single responsibility system as this made it quite easier for us to work on the system without worrying about understanding each other's code and just asking for the relevant data needed to work on the next Class. This also made it easier to divide the work fairly and efficiently.

Design Principle - DRY Principle

This principle was used extensively as we refactored a lot of existing classes and reused them to the new requirements of assignment 3. For example we used the same contract class for signing a contract with the same tutor for requirement 3 and it's being used to sign a new contract as well.

Design Principle - Open Closed Principle

This remains same as assignment 2 as the code is made extensible (open for extension) by making use of interfaces in the form of adapters for android. Therefore, more functionalities can be added without breaking the code.

Design Principle - Dependency Injection Principle

This has been the same as assignment 2 as we are using intents to pass on data and call new classes, this removes the internal dependencies. Using intent helps when we have to pass on data that is not an instance variable and using intent also makes it easier to test and read the code.

Design Principle - Interface Segregation Principle

We used interface segregation principle again while implementing a viewing contracts list. The interfaces are in the form of adapters since we are using android. The interfaces used are narrow enough so other classes don't implement behaviours they don't need. So Interface segregation principle has been incorporated in this part of assignment 3 as well.

Design Principle - Acyclic Dependencies Principle

There are no acyclic dependencies in our project as well in classes implemented in assignment 3. It is also included where our classes do not have an acyclic graph of dependencies. Most classes depend on each other in a way that ends at some point for example for student reusing contract details. This is to ensure that the system runs in a smooth manner and improves maintainability.

Design Principle - Stable Dependencies Principle

The design also uses the stable dependencies principle where our main package is only dependent on more stable packages provided by the Android framework. Using this earlier on in assignment 2 helped us do refactoring and changing aspects of our package without much effort.

Behavioural Design Pattern

This remained consistent with assignment 2 as we incorporated behavioural patterns into our design where we use adaptors that are used to give a known interface unknown objects. As we are using an API we created adaptors so that we can convert those interfaces to known interfaces. This has been used in the RecyclerView where we take data from the API but make use of known interfaces to make the RecyclerView work properly. Example of this is showing a list of contracts for a signed in user.

Architectural Design Patterns

The Model View Presenter design pattern was used throughout our project. View is basically the User Interface that includes fragments and Activities. These were based on the Presenter which is the mediator between the UI and the Model that includes our API. This is the Java Classes that are mediating between both. Some important advantages of MVP are that code can be easily reused, the design is more adaptive in a sense that it can be easily extended. This technique also helps to implement another design pattern called *layering*, which forces separate data access layers. Isolated implementation using MVP also allows testing of each component separately.

Client Server architecture is also used in our application where the client responds to the server that is maintained by the FIT3077 team. To communicate with the server API calls are being

conducted. The communication is authorised using an API key with username and password being validated.

Refactoring Techniques

Most of the source code contained very little code smells. For example, for the technique “Replace type code with subclasses” was already applied from the beginning: the class OMSConstants holds the coded type values and therefore subclasses can import the values whenever required, instead of coding it in every class that would require those values. The advantage of this technique is that it enforces the SRP and OCP.

Some code smells were detected. For instance, “dead code” such as the class BidCloseDown was found to not be in use and was thus removed.

The “Hide delegate” technique was also applied, where one class doesn’t depend on two other classes which depend on each other but instead one depends on another class which in turn depends on the another class. For example, instead of having PostReusedContract depend on both ContractDiffTutor and ContractRecs to retrieve the variables subject ID and tutor ID, we have PostReusedContract depend on ContractDiffTutor to retrieve a variable subject ID, but the ContractDiffTutor depends on ContractsRecs to retrieve this variable. The downside is that there may be excess classes in the middle.

References

Kanjilal, J. (2015, August 21). Exploring the dependency injection principle. Retrieved May 02, 2021, from

https://www.infoworld.com/article/2974298/exploring-the-dependency-injection-principle.html#:~:text=The%20DI%20principle%20is%20a,a%20separate%20framework%20or%20component.

Soni, N. (2013, January 24). *MVC v/s MVP - How Common and How Different*. CodeProject. Retrieved 20/05/2021 from https://www.codeproject.com/Tips/31292/MVC-v-s-MVP-How-Common-and-How-Different

Nguyen, Q. (2020, May 1). *Architecture patterns in Android — Android architecture design*. Medium. Retrived on 21/05/2021 from https://medium.com/android-news/architecture-patterns-in-android-abf99f2b6f70

Mian U. Rawoteea Y. (2021) Design Rationale Assignment 2 can be accessed on gitlab