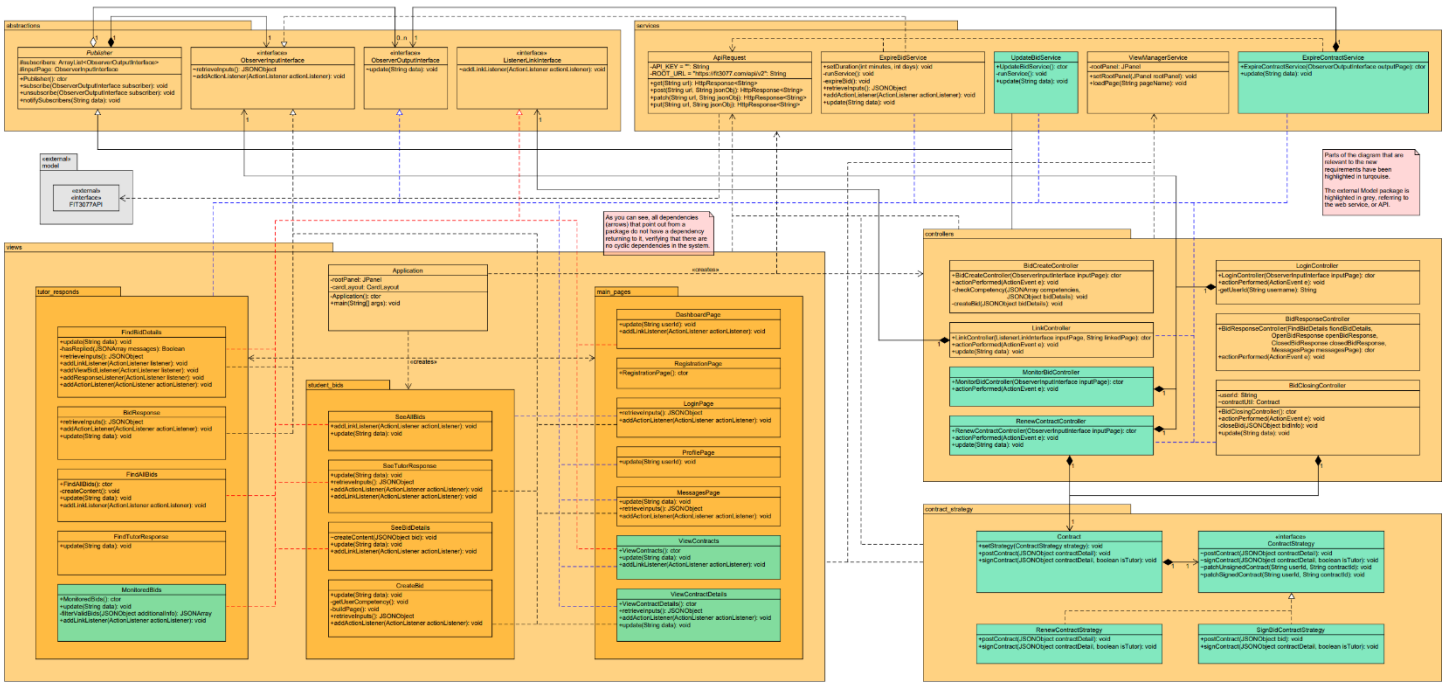
FIT3077 Assignment 3 Design Rationale

-by Tan Chong Ern & Teh Qi Yuan-

Class Diagram

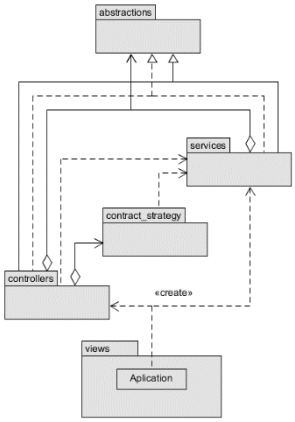


Design Patterns

We used the Strategy design pattern for the renewal of contracts and the signing of contracts when bids are closed down. This is because the code for both approaches are quite similar, but they have minor differences that are at the same time unavoidable. For example, renewed contracts need to be signed by both parties, while closing down a bid and signing the contract for it does not. By using this design pattern, we were able to reduce duplicated code because the varying behaviour between the two ways of signing contracts are extracted into different strategies, and they both use the same Context class which in our case is named 'Contract' to implement those strategies. Hence, doing this gives us the advantage of isolating the concrete implementation details of a strategy from the code that uses it. Another reason why we chose the Strategy design pattern is that it replaces inheritance with composition as each strategy inherits from the common strategy interface, allowing the 'Contract' class to swap between different strategies easily depending on whether it is renewing contract or signing a bid. Using this design pattern also allowed us to have a higher level of extensibility so if another new contract signing strategy is introduced (e.g. signing a contract between one tutor and multiple students), we can easily add the new strategy by having it inherit from the strategy interface so that 'Contract' class can use the strategy without having to change its context.

Note that for the monitoring dashboard, the design pattern that was applied was the Observer design pattern, as an extension to the MVC architecture that was applied to the previous assignment, where the data in the dashboard would be updated whenever a tutor decides to monitor a bid. Hence, new bids can only be added into the monitoring dashboard if the tutor clicks on the "Monitor" button, and not by any other action/event that is not related to it, depicting a separation of concerns.

Package-level Principles



Our system design is centred around four main abstractions, namely ListenerLinkInterface, ObserverOutputInterface, ObserverInputInterface, and Publisher. These four classes allow us to exhibit the Acyclic Dependencies Principle between packages because now all our classes will depend on one or more of these interfaces instead of depending on each other (DIP), thereby avoiding cyclic dependencies. Reducing cyclic dependencies not only makes our code easier to change because classes are not tightly coupled, but it also reduces the complexity of our code, making it is easier for us to debug and add new features. Furthermore, having the details of our system depend on abstractions also led us to adhere to the Stable Abstractions Principle (as you can see in the diagram below) because all the dependencies are pointing upwards (no cycles!) towards the abstractions package, which is the most stable and abstract package in the system. This is true because it is very unlikely for those abstractions to change in the future since they are abstract, minimizing the impact of changing or adding new features.

We packaged the classes that do not fit into MVC by using the Common-Closure Principle. The reason why we chose this principle is because we realise that these “extra” classes can be grouped together under three main categories which are services, the contract strategy, and abstractions. We considered both the Common Reuse Principle and the Reuse-Release Equivalence Principle but both principles caused us to end up with a lot of packages that contain one or two classes within, bringing unnecessary complexity to our system. This is because the classes are not tightly bound to each other as most of them only depend on abstractions. So in our case, we chose to use the Common Closure Principle as it was the most ideal principle in our situation. Hence, packages that perform similar tasks - like servicing the Views without user interaction (service package), applying the Strategy design pattern for signing of contracts (contract_strategy package), and providing stable dependencies to other classes (abstractions package) – fit into the criteria of CCP where classes should be packaged against the same kind of changes (i.e. have similar responsibilities), even if they are just conceptually connected and not physically connected. Another reason why we used CCP is because we want our system to be more easily maintainable so that it is easier for us as developers to fix issues and add enhancements to our application.

Design Principles

In this assignment, we tried to enforce the Don’t Repeat Yourself principle more heavily. As you can see from the class diagram, even though we have added requirements (and hence, classes) for the new assignment, the number of classes in our system has remained about the same. This is because we realised that we had quite a few simple controllers that had similar responsibilities. So, we instead merged those classes into a controller (LinkController) that has the generalised functionality needed to support their tasks. This allowed us to reduce repeated code and therefore the complexity of our system, as doing this resulted in less classes and less convoluted packages. If we had not performed this refactoring, we would have ended up with more and more controller classes that did not differ a lot with existing controllers. Thus, merging similar controller classes and injecting the dependency they needed instead of creating a new class for each new controller has definitely helped increase the maintainability and readability of our system. (More details under Refactoring Performed.)

We have maintained the design principles that we employed in the previous assignment, namely the SOLID principles. However, the design principle that was the most prominent in this assignment was the Open-Closed Principle and the Dependency Inversion Principle which stemmed out from the abstractions package and the use of the MVC

architecture. Furthermore, because the Interface Segregation Principle was applied properly before, new classes were able to only inherit the necessary abstractions and methods. For example, some classes that need to update themselves and notify other relevant classes will inherit the ObserverOutputInterface and Publisher classes whereas simpler classes that only need to update themselves will only need to implement ObserverOutputInterface. Overall, the system design from the previous assignment has heavily support extensibility for the new features in this assignment.

Architectural Pattern

The architectural pattern that we applied was carried on from the previous assignment, which is the Active MVC architecture. We have actually been very pleased at how extensible the architecture was when we implemented the new requirements. The MVC architecture very supported the Open-Closed Principle to a great extent, because whenever we completed a new feature, we could just directly subscribe it to the correct controllers and the application would instantly be able to accommodate them. It was slightly challenging and tedious to incorporate the MVC architecture at first as we had to figure out how to properly integrate our existing classes into the Model, View and Controller components. However, further developing our application with new features or updating old features proved to be very simple and swift.

Applying MVC also allowed us to work on different parts of the system concurrently. For example, I was able to work on the View for a specific page while my partner worked on the backend Controller without needing me to complete my part first before he can proceed with his. This is because MVC provides Separation of Concerns between the Views (frontend) and the Controllers (backend), which is another important reason about why we chose to use it. MVC enables us to speed up our development time as we can develop multiple related components at the same time and there was ease of testing as MVC helped us isolate errors to a specific component as compared to the Single System Approach where the Controllers and Views are merged together.

We averted the issue of having cyclic dependencies in our system by creating abstractions that both the Controllers and Views were made to depend on, which also helped our application in terms of extensibility as we could just create new Views and Controllers and make them inherit from the correct abstractions and we would be able to expect how they will behave when the application is run.

One thing you will notice about our design is that even though the MVC architecture is applied, there are only packages for the Views and Controllers. The reason for this is that we assume the API and the database server are actually the Model component, which makes sense as we are creating a system that conforms to the Client-Server Architecture, because we can have many instances of our application (client) requesting service or information from the database server (server). The provided web service/API is the true component that “manages the behaviour and data” and “responds to requests… (and) instructions” concerning its states, according to the lectures. Furthermore, we designed our application with scalability in mind, where we expect that there could be many users using the application (and hence the server) concurrently in a multi-user system. Therefore, we deemed that it was not very feasible to create Model objects in our application because the server data would be constantly changing and having a Model object would mean that the application components would depend on that object, even if it does not have the most updated information for the application to work with. For example, we could place monitored bids for a tutor into an array of bid objects after acquiring the data from the API. However, the tutor might later click into a bid and try to respond to it while in reality, the bid has been bought out by another tutor. Thus, there will be conflict and errors as the tutor tries to submit their response to a closed-down bid. In our implementation, we want to avoid these sorts of “invalid states” as much as possible to make the application more friendly towards users through the avoiding of errors.

One big drawback about using the MVC architecture was that our code became more complex with extra levels of indirection as we needed more and more controller classes for each specific functionality in the application. This sometimes caused confusion when we were trying to figure out the appropriate Controllers to subscribe each View to.

Furthermore, it also became a little difficult to understand or immediately grasp the flow of the application because of the abstractions that were used. Even though the abstractions allowed for great extensibility, they made the dependencies between classes less clear because all the dependencies that are needed for proper functioning are injected at runtime. Hence, MVC also increased the event-driven nature of the code, as it grew to become a more dynamic system that we had to read and understand each other's code to be really able to comprehend the exact responsibility of each component in the system.

Another disadvantage of using the MVC architecture is that the performance of the application could be affected. Although the Views are only subscribed to the Controller that they are concerned with in an effort to reduce unnecessary polling for other unrelated Views, each subscriber View has to make API calls whenever the Controller is triggered. If frequent updates are made to the Model, there will be multiple API requests to the server for updates even though that View may not even be accessed by the user. In comparison to the alternative of rendering each page as it is called and not updating every other component that is concerned, the MVC architecture could definitely have some drawbacks in terms of performance.

When integrating the MVC architecture into our design, we weighed out the advantages and disadvantages, and decided that the benefits far outweighed the trade-offs because of the maintainability and extensibility that MVC offers. Even though there was a steep developmental curve, we believed that using the MVC architecture was the right step for our application because of the reasons stated above.

Refactoring Performed

Moving from assignment 2 to assignment 3, we performed some refactoring on our code. The most prominent one was the merging of classes that had similar functionality but different concrete dependencies. By doing so, we managed to merge quite a few of the Controller classes into a single LinkController class and inject the dependency that it needs into the constructor so that there is no need to repeat code for multiple classes that have the same functionality but different dependencies. Initially, we performed the extraction of superclass but soon realised that the superclass had almost no differences with the subclasses so we then used the inline class refactoring method to get our final LinkController class. This allowed us to reduce repeated code and hence the complexity of our system, with less convoluted packages. Furthermore, we performed this refactoring to help us better keep track of dependencies between components in an effort to make the system more maintainable.

If we had not performed this refactoring, we would have ended up with more and more controller classes that did not differ a lot with existing controllers. Hence, merging similar controller classes and injecting the dependency they needed instead of creating a new class for each new controller has certainly helped increase the maintainability and readability of our system. Some might argue that by doing this, we are unable to tweak the specific implementation for a similar controller that needs additional lines of code, but then, we could just create the Controller class for it and extend it from the LinkController that we already have and still have less complexity in our packages than before. Thus, there were practically no disadvantages to this refactoring, which is why we chose to do it.

Another simple refactoring that was performed is the extraction of long methods to improve readability and isolate the independent parts of our code so that errors are likely.

Other Design Decisions

In this section, we discuss certain design decisions to our application that we felt were suitable in terms of the business logic and functioning of the system.

In the previous assignment, a remark was made about our program having too many View classes. We would like to address this concern that, even though we do have quite a number of Views, the reason behind this decision was that our application is meant to accommodate an additional type of user that is both a student and a tutor. Because of this, there were many subtle differences that disallowed the merging of Views that look very similar in the application GUI. An example of this is the retrieval of information from the API, as students are more concerned with bids and tutors are more concerned with messages (which is the way that they respond to a student's bid). By merging the Views in that way, we could end up with quite a lot of views that use switch statements and would have the responsibilities of both showing a student's bids as well as a tutor's message. Hence, the Single Responsibility Principle might be breached and doing this might create God classes. This is especially true because we would possibly need another conditional block for student tutors (as we mentioned above), and in the case where even more types of users are introduced. Hence, our design decision of keeping the Views separate has helped us achieve Separation of Concerns and has made the overall application more extensible.

For the creation of bids and the renewal of contracts, we decided to only allow such an action when the student has less than 5 active bids, contracts as well as pending contract renewals because active bids and pending contracts could possibly become active contracts themselves. This is because a student can hold up to 5 active contracts, we do not want them to be able to create bids or renew contracts that might bring the system into an invalid state. An example of this is when a student creates an open bid while having 5 active contracts. It would not be fair to tutors who are searching for a student to teach to find a bid that they cannot even create a contract with. Hence, by filtering the actions that a student can take, we promote a better environment for other users and potentially even fight off internet trolls that will make many invalid bids to annoy other tutors on the platform. In conclusion, we believe that prevention is better than cure when it comes to keeping the invariant of limiting each student to only 5 active contracts.

References

- N. Nazar, "Software Design Patterns," 5-Apr-2021.
- N. Nazar, "Software Design Patterns - II," 12-Apr-2021.
- N. Nazar, "Software Architecture - MVC," 19-Apr-2021.
- N. Nazar, "Object Oriented Design Principles," 15-Mar-2021.
- N. Nazar, "Object Oriented Design Principles - II," 22-Mar-2021.