Design Rationale

Software Architectural Pattern: Active Model View Controller(MVC)

Classes Involved: TutorBids ,DashboardView, Observer and Controller

Here TutorBids is the Model as it acts as a gateway for all the concrete implementations of different features of the system. The DashboardView class is the View since it also acts as a gateway to initialize the UI components of other parts of the system that the instances of Tutor class want to interact with. Since there is a time trigger feature of refreshing the tutor dashboard after every N seconds we had to make use of the observer pattern. So we made Controller a subclass of Observer and initialized an instance of Observer in the model class . This way we avoided acyclic dependency since we could register the Controller as an observer in the model and then call the update (overridden in Controller) to implement this feature.The Tutor class acts as the observer in this case and is associated with the DashboardView. To enforce simplicity we combined the observable and model in one class which is TutorBids.

This is not exactly active MVC architecture since we are using Model, View and Controller classes instead of packages and also since Model and View in our system acts as gateways. However, even though we do not have distinct packages, we can still simulate MVC behaviour.

The advantage of implementing this in our system for the tutors' dashboard is the separation of concern. As the different components (View,Model and Controller)are separated there's ease of change [3].Any new extension would be restricted to the View component since the model does not depend on view adding new view components is easier. The disadvantage of our system is that we didn't follow mvc for the whole application but just part of it. So, overall maintaining the classes or extending the classes other than the ones following mvc would be a little tricky and complicated.

Behavioural Pattern : Template Method

Classes involved : Contract Renewal, SameTutorSameConditions, DifferentTutorSameConditions and SameTutorDifferentConditions.

Here the abstract Contract Renewal class is the superclass which asserts the algorithm for contract renewal under different contexts which are represented by SameTutorSameConditions, DifferentTutorSameConditions and SameTutorDifferentConditions classes[1]. This pattern is used because all three subclasses have very similar responsibilities but with minor differences which in our case is the contract renewal with either same or different tutor and same or different conditions[1]. Plus, the ViewLatestContract class only needs to depend on the abstract ContractRenwal class instead of depending on all 3 subclasses.So, the first advantage we gain here is that we do not need to change our client classes as the superclass could decide under which context a particular message would go. The second advantage is that we do not have a bloated method or a class with numerous methods in it to account for each contract renewal subclass contexts but instead have a clean design to account for all of them. Lastly we made sure the higher level module (ViewLatestContract) does not need to depend on lower level modules like (SameTutorSameConditions etc) and can instead depend on abstraction[1]. The disadvantage here is that we violate the Liskov Substitution Principle since each subclass depends on a very specific context of contract renewal and so are not interchangeable.

Package Cohesion Principles: The Release Reuse Equivalency Principle(RREP)

Our package has reusable classes – GUIAction, GUIContext, ViewContractAction etc all of which contain concrete implementations used by other classes. This means that we do not have duplicate lines of code doing the same thing in other classes but reuse existing ones[9] . Classes that are unrelated to the purpose of the package are not included[9] . This principle is chosen because in a complex system like ours, we expect to have high associations and polymorphism and so having reusable components facilitate that process.

Package Coupling Principles: Acyclic Dependency Principle(ADP)

When we were trying to implement the tutors dashboard we wanted to use passive mvc. But later we realised we had a time triggered method in the model that the controller needed to keep track of. So we had to find a way to find a way to call a method from the controller to update the view in the dashboard from the model. If we were to do that directly then there would be a cyclic dependency between the model (TutorBids) and the controller (Controller). So, to avoid that we implemented the active mvc which makes use of the observer pattern to allow no cycles to form by creating an interface in between[10]. The advantage of following ADP is simple: the system will become more coupled and harder to maintain since the developer wouldn't know which class to start from[10]. There's no strong disadvantage to this approach except that we need to create a new class.

Refactoring Techniques

The createContract method inside the OpenBidAction class was too long[1]. Since this method is an important one, in order to improve readability, we used **Extract Method** which isolates the sections of code which deal with retrieving bid offer data from other classes of the system in the creatOffer method which is now called to get offer data[4]. This has improved the readability of the class but on the other hand has increased the dependency between the methods as a single change may be propagated to the other methods.

The **Extract Method** refactoring technique is used to split the showSubDetails method in CloseBidAction class[4]. The section of code in showSubDetails concerned with the details about teaching arrangements in a close bid, is moved to the showLessonDetails method. This is done so we are able to isolate specific UI texts, labels and messages which in turn improve the readability of our UI code logic[4].

The **Extract Method** refactoring technique is used in MakeOpenBidOffer class to isolate UI specific code from the constructor to the showUI method[4]. This is done so that all UI responsibility is placed on showUI and the constructor can now only deal with creating the bid offer instance. Furthermore the sections of code to create the bid message and for posting the message are fragmented into their own methods: createMessage and postMessage methods. This improves overall readability as well as frees the built in actionPerformed method to only handle button clicks without having to worry about the logic that needs to be executed. Initially, the postMessage method also handled calls to the web api but the GUIAction interface already has concrete implementation to do just this. So we removed the code to call the api in postMessage method and replaced it with a call to updateWebApi method in GUIAction. This is done to remove repeated code from the system as having repeated code raises the risk of having to fix numerous codes after a bug is found[4].

The **Extract Method** refactoring technique is used in Student class where the notificationUI method is created to display the notifications about signing contract and contract expiry to the user. This is done to improve readability of the overall class and this method can also be further extended for showing more notifications[4]. The implementation for making open bid offers have remained relatively similar and this is due to the design decisions made in the last milestone where through UserFacade, Student was made to handle making offers as one of their complex responsibilities. The only change is that we used the **Extract Method** to make the createOpenBidOffer method to help create the bid object before it is stored in the database and isolate bid creation functionality for better readability of the code[4]. As for checking expired contracts the helper method checkExpriyDate is made which simply checks if the contract is expiring and triggers the notificationUI method. Overall the Student class makes use of multiple methods with proper cohesion between the methods making it flexible for further changes and so adheres to **Open-Close Principle**[5].

Previously, the createContractAction class had two very distinct responsibilities which were to create a contract between student and tutor and to show contracts that have not been signed by the student. This violates the Single Responsibility Principle[6] and so the **Extract Class** method is used to create the ViewContractsToSignAction. The fields, UI components and methods to show unsigned contracts have been moved to ViewContractsToSignAction class[4]. The advantage here is that we adhere to the **Single Responsibility Principle** and so all components to consider are already there. The disadvantage of doing this is that the coupling between the classes have increased as both Student and createContractAction can trigger the ViewContractsToSignAction class.

The ViewContractsToSignAction class has a method to update a contract that uses PATCH and this is the only place where PATCH is needed. However, the GUIAction class is responsible for handling communication between the classes and the API and so to be consistent with the existing design that is the Strategy Behavioural Pattern, the patchWebApi method is implemented in GUIAction and called from there. This also means that if any other classes need to be extended so that they can PATCH data they can reuse the patchWebApi method in GUIAction. Therefore this design decision adheres to the **Open-Close Principle[5]**.

Previously the built in actionPerformed method in ViewBidOfferAction class, contained all the functionality of this class. This made it bloated, difficult to trace through and since such methods depend on button clicks there were numerous conditional blocks of code[2]. Therefore a long method with lots of if-else statements constitutes poor readability. This problem is solved using the **Extract Method** refactoring technique[4] where the if-else statements are split into individual methods therefore isolating each conditional block of code. These are:

1. showAllBidViewDetails method to show the messages sent on the bid which was previously in the first if block
2. showOpenBidOfferDetails method to show the bid offer details made by the tutor
3. closeOpenBid method to allow student to sign off on a bid while providing a contract duration
4. A refactored built in actionPerformed method with fewer conditional blocks which now only call the three new methods above.

The advantage of using the **Extract Method** here is that now we have improved code readability as well as isolated only specific independent sections of codes which helps us focus on a single aspect of the code at a time[4]. The disadvantage here is that now there are lots of methods making the overall class bigger than the desired size.

Previously, similar to ViewBidOfferAction class, the built in actionPerformed method in ViewMessages class, was bloated, difficult to trace through and since such methods depend on button clicks there were numerous conditional blocks of code. Therefore a long method with lots of if-else statements constitutes poor readability[2]. This problem is solved using the **Extract Method** refactoring technique where the if-else statements are split into individual methods therefore isolating each conditional block of code[4]. These are:

1. selectedTutorMessages method to show the messages sent on the bid which was previously in the first if block
2. storeMessage method to store the message from the user into the database
3. selectTutor method for student to select a tutor and close bid
4. A refactored built in actionPerformed method with fewer conditional blocks which now only call the three new methods above.

The advantage of using the **Extract Method** here is that now we have improved code readability as well as isolated only specific independent sections of codes which helps us focus on a single aspect of the code at a time[4]. The disadvantage here is that now there are lots of methods making the overall class bigger than the desired size.

The **Self Encapsulate Field** refactoring technique is used to get the private tutorId field through a getter method. This is done because we want to access the private tutorId field and do not want to change it. The advantage of doing this is that there would be no indirect access to this field which might let users change the field value and see other user's information.The **Extract Method** refactoring technique is also used in this class. The nested if-else block inside the actionPerformed method is extracted to the viewOfferMessages method. Furthermore the section of code to show the bid messages as a list is extracted into the getMessageDetails method. The Extract Method is used here to isolate sections of code that can exist independently which would enhance readability[4]. The disadvantage here again is that the class would now have too many methods with coupling in between them.

The **Extract Method** refactoring technique is used in the viewRequestAction class. The section of code to put details about an active request is extracted from the showAllStudentRequests method to the putOpenRequests method. The nested if-else block of code in the actionPerformed is extracted to the shoeReqDetail method. This is done because both the extracted sections of code can exist on their own and so are isolated. This in turn improves the readability of the class but on the other hand the size of the class is not still shortened, so we still have a very long class.

The very first implementation of the viewContractAction class had a single method, showFinalizedContracts() which was 121 lines long and it was to show the signed contracts of the student. The showFinalizedContracts() was a bloated method due to its length which is bad design since there were sections of the method that can exist independently[4]. Moreover this made readability difficult. The **Extract Method** refactoring technique is used to split the large showFinalizedContracts() method into five independent methods each concerned with a specific task[4]. For a given node:

1. getStudentAndTutorNames() method would provide the details of the two parties
2. getcontractDetails() method provides the agreed contract details between the two parties
3. showFinalizedContracts() method simply displays the data received from the previous two methods in the UI.
4. contractNotification() method checks if a contract is about to expire soon or not
5. and expiryNotification() method adds an expiry alert on each contract.

The first three methods are where we applied the Extract refactoring method to separate the data retrieval methods - getStudentAndTutorNames() and getcontractDetails(), from showFinalizedContracts() . Therefore we were able to isolate independent sections of the previously bloated showFinalizedContracts() method into three separate methods which in turn also improved readability[4]. The fourth and fifth points prove the advantage of using the Extract method because with the refactored design we could easily extend the class to include two new methods for contract expiry notifications.

Furthermore since showFinalizedContracts() displays the data it received from getStudentAndTutorNames(), getcontractDetails() and expiryNotification() methods, the **Extract Variable** refactoring technique is used to place the complex return from the methods into readable variables[8]. This further improved the readability of the code. The use of the Extract Method caused our class to have 5 methods in total which puts too much responsibility on the viewContractAction class. The use of the Extract Variable refactoring technique means that the viewContractAction class has lots of local variables in it which increases the overall length of the class and makes it cluttered[8].

References:

[1] Template Method

[2] Week 9 lecture slide 13

[3] Week 7 lecture slide 42

[4] Week 9 lecture slide19 (Extract Method)

[5] Week 3 lecture slides 20-24 (OCP)

[6] Week 3 lecture slides 16-19 (SRP)

[7] Week 9 lecture slide 27

[8] Week 9 lecture slide 21(Extract Variable)

[9] Week 4 lecture slides 8-10 (RREP)

[10] Week 4 lecture slides 19-24 (ADP)