Design Report

MA_Lab03Team13

Note: The linking between some of the classes/packages is not shown as they are indirectly connected by the java frame.

Class Design

The Login System was initially designed to follow the Abstract Factory Design Pattern. This design was thought out on the basis that a common Login interface was available to all roles of users (Customers, Administrators/Receptionist and Healthcare Workers). Upon correct login authentication, an object based on the type (role) of the user, will be created. Using the Abstract Factory Design Pattern, would ideally be the general approach as this design pattern allows you to create families of related objects without specifying a concrete class. However, this idea was scrapped as it made the design very complicated due to a large number of interfaces and classes required in order to implement it. During this iteration the subclasses have very little data in them and are not the primary components of the whole system. They are simply used to store relevant user data and in some instances, use the data for certain features. Therefore, we decided it was best to simply use abstraction, i.e an Abstract class called AbstractUser, to create role specific users that can be instantiated.

The Login System has three features to be implemented. The login page, JWT Token validity verification and decoding the JWT Token. In order to respect the Single Responsibility Principle, the login feature was implemented in LoginPage class and the JWT related features in the JwtToken class. It was important to follow the Single Responsibility Principle as it makes the implementation easier, cleaner and it prevents unexpected side-effects of future changes.

We apply an observer design pattern for offShoreTestingSite. We have a class called offShoreTestingSiteDataSource that acts as an observable that updates information about offShoreTestingSite and listens from from web Service. Then we have another class called offShoreTestingSite that acts as an observer. Whenever there is an update(ex: new booking is added, waiting time of the facility changed) from the web service for offShoreTestingSite, offShoreTestingSiteDataSource will notify all its observers so that the observer's data is updated. The reason for us applying this design pattern here is because we don't hope that we have to keep calling a get request every single second even though there are no updates from the web service. With this design pattern, it helps to solve this problem as we only need to send a GET request whenever there is an update and it reduces the load to the server. Beside, with observer design pattern another problem is also solved as the observer only listens to the observable that it is interested to, for our case, the specific testingSite will also update its information when the particular TestingSite's data from the web service is changed. With the observer design pattern, open/closed principle is applied as we now can introduce a new observer class (if there is a different TestingSite in the future) without having to change the observable code. Moreover, observables and observers are reusable independently of each other. However the Cons of observer design pattern is the observers are notified in random order.

We also apply singleton design pattern to our systems for our local “database” (HomeBooking collection, TestingSite collection and offShoreTestingSite collection). Singleton is applied because we don't hope that we have multiple databases that save the data. With singleton, we ensure that all the databases have only a single instance throughout the whole lifecycle of the project and we will have a global access point to the databases. However singleton violates Single responsibility since the databases control how they are created (getInstance method). Besides, due to the couling by the usage of singleton, it may be difficult to do the unit test since it becomes very difficult to identify the dependency chains.

We have also applied Facade design principle to our system for onSiteBookingSystem, with facade we hide the complexity of using the subsystem behind the client (for our case onSiteBookingPage) and it help us to decouples the client from our subsystem. Without facade, when the client wants to make use of the onSiteBooking, the client will need to initialise all of those objects and keep track of the dependencies and execute all the methods in correct order. However the down side of a facade is that it might become a god object that is coupled to all the classes of our project, however this can be solved by creating more facades.

We also apply interface segregation principle for Homebooking and OnSiteBooking as both of the classes are not forced to implement an interface that it doesn’t use. Moreover, Open Closed principle is also apply as we are able to add new features to our program (if in the future there are different kind of booking is added) we don’t need to change the existing code (the client: in our case client code might also be the java frame class as we implement it in netbean).

Furthermore, we also apply single responsibility principle to Booking and covid test as instead of sending a post request of covid test to web service in the booking class itself, we create another class (covid test) that sends the post request when a new covid test is created. Without single responsibility applied the booking class might become complex when more features are added and become a god class.

Package Design

The Stable Dependencies principle

Instability metric of TestingSite package: 2/(1+2)=0.66
Instability metric of Booking package: 2/(2+2)=0.5
Instability metric of Login package: 1/0=0
SDP stated that the package has to depend upon the package whose I metric is lower ,from the Instability metric we have calculated the Instability metric of the TestingSite package is higher than the Instability metric of the Booking package which fulfils the statement .For the Login package , the outgoing edge is pointing to a support package (enum) which seldom changes .

When grouping our packages, we follow acyclic dependencies principle as there are no bidirectional relationships or a cycle between the classes.

When grouping the class in the packages, we try to have a balance between CCP and CRP so that we don't create a really big package that contains all the classes (CCP) or create too many packages that only contain one or few classes (CRP). Last, the testing site package is initially formed by onSiteBookingSubsystem as it applies facade design principle and it is one of its outcomes.