Design Principles (2)

1. Single Responsibility Principle
   This principle is applied in the MVC Software Architecture.
   Also, the ViewExpiredContractsController, ChooseTutorController, ChangeContractDetailController follows this principle as well because each of them carry out their specific roles and responsibilities. We follow this principle as to not create “god” classes, which will make the code harder to extend and refactor.

Design Pattern Applied

1. **Facade Design Pattern**
   The ViewExpiredContractsController Class and it’s UI follows this pattern. We chose to use this design pattern for this class because the users can interact with the expired contracts in many ways such as renewing or reusing these contracts. Thus, we used this design pattern to provide the users the services of the subsystem involved with the use of these expired contracts. This can also be said the same for the HomePageController, ViewBidsController, ViewOwnBidsController, ViewOfferController and their respective UI classes as they all provide the users access to services that is related with what they are displaying.

2. **Observer Design Pattern**
   The StudentActions Interface, TutorActions Interface, User subclasses and NotificationController Class in a way follows Observer design principle. The user action interfaces together with the user subclasses act as the publisher class in the observer pattern because they will detect whether a contract will expire within 1 month and the NotificationController Class and it’s UI will act as the subscriber in the observer pattern context as it depends on the publisher class to update it on any soon to expire contracts. One difference this has in comparison to the observer pattern is that the system would only have 1 subscriber and 1 publisher since there will only be 1 instance of the user when the application is run. The design pattern follows Open/Closed Principle where we can have more notification classes for a single user sub class instantiated without need to change the publisher class but this is not necessary as it is redundant to have multiple notification pages showing the same content to the user and also the disadvantage of having subscribers being notified at random order is negligible because we only have one subscriber.

Software Architecture

The architecture our design follows is **MVC Architecture**. This is because our classes are divided into the three types which are Model classes, View Classes and Controller Classes. The Model classes represent the objects that the program will be interacting with and they determine the behaviour and data the system will have to work with. The Model Classes include the User Class, Action Class, Student Class, Tutor Class and StudentTutor Class. The View Classes are responsible for displaying information to the users. The View Classes are the View Class, UI Class and UI Class’s subclasses. In our system each page of our application has a UI class that is in charge of displaying it. The controller classes in our system are responsible for determining how the application will act depending on the user inputs and interactions. They are also in charge of determining what will be displayed on the view class. The Controller Classes are Controller class and it's subclasses. Each UI page in the application will have a controller that is in charge of it. We use MVC Architecture because it separates the functionality into classes with clearly defined roles to prevent creating a "god" class and it is to follow acyclic dependency principle because it prevents the model classes and the UI related classes from having a cyclic dependency.

Almost all of our application follows passive MVC architecture because only the controller of a UI page can trigger a change in the View class it is in charge of and change the model class it is interacting with. The only problem with this is that the model class being used has no way for it to report that it's state has changed. But this problem is negligible because the model only changes under the control of the controller class. So in our application, the model only changes when the UI page is changed or when the page needs to be refreshed.

The only class that does not follow this variation of MVC Architecture is the TutorDashboard Controller/UI classes. It follows an active MVC Architecture because it has a time-based trigger where it will update the page every 10 seconds.This other variation of MVC Architecture does not depend on view class, the scope of the changes are only restricted to the view class only and it makes testing the UI for this particular class easier. The disadvantage of using this variation is the added complexity because of the added time-triggered event for updating the page and use of an observer design pattern. The class will also not suffer significantly because of update requests for the view class because the time-triggered events are well controlled and will not overload the view class with update requests.

MVC Architecture is chosen because it allows us to separate the GUI components logic with the business logic of the GUI. This reduces the complexity of GUI classes and allows for ease of extension to the classes. We can change the controller class without affecting the View class. Also, MVC is in compliance with the Single Responsibility Principle where one class should only have one responsibility. The Controller class is in charge of the logic of the UI and the View Class is responsible for the GUI components and layout. This increases readability and reduces errors.

Our application makes use of **Service Oriented Architecture**. This is because the application makes use of RESTful web services provided by the fit3077 website to do certain functions that are required for each functionality of our application to work properly. This includes retrieving information or altering information from the website's database such as getting a student's information or creating a contract between a student and tutor. This architecture allows our classes in our application to be loosely coupled to allow our classes to be easier to reuse.

Refactoring Techniques

1. <u>Composing method</u>
   Extract method

In the subclasses of User class (Eg. Student, Tutor, StudentTutor), we noticed there is a duplication of code in the constructor and the method updateActionUI(). Thus, following DRY Principle and for future maintenance of the system, the code is extracted and combined into the method updateActionUI() which is then called by the class constructor. This reduces duplication of code and improves readability which will allow future extensions or modifications to be carried out easily as we would not need to modify the same code at multiple places.

2. Moving features between objects
   Extract Class
   Before refactoring, the GUI components and the business logic of the UI are all placed inside the same class. This makes it hard to extend the class as the class is large and the GUI components and business logic are not independent of each other. Thus, the business logic of the GUI components (eg. ActionListener for a button) is extracted to another class that inherits the Controller class. This adheres to the Single Responsibility Principle where each class should be responsible for one responsibility. The UI classes are now only responsible for the GUI components and the UI Controller classes are responsible for the business logic for the GUI components and the UI.