Pink coloured classes are the classes created/edited for Assignment 3 extension.
Engine
Driver
- UserAPI userAPI
+ main()
<<creates>>
View
LoginView
UserTypeView
DashboardView
BiddingCreationView
BiddingView
OpenClosedBidView
TutorBidOffersView
BidOfferView
CloseBidMsgView
SignContractView
ContractCreationView
RenewContractView
ContractExpiryNotificationView
javax.swing
Controller
LoginController
UserTypeController
DashboardController
BiddingCreationController
OpenClosedBidController
TutorBidOffersController
BidOfferController
CloseBidMsgController
SignContractController
ContractCreationController
RenewContractController
java.awt.event
java.util
Observer
Observable
LoginSystem
UserCookie (Singleton)
UserFactory
Tutor
Student
User
Model
DashboardModel
BiddingModel
BidOfferModel
BiddingCreationModel
ContractCreationModel
RenewContractModel
Bidding System
ActiveBidsIterator
Message
Subject
Competency
BidFactory
Bid
OpenBid
CloseBid
BidInterface
LessonInfo
Iterator
Contract development
ContractLessonIterator
Contract
ContractExpiryIterator
API
APIFacade
APIService
APIAdapters
APIRouter
UserAPI
BidAPI
MessageAPI
ContractAPI
ClientInterface
BaseAPIInterface
UserAPIInterface
BidAPIInterface
MessageAPIInterface
ContractAPIInterface
com.fasterxml.jackson
org.apache.http

The **Adapter pattern** is used in handling the API requests throughout the program implemented in the APIAdapters and ClientInterfaces package. Here the concrete classes inherited from the APIRouter abstract class are the adapters and the APIService class is the adaptee. The interfaces in ClientInterface were introduced adhering to each concrete adapter class to provide a clear protocol for the client to use the adapter. This pattern is used to ensure that the adapters are getting raw data from the client and passing the correct data structure to the adapter. Furthermore, this pattern allows new APIs requests to be added easily by just introducing a new adapter class extending from the APIRouter without changing anything from the APIService which upholds the **Open/Closed Principle**. The downside of this pattern is that every new adapter introduced needs to introduce a new interface in order to adhere to the adapter pattern rule.

The **Facade pattern** is used in the API package. This is to provide convenient access for the client to make API calls and also isolates the systems from the complexity of the API package (subsystem). Client classes only have to call the appropriate function in the APIFacade to make an API request. With this pattern, clients do not need to initialize all the APIadapters, keep track of dependencies and execute methods in the correct order, as all these will be provided in the functions of the Facade class. Therefore, it also greatly increased the readability of the code outside of the API subsystem. [1] Moreover, the APIFacade class is also transformed into a **Singleton** since a single APIFacade object is sufficient across the system. However, this violates the Single Responsibility Principle as it contains a large number of function calls which interact with all concrete classes of the APIAdapter package.

The **Factory pattern** is implemented in the UserFactory class and the BidFactory class. These factory classes are used to create variants of User and Bid objects where the client only has to pass in a param and the factory class will return the client the appropriate subclass. This is to ensure that we avoid tight coupling between the UserFactory/BidFactory and the Student/Tutor or OpenBid/CloseBid classes. Hence, these classes also uphold the Single Responsibility Principle as product creation code is extracted into one place in the program which also increases the readability of our code.
Moreover, it also provides the convenience of **Open/Closed Principle**, where we can easily introduce new variants of the product into the program without breaking the existing code.

Finally, we implemented the **Observer pattern** for the **active MVC architecture** used throughout the program which depends on the classes provided in java.util. We have decided to implement this as the View classes need to handle data changes in the Model class dynamically. Therefore, this pattern is enforced so that whenever a user presses the refresh button, the Controller will listen to it and inform the Model (observable) a change has happened and handle the data update, the Controller observing it will trigger its update() functions and updates the View's content. This pattern solves the problem of hectic and abundance of function calls from Model to Controller and finally to View.

The **Common Closure Principle** is used in the BiddingSystem and API packages, where all the classes related to the bidding functionality and all the classes related to API requests are grouped together accordingly. For instance, when a Bid object is changed, all the affected classes have to be changed and only these classes have to be changed, same applies to the API's classes. This way it helps in increasing maintainability where developers change something in the package, they know which classes have to be changed accordingly that are within the package.

Newly implemented design pattern according to the requirements for Assignment 3 is elaborated below.

The **Observer pattern** is used for implementing requirement 1 where the tutor subscribes to open bids. In this case, the tutor is the Observer and the Observable is the Bid class. This is to efficiently notify the tutor whenever the bids that this tutor is observing are updated.

The **Iterator pattern** is used to handle the operation of getting all of the active bids and contracts from the server in the ActiveBidsIterator, ContractLayoutIterator and ContractExpiryNotificationIterator class. It provides the benefit of traversing elements of a collection without knowing the underlying representation stored in the server[2]. By implementing this pattern it also hides the complexity of the data structure from the client, which makes the source more readable. Furthermore, it also served as a refactoring technique where in the iterator classes, we further extracted out the button instantiation to modularize a big and huge function into smaller functional functions. It upholds the **Open/Closed Principle** which makes our program to be easily extensible. Hence, new functionality and classes can be added to our program without changing much of the previous source code. The downside of implementing this pattern is that it might create an unnecessary iterator class when the list is not performing too complex business logic.

The **Prototype pattern** is used to handle the renewal of contracts. We have implemented it in the Contract class, this is because we want to get exact copies of the previous expired contract without coupling the code to their concrete class.

In our program, we have implemented the **Model-View-Controller architecture**. We have chosen this architecture because it provides encapsulation between the user interface and the underlying logic behind. This will greatly improve our code readability as the UI interface does not directly update or modify the data within the model. Thus, our program does not fall under the Acyclic Dependency Principle which greatly reduces the coupling between classes in our system. Other than that, the responsibility of each class can be kept minimum and makes the flow of data more efficient and clear.

As mentioned previously, we are implementing **Active MVC**, this is because some of our functionality requires our Model classes to change state independently of the Controller and it has to alert the View to update its displaying data. Hence, the decision of implementing this variant of MVC as active MVC implements the Observer pattern. Other than that, this variant of MVC also provides ease of testing, when an error occurs, we can easily debug if the error happened at the View, Model or Controller component.

Refactoring Documentation

Composing Method

Extract Method

We have implemented the Iterator design pattern to perform this exact method technique in some of the View classes that is to display a list of items (i.e. list of bids, list of contracts). This is because initially the View classes are having some functions with a long body that performs a lot of buttons and table instantiation. We have extracted out these small components in the iterator class, to avoid duplication and isolate the independent parts and also achieve higher readability.

Moving features between objects

Move Method/Field

In Assignment 2, we have already implemented this technique. For example, a tutor can offer a bid, instead of putting the offer bid function into Tutor class, we moved to Bid class, because tutors can only offer a bid when there's a bid. If putting the offer bid function in Tutor class, the function has to have dependency to the Bid class to check if this Bid object exists, however, we already know tutor can only offer the bid only when the Bid object exists.

Organising Data

Self Encapsulate Field

Noticed in all our concrete object classes (e.g. Student, Tutor, Contract, OpenBid, CloseBid), we have all getter and setter methods for all the fields in the class. This is to ensure the convenience of the client to get and set the fields easily without having to go into the class and understand how to access the fields of the object.

Replace long method parameter with parameter Object

Notice that functions such as renewContract() in the RenewContractModel class, we tried to keep the number of parameters to the lowest by passing in an JSONObject, which then the function call within gets its parameter from. This cleans up the source code and increases the readability of it.

Replace Data Value with Object

From Assignment 2, we noticed that Contract, Bid, and BidOfferModel are having similar data values for the information of the lesson, hence we decided to replace these data values as an Object called LessonInfo by creating a new class name LessonInfo. This is to help in having repeating getters across the Classes that need to know about these data values. This technique has directly adhered to the **Single Responsibility Principle**.

Replace Type Code with Subclasses

We have adhered to this refactoring technique in our User subtypes (i.e. Student, Tutor) and Bid subtypes (i.e. OpenBid, CloseBid). This approach enhances **Single Responsibility Principle** and **Open/Close Principle**, because we extract out the methods only the subtype can perform while extending the common methods from its base class.

Resources:
[1] Relations with Other Patterns https://refactoring.guru/design-patterns/facade
[2] Iterator pattern https://refactoring.guru/design-patterns/iterator