Design rational

Facade pattern

In the previous assignment, we used the bridge design pattern between the GUI class and service class. In this assignment, we decided to re-use the facade design pattern in our code. We have a UserFacade and AdminFacade class which interact with the services classes. They provide methods for the GUI class. By using the facade pattern, the classes in the view which control the display on GUI don't need to know the complex logic or code in the Service class. They can just use the methods which are provided in the facade class. It decouples a front end from the complex Services classes which will be a lot easier for development. For example, when the AdminEditBookings class wants to modify a booking, it doesn't need to know how to interact with the BookingService or notify the admin. It just needs to know, there is a method in adminFacade that can be used for. However, by using this pattern, the facade class will violate the Single Responsibility Principle. It will increase the risk of maintaining and extending our system.

Observer pattern

In the assignment specification, all the admins will get a notification if any booking is modified which has the testing site as their working site. In this case, we think that the observer pattern will be useful. In our implementation, the admin will be the subscribers, which subscribe on the FacilityPublisher. Once there is a booking modified related to a facility, then the facilityPublisher will send the notification to all the subscribers. By using this pattern, we can easily add more subscribers or remove to the publisher. The pattern makes the system more extensible, which applies the open closed Principle which will be discussed more in detail below. And also, it separates the publisher and subscriber, the change on one of them will not affect the other one. However, using these patterns increases the complexity of the system. Another disadvantage of the observer pattern is that subscribers are notified in random order.

MVC architecture pattern

In this assignment, we apply the mvc architecture pattern in our code. In our design, view classes will provide interfaces for users to interact with the controller component to operate the data in the model. The view and controller are separate, which apply the acyclic dependencies principle and this will be discussed in more detail below. Another advantage of this architecture pattern is it makes the testing easier. Since it separates the view, controller and the model, we can test on them individually and detect the bugs easier. Also, it is easier for us to develop. We separate the work that one of the members handled the view part and another member handled the controller and model part. Both of us don't need to worry about how the other part works but only focus on the part we handle. It makes it easier for us to collaborate.

Open closed Principle

By implementing the Observer pattern, one of the benefits is it follows the Open closed principle. It gives us the advantage of "open for extension and close for modification". For example, in the future design, if we want to extend the functionality for subscribers, we can just add methods in the interface. And if we want to have other types of subscribers, we don't want to change the current code. What we can do is extend the subscriber interface, and let the new type concrete subscribers implement the new interface. So that it would be easy for the future extension.

The Common Closure Principle (CCP)

In the package level, we put the classes which are relevant and similar together in the same package. It increases the maintainability of the system. For example, the service component contains the classes which handle the data operation and interact with the webAPI. If there is a change in the model component, which is the component operated by the service class. Then It only affects the classes in the service component.

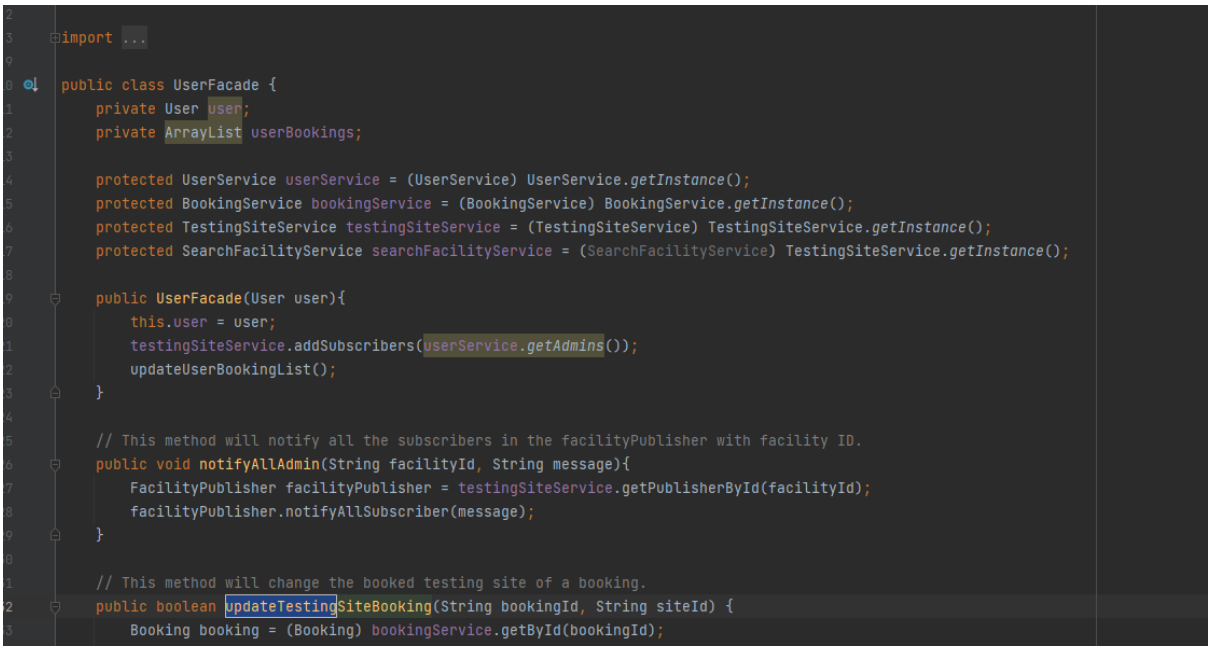
The Acyclic Dependencies Principle (ADP)

To avoid MAS (Morning After Syndrome) issues happening in the code, apply the Acyclic dependencies principle, which means no circular dependencies should exist in our code. By applying the mvc architecture pattern, the structure is very clear. The controller package depends on the view package, the service package depends on the controller package, the model package depends on the controller and view package (which needs to record the current user/admin). So there are no circular dependencies.

Refactoring

One of the biggest refactoring from assignment 2 to this assignment is, in assignment 2, the view class directly interacts with the services class. Because each service is only responsible for the operation on one model object.(For example, the bookingService is only responsible for Booking), that will cause that the method in one service class can not interact with the other service. If the view class wants to operate with multiple types of objects, then the logic has to be written in the view class. In this assignment, we are using a facade design pattern. A facade class will be the class that interacts with the services class and view class will only know about the facade class. By doing this, we can have complex logic which interacts with multiple services. It makes the structure better and increases the code readability.

We also have stored the user that has logged in, into the user controller, by doing this we're able to use the facade design pattern when passing through the user to further ensure that we have a much better structure.



```
import ...

public class UserFacade {
    private User user;
    private ArrayList userBookings;

    protected UserService userService = (UserService) UserService.getInstance();
    protected BookingService bookingService = (BookingService) BookingService.getInstance();
    protected TestingSiteService testingSiteService = (TestingSiteService) TestingSiteService.getInstance();
    protected SearchFacilityService searchFacilityService = (SearchFacilityService) TestingSiteService.getInstance();

    public UserFacade(User user){
        this.user = user;
        testingSiteService.addSubscribers(userService.getAdmins());
        updateUserBookingList();
    }

    // This method will notify all the subscribers in the facilityPublisher with facility ID.
    public void notifyAllAdmin(String facilityId, String message){
        FacilityPublisher facilityPublisher = testingSiteService.getPublisherById(facilityId);
        facilityPublisher.notifyAllSubscriber(message);
    }

    // This method will change the booked testing site of a booking.
    public boolean updateTestingSiteBooking(String bookingId, String siteId) {
        Booking booking = (Booking) bookingService.getById(bookingId);
```

Long list parameter

In the model component, the constructor of the object, for example, booking object has a long parameter list in the code in assignment 2. We refactored this by removing the long parameters list, which makes an empty parameter constructor. And then when we create a booking object, we use the set method to set the booking's attributes.

Duplicate code

In the service package, each service is using the same method to send post, patch, delete requests to the webAPI service. To follow the don't repeat yourself principle, we extract the methods in all the service classes into the baseService class, and let the service classes extend from the baseService class, so that they can use the shared methods.

We also have service classes which will allow us to retrieve certain objects depending on what we pass into it such as a booking id, this means that we are able to retrieve bookings from anywhere in the code without having to re-write all the functionality to retrieve the appropriate booking.

```java
public ArrayList getListOfSearchResults(int numOfBookingsToDisplay, String searchQuery)
{
    ArrayList<Booking> bookings = getList();
    int numberOfFoundBooking = 0;

    ArrayList<Booking> foundBookings = new ArrayList(numOfBookingsToDisplay);
    for(int i = 0; i < bookings.size(); i++)
    {
        if(numberOfFoundBooking == numOfBookingsToDisplay)
        {
            break;
        }

        if(bookings.get(i) != null)
        {
            Booking booking = bookings.get(i);
            String bookingID = booking.getBookingId();
            String bookingTime = booking.getStartTime();
            String bookingPatientName = booking.getCustomerId();
            String bookingFacility = booking.getSiteId();
            String bookingTestType = booking.getTestType();

            if(bookingID.contains(searchQuery) || bookingTime.contains(searchQuery) || bookingPatientName.contains(searchQuery) || bookingFacility.contains(searchQuery)
                    || bookingTestType.contains(searchQuery))
            {
                foundBookings.add(booking);
                numberOfFoundBooking++;
```