Lim Jin Yung (29036186), Chew Shen Min (29640938)

FIT3077 Assignment 2

Design Rationale

Observer Pattern
Given that an open/close bid will close once the time for bidders to bid has ended, the observer pattern is applicable for us to implement into the system because it is simple for us to attach a *BidObject* to *BidObserver* once the bid is created, and detach the *BidObject* when the time for bidding has ended. This pattern offers a high degree of independence between the *BidObject* and *BidObserver* that orient themselves according to the state of *BidObject*. The *BidObserver* will not need to know any information about the *BidObject* since the interaction is completed regardless of the *BidObserver*

Common Closure Principle
As there are different routes/URLs defined for different purposes, some routes are for users (for example login and logout), some routes are for bids (for example create bid, offer_bid, close bid). If all the routes are packed into only one route file, for example *routes.py*, then it would be hard to maintain and refactor. Hence, to increase the maintainability and easy to scale the project, routes and related files are placed in a package and imported by the app. By applying this principle, the number of components that are affected when we change our software is kept to a minimum because we've already split the routes into different packages.

Open/Closed Principle
The Open/Closed Principle is implemented in the methods like *user_profile_details*, *check_bid_status* and many more. These methods are closed to modification, but they can still be used and extended by other methods. Implementing this principle made the system more resilient to change as it allows methods to communicate which are unlikely to modify. Since new functionality will be introduced in the future, resilience to change is critical for this app where we need reliable methods with fixed implementations that cannot be changed.

Single Responsibility Principle
Functions like *get_user_competencies*, *user_subject* and many more are all applying single responsibility principle as these functions are easily interchangeable and mixable depending on the task, allowing the code to be easily extended. The code is DRY and not repeated, hence, if we want to make any modifications in one of these functions, we will only need to modify it in one place. By applying this principle, the app will be clean, organized and easy to understand.

Chain of Responsibility

One of the examples in the code that implements the chain of responsibility is when a requestor chooses an offer, the request will be passed along to a chain of methods, to find the bid object from *BidObserver*, detach the bid, close the bid, and then generate the contract for both parties. By applying this design, it simplifies the *BidObserver* as it doesn't have to know the chain's structure and constantly provides reference to other methods. It also allows the system to add/remove the responsibility dynamically by modifying the methods or the order of the chain.

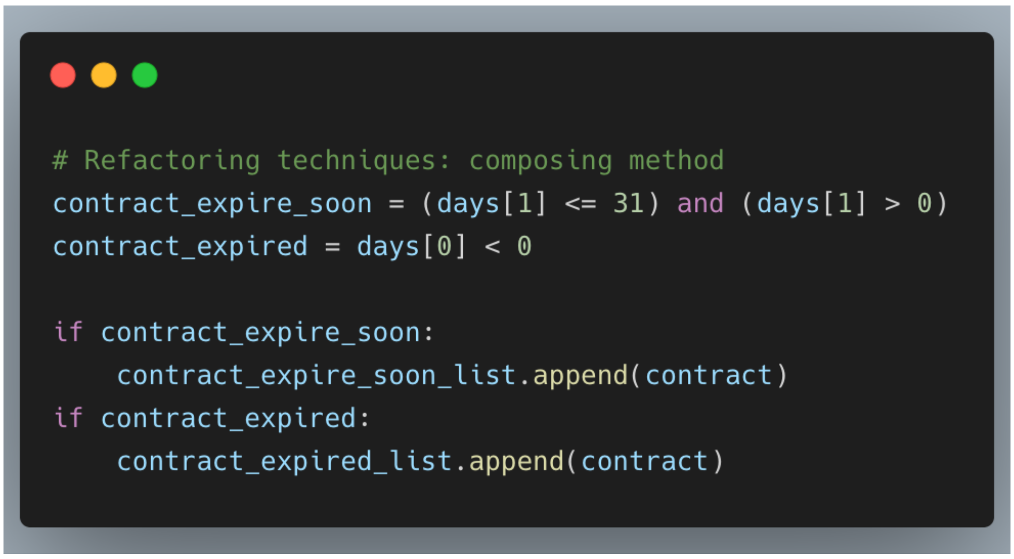
MVC

The project is using MVC architecture, where the project has a *templates* package as View, *models* package as Model and *users, bids, main,* and *contract* package as Controller. This makes the project have a clear separation of the UI logic, input logic and business logic. Using MVC, it provides clean separation of concerns, which makes the development of the various components can be performed parallelly as long as the team agrees the input and output accordingly. However, the downside of using MVC is also obvious when the team is refactoring the code from Assignment 2 to Assignment 3 where there are a lot of codes in the Controller need to be change/modify to update the project

Refactoring Methods

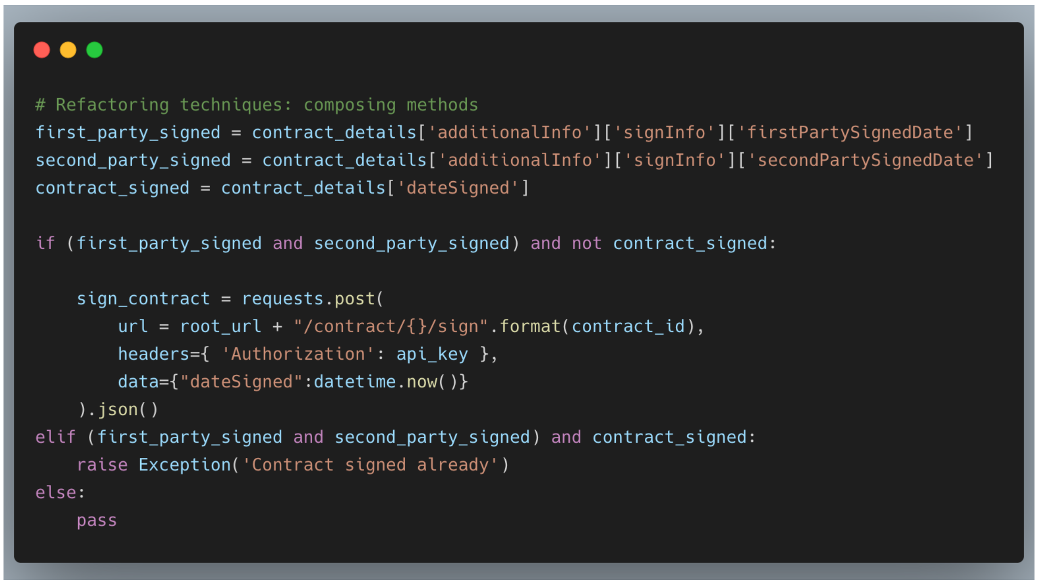
Composing method: extract variable

Function *check_contract_expire_soon* in *online_matching_system/contract/utils.py*



```
# Refactoring techniques: composing method
contract_expire_soon = (days[1] <= 31) and (days[1] > 0)
contract_expired = days[0] < 0

if contract_expire_soon:
    contract_expire_soon_list.append(contract)
if contract_expired:
    contract_expired_list.append(contract)
```

Function *check_both_parties_signed* in *online_matching_system/contract/routes.pu*

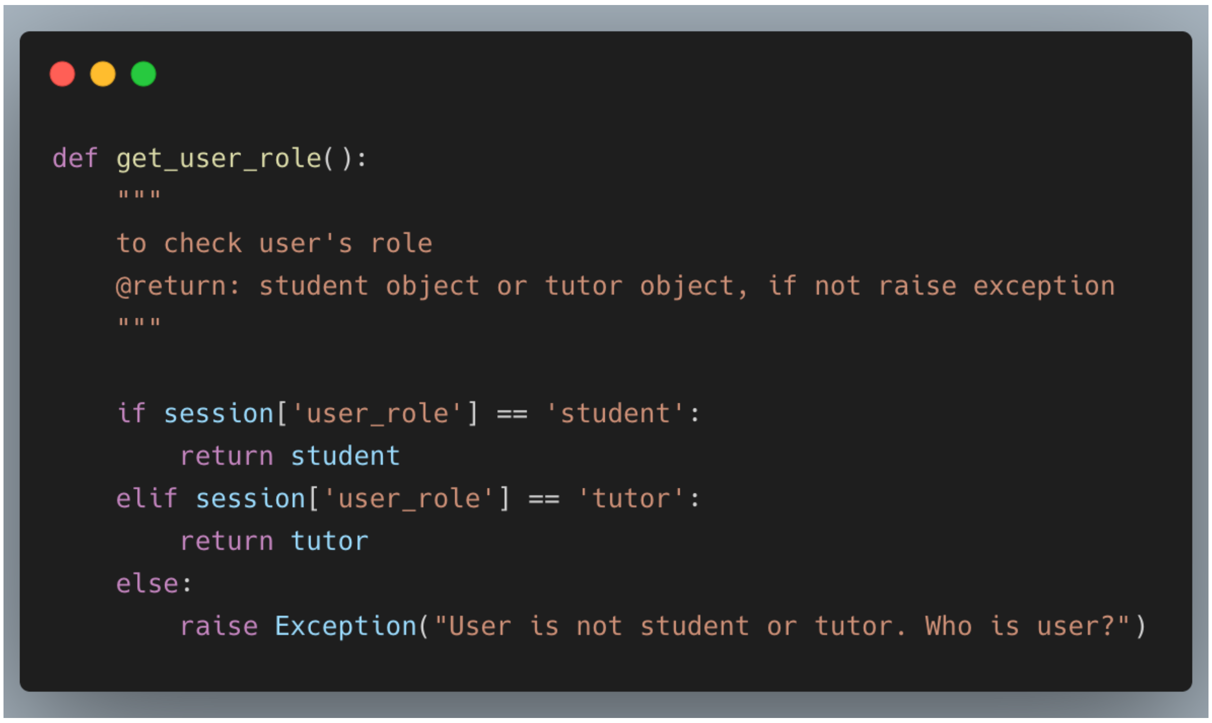


```python
# Refactoring techniques: composing methods
first_party_signed = contract_details['additionalInfo']['signInfo']['firstPartySignedDate']
second_party_signed = contract_details['additionalInfo']['signInfo']['secondPartySignedDate']
contract_signed = contract_details['dateSigned']

if (first_party_signed and second_party_signed) and not contract_signed:

    sign_contract = requests.post(
        url = root_url + "/contract/{}/sign".format(contract_id),
        headers={ 'Authorization': api_key },
        data={"dateSigned":datetime.now()}
    ).json()
elif (first_party_signed and second_party_signed) and contract_signed:
    raise Exception('Contract signed already')
else:
    pass
```

Composing method: replace temp with query

Function *get_user_role* in *online_matching_system/users/utils.py*

get_user_role is a function that checks the user's role and returns either a StudentModel or a TutorModel object.



```python
def get_user_role():
    """
    to check user's role
    @return: student object or tutor object, if not raise exception
    """

    if session['user_role'] == 'student':
        return student
    elif session['user_role'] == 'tutor':
        return tutor
    else:
        raise Exception("User is not student or tutor. Who is user?")
```

Then other functions can easily access the user's role by querying the method and then access the user's details and other information like bids, subject, competencies etc.

Composing method: moving features between package

Function *check_contract_expire_soon* in in *online_matching_system/users/utils.py*

This function should be placed in the contract package originally since it was related to the contract, but since it was used more often by the user package, to eliminate code smell, it has been moved to the user package.

Reference

1. *Observer pattern: what does the observer design pattern do?* IONOS Digitalguide. (n.d.). https://www.ionos.com/digitalguide/websites/web-development/what-is-the-observer-pattern/
2. Zubova, A. (2019, September 21). SOLID Programming (Part 1): Single Responsibility Principle. Retrieved from https://dev.to/annalara/solid-programming-part-1-single-responsibility-principle-1ki6
3. Design Patterns and Refactoring. (n.d.). Retrieved from https://sourcemaking.com/design_patterns/chain_of_responsibility
4. MVC Tutorial for Beginners: What is, Architecture & Example. (n.d) Retrieved from https://www.guru99.com/mvc-tutorial.html