# FIT3077 Assignment 2 Design Rationale

by Teh Qi Yuan and Tan Chong Ern

## Class Diagram

## Design Rationale

In this project, we design our application around the Model View Controller architecture[1]. The Model is the API interface provided, the Views are all the GUI components written in Java Swing, and the Controllers include Swing components that take in user input and various classes that inherit from the Publisher abstract class responsible for processing user interactions and notifying related components. In order to avoid cyclic dependencies, we applied Dependency Injection[2] design pattern by injecting the controller into the constructor of each publisher object and we also pass in the pages (Views) that need the publisher into the publisher’s constructor. This allows us to remove hidden dependencies between classes because the classes are asking for what they need before they are built. Hence, you are able to see from the class diagram that there are no cyclic dependencies at all, even though they are common with native applications that implement the MVC architecture in Java. However, one downside to this is that the complexity of our project grew quite rapidly as we needed more and more listeners and publishers for specific actions. After weighting out the advantages and disadvantages of the design, we decided that the extra complexity is needed to avoid cyclic dependencies to work towards a system that is more readily extensible.

Using the MVC architecture also helped us to incorporate the Observer[3] design pattern into our system. The Observer pattern allows us to only notify interested parties and equips our pages with the ability to change dynamically according to the user’s details and interactions. This architecture follows the Facade structural design pattern because it shields the underlying complexity of passing data and redirecting pages from the Application class to other views. Applying the Observer pattern also means that we are applying the Open-Closed Principle[4], and this is evident because we can easily subscribe new components like views and services to publishers, resulting in better extensibility to add on to our software in the future without modifying the existing components. For example, if we decide to implement the collaborative learning sessions for tutors, we can easily incorporate that function by subscribing the relevant classes to the controller for contract creation. Using the Observer Pattern also allows us to avoid cyclic dependencies (Acyclic Dependencies Principle[5]) because now instead of the views passing information into the controller to update the view, the action listeners (e.g. buttons that listen to presses) help to break this dependency by acting as a medium of communication between the controllers and the views.

By applying the MVC architecture, we also adhered to the Single Responsibility Principle (SRP)[4] as each class only has a single responsibility - the Views display the most current/updated information, the listeners process user interactions while the publishers notify every object that is concerned with the new data, and the Model handles requests from the Controller to change state or provide its current state to the Views. Applying SRP allows us to minimize the number of changes required by a class because it will only be modified for one reason and thus reduce the chances of breaking other components that depend on it. SRP makes our classes smaller which improves the readability and maintainability of our system.

The Interface Segregation Principle[4] was also considered in our design by separating interfaces based on their functionality rather than have all the classes implement one interface with methods that it does not need. At first, we only had the ObserverOutputInterface and ObserverInputInterface. However, as we developed our application, we realised that some of the pages did not fully utilise all the methods in ObserverInputInterface. Hence, we created the ListenerLinkInterface containing a subset of the methods that some Views were only interested in.

In accordance with using interfaces to abstract out methods, we also applied the Dependency Inversion Principle[4] along with the Dependency Injection[2] design pattern by reducing the use of concrete classes in method declarations through injecting the dependencies that an object needs into its constructor or methods. Hence, this enables our design to be more easily extensible in the future if additional components are required. This is most prevalent in the various constructors for links, listeners, and controllers/publishers (from their respective package) as well as some helper methods that they use.

We chose to use JAVA Swing classes rather than GUI Forms in the Swing UI Designer to give flexibility, maintainability and extensibility to our application and avoid breaking the Single Responsibility Principle where we would have a God class containing every GUI component in the application. Hence, a lot of our views have similar codes; but because most of the pages contain similar components with subtle differences, this repetition of code is unavoidable.

## References

1. N. Nazar, “Software Architecture - MVC,” 19-Apr-2021.
2. “Dependency injection,” *Wikipedia*, 26-Mar-2021. [Online]. Available: https://en.wikipedia.org/wiki/Dependency\_injection. [Accessed: 02-May-2021].
3. N. Nazar, “Software Design Patterns - II,” 12-Apr-2021.
4. N. Nazar, “Object Oriented Design Principles,” 15-Mar-2021.
5. N. Nazar, “Object Oriented Design Principles - II,” 22-Mar-2021.

## External Libraries

1. JSON – Java (org.json-20210307) in StuTor/lib/json-20210307.jar