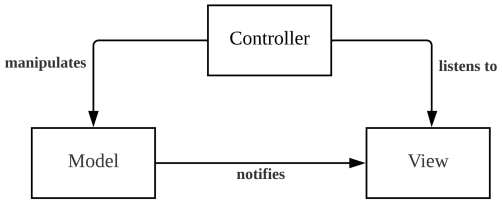
Design Rationale

The following discussions on design rationale are built on top of the assignment 2 design such that only relevant parts that were modified in assignment 2 to adhere with assignment 3 requirements were discussed below.

1 Software Architecture

Model-View-Controller (MVC) Architecture



MVC architecture was implemented in assignment 2 and extended in this assignment for:

1. Monitoring (MVC) that allows students to view the bids to be monitored.
2. ContractRenewal (MVC) that allows students to perform contract renewing without interfering with other classes.
3. ContractConfirm (MVC) that allows both students and tutors to perform contract confirmation, along with its contract duration upon selecting an offer (for student) and buying out a bid (for tutor).

These new requirements were added into the architecture as it supports greater maintainability as the model, view and controller are separated in different packages, obeying **Single Responsibility Principle,** while ensuring that Views know about the Model but not vice versa (following the **Acyclic Dependency Principle**).

Advantages

The advantage is that tasks in the frontend and backend can be carried out autonomously without impacting each other's work as dependencies are easier to understand with clear separation of responsibilities. Since (in general) UI code is difficult to test (require manual testing), the clear separation of concerns with MVC allows us to test the UI independently of the business logic.

Disadvantages

Tight coupling can be seen in controller, view and model. For instance, adding a button requires a modification of the corresponding controller. This is an inherent limitation of MVC (from the use of Java Swing). It increases the complexity of the code, leading to possible difficulties in maintaining it. As our code makes heavy use of events and observes, if a bug occurs, it will require an inspection of propagating classes (of model-view-controller).

2 Design Patterns / Principles

Note: Only 3 design patterns are added into assignment 3, while other new requirements have utilized (extended from) the old design patterns instead.

Strategy Pattern

A new design pattern of Strategy Pattern was implemented on the aspect of contract renewing. This allows us to use different variations of contract renewal algorithms **interchangeably** and **dynamically**. It improves the extensibility of the system by enabling future operations to instantiate the existing strategies or adding new strategies without needing to modify existing implementations.

Advantages

An advantage of this is the implementation of the renewing contract is separated from the code that uses it, adhering to **Separation of Concerns**. Furthermore, these algorithms/classes (RenewByNewTerms, RenewByOldTerms) are **loosely coupled** with the context entity (RenewStrategy) and they can be easily changed and replaced without changing the context entity.

Disadvantages

However, the disadvantage is that the client (ContractRenewalController) needs to instantiate all possible strategies in a HashMap to be ready to use, this may result in **many objects within a single class and could be bloated**. Furthermore, the client also needs to know the existence of different strategies in order to select the

right strategy and **does not entirely hide the implementation** of the aggregate structure. For instance, the ContractRenewalController needs to explicitly specify the strategy to use: `renewStrategies.get("new").renew(contractSelection)`, which becomes slightly challenging in the aspect of **maintenance**.

Builder Pattern (Modified Version)

A slightly modified version of the actual Builder Pattern is designed with a builder package that contains classes for building Bid and Contract(s). Although the actual builder pattern is used for complex object construction, our system does not have complex object construction. Thus, the builder pattern is modified such that it allows us to create a package that is solely focused on one functionality - to build objects, with a BuilderService class that focuses on providing an interface on utilizing the builder package. Although the full builder pattern is not applied here, but the concept of "building objects" are applied, hence the "modified version of builder pattern".

Advantages

An advantage of this is that it creates future **extensibility** as it allows new forms of user-defined objects to be built, while subsequently added into BuilderService without modifying much code, adhering to **Open-Closed Principle**. The construction of objects are also separated from the code that is using it, adhering to **Single Responsibility Principle**.

Disadvantages

The disadvantage of our current approach is that our builder package (namely the ContractBuilder) does not break its object constructions down into separate parts and any new forms of contract construction that is added into the ContractBuilder may cause the class to be **bloated**. As such, code is required to be refactored further.

Template Pattern

Template pattern is used to group and generalize views that have almost identical algorithms when minor differences, while avoiding repeated code with the use of "**common class**". Other views can easily inherit this ViewTemplate class, using default methods and only implementing abstract methods.

Advantages

The first advantage is that it enforces a structure to view classes, streamlining the addition of further view classes. Furthermore, it **reduces duplicated code** by extracting common methods and placing them into the template class, where modifications such as the default size of the Frames, or the Colour of the frame titles, etc. are easily completed. This makes debugging and testing easier as now all views have the same structure, and duplicated code is in one place, leading to **better flexibility** and **future views extensibility**. [1] Using this pattern also allows us to maintain **clean code** because subclasses only override functions that varies while reusing common functions in the base class.

Disadvantages

The disadvantage is that views are limited as new views may be **constrained** by the skeleton provided by the template. It also has to conform to all the abstract methods defined in template view, where at times certain methods **implement the abstract method when it does not need it**. As such, the effect of this disadvantage is alleviated in our system by making abstract methods as general as possible while inheriting from the Template only when it is needed. [1]

Single Responsibility Principle (SRP)

SRP is applied throughout the entire system where every class is responsible for a single functionality of the program. The benefit is that complexities are reduced and if there is a change with any aspect of the program, only corresponding classes will be changed, preventing the chance of breaking other parts of the code.

Liskov Substitution Principle (LSP)

LSP is applied in all our inherited classes, whether in Strategy, DashboardView ect. These properly generalized abstractions ensure that the pre/postconditions and invariants of the parent classes are maintained. This strengthens polymorphism and ensures that child classes do not have unexpected behavior that would complicate testing, debugging or extending.

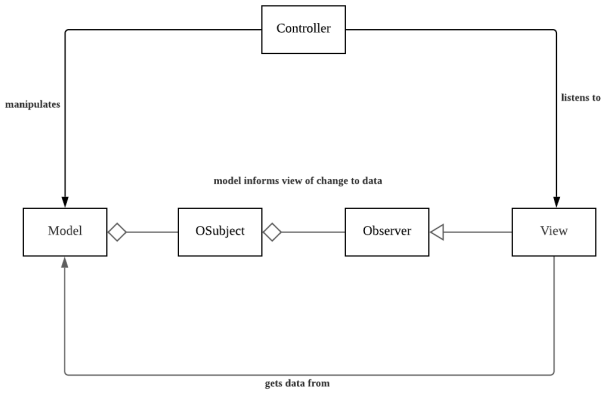
Dependency Inversion Principle (DIP)

DIP is used alongside with Strategy Pattern where ContractRenewalController depends on high-level abstraction of RenewStrategy rather than specific renewing strategies. The benefit is that it provides hinge-points in the design so that new subclasses can be added without affecting the dependencies on abstract classes. It also helps to decouple the details of renewing strategies from the client (ContractRenewalController) that uses it and prevents rippling effect from changes inside low level modules (RenewByNewTerms, RenewByOldTerms)

3 Package-Level Principles

Acyclic Dependencies Principle

Acyclic Dependencies Principle was used throughout the entire system. None of the packages developed creates a cyclic dependency, thus if there is an issue with a package, it is less inclined for that one package to break other packages in a domino effect. This is supported by the use of MVC architecture as discussed above.



The diagram above illustrates how cyclic dependencies are prevented in our system.

- The use of built in Event Listeners for Java Swing UI elements, allowing user interactions to communicate only through event listeners rather than needing views to manually call methods within the constructor.
- The user of observer pattern decouples model and views such that model does not need to call methods within the views.

Common Closure Principle (CCP)

Classes with similar functionalities are grouped together in packages throughout the entire system, namely:

- All views (user interface) are grouped into `views` package.
- All application data are grouped into `model` package.
- All interpretations of user inputs are grouped into `controller` package.
- All interactions with API are grouped into `api` package.
- All streaming of API data are grouped into `stream` package.

This benefits our system because a change that occurs (in a package's class) will propagate only changes within the same package, increasing overall maintainability as the number of classes affected is reduced. However, the use of MVC violates CCP such that when a modification is made on the view, a change is propagated across its corresponding controller and model (if needed).

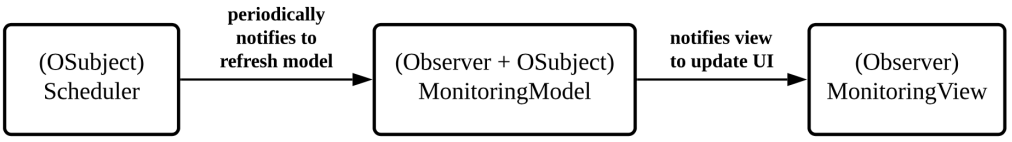
Common Reuse Principle

Certain components in our system are also grouped together based on likelihood of reusing together such as Observer and OSubject are always reused together. However, the adherence to CCP as above implies that certain packages contain classes that might not be reused together. For example, not all API classes are going to be reused together. Since it is unlikely at this point that many of our classes are going to be reused for other functionalities, thus this trade-off is balanced. As this is a tailored application with a narrow use case where it is unlikely for login functionality to be reused at other parts of the program.

Note: Both CRP and CCP are used in our system, thus balancing the aspect of **maintaining module** and **reusing module**.

4 How the Initial Design Supported New Requirements

Observer Pattern



In assignment 2, observer pattern was used to ensure that the UI (views) always displays the latest information according to the model. As the new requirement of monitoring dashboard was introduced, thus a Scheduler class was developed that extends from the OSubject class, making Scheduler to be a subject that periodically informs and updates the Observer(s) - MonitoringModel, who will refresh its model and subsequently notifies the view (Observer), as illustrated in the flow above. Since the main responsibility of the Scheduler is to notify the observers periodically, thus no refactoring was involved in this Observer pattern as existing functionalities were **overlapped** and **reused** rather than creating new methods, adhering to **clean code** and **better maintainability** over time.

Open-Closed Principle

Since we are required to design a contract renewal system, a model needs to be developed (ContractRenewalModel). As this contract renewal system has different attributes from the existing models (of assignment 2) and share no common properties across all extended classes of BasicModel, thus ContractRenewalModel was extended from BasicModel to **utilize the common properties** in the base class such as ExpiryService, OSubject and UserId. This also follows the **Open/Closed Principle** as the superclass BasicModel is not modified but its subclass is created and open for extension. This is beneficial as it provides a layer of abstraction and ensures **loose coupling**. In addition to that, properties and methods which are distinct to contract renewal only are also added into ContractRenewalModel only, adhering to **Single Responsibility Principle** at the same time.

5 Refactoring Techniques Used

Pull Up Field / Pull Up Methods

Pull Up Field is used in removing fields and methods from the views while moving it to the superclass (of ViewTemplate). This eliminates the duplication of fields and methods in the subclasses.

Rename Method

Rename refactoring method is applied throughout the system to provide better code readability. For instance, renaming the instantiation of buttons from `updateButtons()` to `createButtons()` to better reflect its functionality.

Move Method

When a new feature of contract renewing (with old and new terms) is added, the BuilderService is refactored such that the methods are extracted into a separate class of BidBuilder as extension on ContractBuilder is performed in the system.

Replace Temp with Query

Our views need to query information from the models, as such separate methods in the view classes were created to return the information as opposed to direct query from the model. For instance, having `getBidInfoList()` in OpenBidView to query a list of open bid offers from OpenBidModel.

References

[1] nagnath. (2021). Pros and Cons of Using Template Design Patterns. Retrieved from https://www.codechef4u.com/post/2018/10/25/Pros-and-cons-using-Template-Method-Design-Pattern