# DESIGN RATIONALE

## ASSIGNMENT 2's REMAINED DESIGN

### Model classes (User, Bid, Contract, BidAddInfo, ContractAddInfo)

These classes are majorly unchanged except for new queries added.

***Pros:***

High separation of concern and high usability. As indicated by their names, these classes are in charge of communicating with one and only one specific API endpoint. This allows clients to easily search for the query they need.

***Cons:***

The use of *BidAddInfo* and *ContractAddInfo* gives code smell "Inline class" as these classes are mainly created to bundle up their attributes and provides no specific functionality other than getters and setters for these functionalities. Yet they are kept in the system to avoid crowding their "hosting" class with an overwhelmingly large number of attributes.

## REFACTORED DESIGN

### Architecture: Model - View - Controller

To achieve Separation of Concern and avoid cyclic dependency, we've refactored the entire system into MVC architecture with Observable pattern. Models, Views and Controllers live in their designated packages whereby Models do not know of classes beyond their package and changes to Models can only be executed by Controllers.

***Pros:***

It allows decoupling within the system resulting in *models, views* and *controller* packages that can evolve much faster independently. Business logic often appears in Model but hidden from View, thus the controller is responsible to manage the interaction between Model and View.

***Cons:***

In order to adhere to MVC, the listeners must be extracted to the Controllers, significantly increasing the Controllers' complexity.

It is resource intensive where every user's action will invoke the view → controller → model chain again. On top of that, the Swing GUI does not cleanly separate the view and controller, increasing the complexity of the codes.

### Refactoring Techniques: Extract Method

*StudentController* and *TutorController* take in the **Extract Method** when dealing with duplicate codes. When activation of student and tutor are subscribed to creation of bid from a model, we can reduce the repeated codes by calling the extracted methods from both tutor and student activation.

***Pros:***

This adheres to the DRY principle and enhances the readability of the codes.

***Cons:***

This significantly increases the number of methods in the controller classes.

### Packaging

The design of packages complies with the **Common Closure Principle (CCP)** and **Stable Dependency Principle (SDP)**. In particular, the packages *model, studentview, tutorview and*

*mainview* are stable as it doesn't have any outgoing dependency and hence package *controller* depend on it (SDP).
Classes in *studentviews* are closed together against changes in Student's functionality, while those in *tutorviews* are closed together against changes in Tutor's functionality (CCP).
***Pros:***
CCP ensures changes are closed in one package and hence, enhances the system's maintainability.
SDP ensures that the system is resistant to changes by having the less stable package depending on more stable ones.
***Cons:***
This increases the number of packages as there are now 3 different packages for views instead of just 1.

## NEW DESIGN
### REQUIREMENT 1: Tutor monitoring dashboard
**Observer Pattern**
TutorMonitorView is an Observer subscribing to *BID_FETCH_RESPONSES_FROM_API* EventType of the Bid Model, allowing it to automatically refresh whenever the Bid is updated.
***Pros:***
This pattern fulfils the **Dependency Inversion Principle (DIP)** where TutorMonitorView depends on the abstraction of Observer interface instead of concrete Bid class and vice versa, Bid depends on Observable interface instead of concrete class
***Cons:***
This implementation is spread out over many classes. Without a solid grasp of MVC and how the system works, it is a challenge to maintain.

### REQUIREMENT 2: Contract expiration and renewal
**No special design implemented, all implementation complies with existing MVC architecture**

### REQUIREMENT 3: Reuse previous contract
**Strategy Pattern**
As per Strategy Pattern, *Controller* is the Client, *ReviseContractTerm* class is the Context, *CreateDifferentTutorContract* and *CreateSameTutorContract* are Strategy. On receiving user's input, *Controller* will set the appropriate Strategy to *ReviseContractTerm,* which then executes these Strategies.
***Pros:***
This implementation allows a clean approach to switch between different ways of reusing contracts, overall making the system more comprehensible and maintainable.
***Cons:***
When integrated with MVC architecture, the Context (*ReviseContractTerm)* is left with little functionality other than switching between Strategies. Conversely, the Client (*Controller*) has to know more about the Strategies than it ideally should in order to retrieve data from these Strategies.