Introduction

The big picture of our design is to focus on separating components(classes and packages) and bundling all related components together. We have 4 main packages in our system.

1. API - Api calls are made in this package. The python requests library is used and most of the time, some json data is returned
2. Details - This package makes interaction with api easier. This is done by having classes to be able to return particular data.
3. SRC - This package is where all the business logic happens. The data that is fetched from the server is dealt appropriately to match the requirements of the system.
4. UI - This is where the code for the user interface of the system is.

The main reason we divided our system into different packages was because we wanted to obey the single responsibility principle. This is to make sure that modules that are related to one another are encapsulated in one module so that the modules are easily reusable.

Design Patterns

This program follows a few design principles that we have learned in FIT3077. The first principle we applied was Abstract Factory. Seeing that most of the RestAPI calls have similar calls, we decided to implement an interface. Abstract_API_methods class. The class is inherited by all the API files we have created to ensure that these methods are implemented in the respective classes in the files. The advantage is that the Single Responsibility Principle is maintained and if we decide to modify our code, we can do so without breaking the client code. Creating an interface also allowed us to implement a Strategy Pattern as it allowed us to add any additional methods without making changes to all the classes that have implemented the interface.

We also implemented a Façade Design Pattern. Seeing as Façade provides an interface to connect with a more complex subsystem, we created API classes that allowed us to connect with FIT3077_API and obtain or create data as per the interaction with the Graphical User Interface. The inner workings of an API maybe complex, therefore it was crucial to create something more straightforward to work with the data.

Moving on, Observer pattern was also implemented. The reason is because as we are using a GUI to work with a RESTAPI, the requests made to the RESTAPI are dependent on the interaction between the GUI and the user and as such, small changes in one area would affect the next GUI. For example, the selection of Subjects. If a particular subject that a student wants is not in the system, the user must select other and a new box opens, requesting an input. This would create a new subject in the system. Moreover, when a user is logging in, the observe pattern comes into play as tutors and students have different main screens. Template method was also implemented to prevent redundancies. The contractUI class inherits from the Contract_creation class and uses the methods in the Contract_creation class to generate the UI for contract along with implementing additional methods.

Design Principles

The Design Principles we followed, one of them was Single Responsibility Principle. We created classes that were responsible for one thing alone and integrated them with out GUI. This prevented us from forming God Classes and allowed us to debug our program in a more comprehensive way.

The implementation of an interface enabled us to follow the Open/Closed Principle where changes to the interface would not affect the classes that are implementing the interface. Interface Segregation Principle was also followed as the interface we created only contained the behaviours that would be implemented by the client classes, preventing a 'fat' interface from forming.

Seeing that we have created separate packages for different functionalities of the program, we also followed Package Cohesion Principle where we imported classes to allow the code to be reused instead of copying codes at every instance.

Inside the Bid package, we have a DisplayBids abstract class which TutorBids and UserInitiatedBids inherit from. This was done because we needed to have much abstraction as possbile so that we can follow the Open close principle. This means that in the future, if we want to have another criteria of users, we can just extend from that abstract class instead of copy pasting that code. This same idea is also applied for the Messaging system between tutor and student. There is an abstract class AllMessages. In our current design, the StudentMessagesView and TutorMessagesView inherit from this AllMessages class to display the messages depending on who has logged in. It is also easily extensible and follows the Open/closed principle.

In the above mentioned abstract classes, we follow the liskov substitution principle. This makes sure that the methods in the class are not different from the parent class.

Inside the UI, we have BiddingUI, ContractUI, LoginUI etc. This design choice is made so that we separate the modules from other parts of the system and encapsulate only whats necessary within that particular package. This also makes it easy to find the relevant modules when the system gets bigger and bigger.

Lucid chart link - https://lucid.app/lucidchart/invitations/accept/inv_d836db6e-d6b7-456c-8da1-0e5a4cc93572