RATIONALE

The Common Reuse Principle (CRP): We have attempted to follow this principle to the greatest extent. For instance, in the LOGIC package. All the classes are in one way or another related to each other, mostly through redirects, therefore, if one controller class is being used, all the controller classes have to be used.

SOLID principles

Note: In this assignment, we have attempted to follow all the solid principles. However, we have put a heavier emphasis on the single responsibility and open closed principles for the sake of maintainability of the system.

Single Responsibility Principle (SRP): To abide by the SRP, we have attempted to divide each class into reasonably smaller classes to balance out the complexity of the system and the responsibilities of each class. For instance, in the LOGIC package, we have a class "CheckBidValidity" that is responsible for checking the validity of the bid that is being printed. Although this could have been implemented in the classes that have dependencies with this class, the choice was to make it a separate class as it is a large responsibility that could potentially require to be reused. We have also used abstract classes for users and bids in order to separate the potential difference in functionalities for the different bids. This way, different bid subclasses may have slightly different responsibilities, while the abstract class may be able to handle the common behaviours.

***Open Closed Principle (OCP)*:** The main way we have tried to abide by the open/closed principle is by introducing abstraction into our design. We have several interfaces, such as ViewOffersControllerInterface and/or Observer interface, and through the use of abstract classes. These may hold abstract methods, which in turn, allow us to extend the system with more classes that may have similar functionalities by implementing these interfaces or extending the classes.

Patterns used:

***Adapter*:** The adapter pattern is used to convert JSON data into classes and class data into JSON. Specifically, for this, we have created a JSONAdapter utility class in the API package. This choice was made to ensure that each class follows the single responsibility principle. For instance, without the adapter class, to convert data about the user from REST api into the user class instance, we would need the method to do so in the user class. This breaks the single responsibility principle since the User does not necessarily need to be responsible for converting data.

Factory pattern: In our design we have used 5 factory patterns. For construction of User object, bids, main page of the application, the controllers of the mainpage, and lastly, the ViewBidsControllers. We found the use of factory methods here very helpful in order to support the open/closed principle and make the application more extensible by introducing factory classes where new functionality can be added by allowing extending it to be able to create more concrete objects. Not to mention that it also helps us to support the single responsibility principle (SRP), by allowing us by separating the responsibility of the product

creation from the client code into a separate class. Having said that, we at times, have modified the factory methods by removing the concrete factory methods. This is mostly due to the fact that, upon our analysis some of the functionality was unlikely to extend to the point where the single factory class would have too many dependencies to the classes it is responsible for creating. Therefore, it was in our interest to reduce the complexity of the system by removing the concrete factory classes and have one factory class responsible for creating the concrete objects.

Observer: In our design, we have used observer pattern to establish a subscription mechanism between the JSONAdapter that acts as a publisher and Closed bid, bids, messages and contracts that act as a subscriber.[1] This behavioural pattern keeps the subscribers notified at runtime and also makes sure that the JSONAdapter can subscribe subscribers at any time and implement additional update functions without breaking client code with the help of Observer Interface that is implemented by subscribers. For instance, we use this functionality to update the model package class's data during runtime, but listening to method calls that override data in the REST api. This way we know when and what data we need to change in the model. We have found this design pattern very useful to support Open/Closed Principle as you can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface). It also helps us follow a single responsibility principle by having a separate responsibility; Publisher to notify subscribers and invoke update actions when some change is observed and subscriber to call update function.

Singleton: In our design we have used a singleton pattern to make sure that the class with singleton has only one instance and this instance holds the access point globally and keeps the content cached.[2] We have used this singleton pattern for JSONAdapter, SubjectCollection and UserCollection to make sure that it is globally accessed and keep the cache content. This breaks the single responsibility principle since it controls their own creation and life cycle of the instance. However, this is useful to implement observer pattern and some caching as it will remove the possibility of creating more than one instance of these classes and introducing bugs. Hence we had to compensate for this violation as it would in turn make our system more extensible.

MVC: We have also attempted to use the Model View Controller architecture by separating the responsibilities of having a package of classes responsible for holding data (Model) which we have named "APP", a package responsible for the GUI aspects of the system (View), named "GUI" and a separate package responsible for the interaction between the view and the model packages (Controller), which we have called "LOGIC". By using this architecture, we have hoped to increase the extensibility of our application by separating the responsibilities of holding data, rendering the UI on multiple page views and performing the logic on UI using the data. This allows us to follow the single responsibility principle and hence would help us extend the system in the future. Furthermore, we began to use the MVC architecture in hopes of removing the cyclic dependency between the View package and the Model package by introducing a Controller package in between that would be responsible for updating the view with the data from the model. However, using python's tkinter we were not able to succeed in that goal, as the GUI page needs to operate on an

instance of a controller, which causes cyclic dependency between Controller package and View package.

ASSIGNMENT 3

Package principles used:

The Common Reuse Principle and Common Closure Principle:

In order to follow these package principles, we have attempted to maximize package cohesion while minimizing coupling between classes in different packages. In assignment 3 specifically, we have used the mediator pattern to reduce dependencies between classes from the LOGIC package and the classes in the APP package. By doing so, we have effectively increased the LOGIC package cohesion since instead of depending on Collection classes in the APP package they now depend on the Mediator class in the LOGIC package. This allows us to reduce the number of coupling that we have between the LOGIC and APP packages since the mediator package is the only class in the LOGIC package that is coupled with the collection classes from the APP package. Hence, by doing so, we have packed classes that are reused together, into one package.

At the same time, we have tried to follow the common closure principle by separating classes into different packages. This way, we are making sure that classes that aren't used together, are not packed together. For instance, no classes in the LOGIC package directly depend on the subject class in the APP package. Hence, they are not reused together, and are therefore packed in different packages. All in all, we have attempted to find a balance between CRP and CCP by separating classes into different packages that have different responsibilities. While doing so, we have also attempted to increase cohesion and minimize coupling in each package in order to follow the CRP for each package that we have created.

Patterns used:

***State**:* This is a new pattern that we have made use of in the third assignment. We have used the state pattern on contracts. Using this pattern allows us to extend the system easier in the future. This is because by using the state pattern we can have separate states for contract objects that are their own separate objects. We can also allow the contract to change its state based on some variables. Whenever we need the contract to do something that is related to its state (in our case, it to return the correct notification for the user), we do not have to have a lot of if statements in classes that utilise the contract's state, but rather we can just call the state's doState() method. Adding another state to contracts will be very easy since we would simply have to add one more state class that returns its appropriate state's notification. Hence, we would not have to change any implementations in the code that make use of the state.

Observer: We have updated the observer pattern to include contracts. Whenever a contract is renewed by the user, using the observer pattern we can know when the data in the api has been changed, in order to update the data in our model.

Mediator: We have refactored the LOGIC package to use a mediator pattern that is used by the classes in LOGIC package to access collection objects from the APP package. More specifically UserCollection and SubjectCollection. This allows us to increase package cohesion in the LOGIC package and reduce coupling of the LOGIC package with the APP package as classes from the LOGIC package will depend on the mediator class rather than specific objects in the APP package. This also allows us to increase our extensibility since the access of the classes from LOGIC package to those in the APP package will be through the mediator. Hence, to change or add to the implementation, changes will be limited to one class in the LOGIC package. Not to mention that any changes made to the Collection classes in the APP package will no longer impact the classes in the LOGIC package directly. Hence, any changes that need to be made in the LOGIC package, will only have to be made to the mediator class.

Refactoring to prevent dependencies:

While refactoring we have mainly applied the refactoring technique “Extra Class” in order to follow the single responsibility principle and by that also allow our system to be more extensible. The refactors that we have done by following this technique are mentioned below under “Single responsibility principle”, as this is one of the main benefits this refactoring technique aims to achieve.

Single responsibility Principle:

- We have created a class for creating contracts to reduce dependencies to JSONADAPTER and abide by the single responsibility principle stricter. This also increases package cohesion as less classes in the package have to make calls to JSONAdapter, located in another package, and instead make calls to the CreateContract class in that same package. Then, controller classes that need to create a contract, delegate the task to the ‘CreateContract’ class. This also allows for better extensibility since changing or extending the implementation of creating contracts in the JSONAdapter will be easier as it depends on a smaller number of classes in the LOGIC package.

- Single class for Refreshing the tutor’s monitor page. This class is responsible for calling the JSONAdapter and repopulating the changed bids and offers on the tutor’s subscribed bids. This separation was done to follow the single responsibility principle. This was initially done in the TutorViewMonitorController class, however, by following the “Extra class” refactoring technique, we have separated the two into two classes as these are two separate responsibilities. Thereby adhering to the single responsibility principle.

- We have refactored the observer pattern. Instead of having it combined with the JSONAdapter, the observer class is now a separate class. This was done in order to

allow for single responsibility principle, to ensure a better extensibility of the system and to ensure that the JSONAdapter class does not become a bloater as the system is extended. If the system were to be extended so that it may require more objects to be subscribers to the observer class, it would be easier to extend as we have separated our observer class based on the type of subscribers. At the moment we have the observer class that is meant to observe collection classes. Extending the system to have observer subscribers other than the collection classes, would be easy since we would simply need to make a new observer class and associate it to the JSONAdapter. As mentioned, it would also help to prevent JSONAdapter from becoming bloated as the system extends. This is because by separating the observer, if the observer were to be extended, it would not affect the JSONAdapter class as the changes would be made in the specific observer class. New observer classes can be created and associated with the adapter class. Hence, only increasing the dependencies from the adapter to the observer classes in the same package, but never disobeying the single responsibility principle.

Open Closed Principle:

- In the third assignment, we have paid close attention to follow the Open closed principle mainly when creating contract states. Knowing that Contracts might have different states we may need to perform the contract's actions based on the state that it is in. In order to make the system more extensible we have used an Interface "ContractState" which every state class should implement. This way, if , when extending the system, the contracts need to have more states than they do at the current state. It will be very simple to extend as we would simply be able to add the contract state class, and implement its behaviours. No other classes would have to be touched. Therefore, since new functionalities can be added without modifying the existing code. This follows the open closed principle.

Dependency Inversion Principle:

- Similarly, for contract states, by adding the interface "ContractState", we allow the Contract class to depend on abstraction rather than on concrete objects. This allows. This too helps us with extending the system in the future since any changes to the state objects may break the contract object and vice versa. In other words, changing contract states without the presence of the ContractState interface may cause us to have to change code in the Contract object. By using the inversion principle. We can ensure that by changing any implementation of a contract state, we would not have to modify existing code in the Contract object.

Reference:

[1]: Unknown Author, Observer Pattern, Retrieved May 02, 2021, from https://refactoring.guru/design-patterns/observer

[2]: Unknown Author, Singleton Pattern, Retrieved May 02, 2021, from https://refactoring.guru/design-patterns/singleton

[3]: "What is MVC? Advantages and Disadvantages of MVC - Interserver Tips," *Interserver Tips*, 2016. https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/.