Task 2- Design Rationale

MA_Lab03Team7: Chantelle Loh Yi Wei (31171109) and Siang Jo Yee (31159265)

This is the design rationale for the project, designing a Covid Testing Registration System by **MA_Lab03Team7**. We will guide you through each specific consideration we made for a better understanding of our design.

In our Assignment 2, we have make use of some design patterns that was taught over the last few weeks. We will be going through in details the main design patterns used and also some design principles that we have applied throughout the design process of the system.

First, we have made use of the Facade design pattern. Facade as taught in the lectures is a structural design pattern that provides a simplified interface to any complex set of classes or subsystem.

Facade in general is to act as a convenient access to something complicated in the system, usually a complex subsystem's functionality. We find this approach strongly applicable to our current design of the system. This is because as we were designing and implementing it, we realized that there are many complexity to our design.

We will first start to explain the thought process of our design, along with design principles that we are wary of and will elaborate further on the Facade design pattern.

The system's users have different roles and each roles have their respective access control. With this, we created an abstract class User so that the **Single Responsibility Principle, Open-Closed Principle** and **Liskov Substitution Principle** are upheld. SRP and OCP are upheld here, this is because we have separate concerns of each class and thus it makes it easier for extension and avoids modification on existing working classes. Any future extension will be able to easily implement as we also have a UserGenerator class which mainly focuses on the responsibility of assigning access controls to each user role. Next, LSP is also upheld because as a User in general are able to log in to the system. This feature shares among all roles regardless of their responsibilities thus we carefully design the methods such that whenever a method calls for a User Object, all roles Object will still be valid as input.

When we were designing the User, we were also concerned regarding the fetching of the data from the API. As we have make use of the SpringBoot framework, We have created classes such as UserRepository, UserController and UserFactory. However, we later have realised that UserFactory is not fully suitable for our design as we do not really need to expand the number of classes in the system too extensively.

With some changes to the name, we have created a pipeline for the data to be fetch to the backend of the system and deliver to the front end. The User now has UserApi to act as the “repository” to the api (this is because we do not have a database). The UserApi will work with some methods created in the ActionFacade where all the methods necessary are in the Facade. This is so that UserApi can be called at Facade and have UserController to get the methods from the Facade. As you can see, the Facade provides a convenient way to access what is necessary without needing to go through all the other classes and have more dependency.

Of course with that being said, we also keep in mind of **Facade’s** cons in designing, which is becoming a god class to all classes of the system. To avoid such, *ActionFacade* is only open to mainly method that requires a user interaction/action. For TestingSite, it will be on a different Facade called *TestingSiteFacade.* The main reason for this class’s creation is because we considered that in the future, there may be more information tight to the TestingSite (such as rating of the testing process etc) and they may provide more than one type of services. Thus, with that in mind, we have decided to create this so-called wrapper which is the Facade to aid in further development/extension of the system.

For Booking, there is also many separation of concerns as we have few types of booking, on-site booking to home booking thus we also keep OCP in mind so that we do not cluster everything in one class and make it harder for extension in the future.

A new design decision we made in this assignment for Booking is that we have added the design pattern **Strategy**. It is a behavioural pattern which lets us split a family of algorithms into separate class and make all of their object interchangeable. We favour this design because the main idea of strategy is to provide more than one route of solution in triggering that event. Quoting example from the lecture, which is the Google map idealogy. This helps us to understand that having the Strategy design pattern essentially means that we are able to split algorithm that several option of that event will have. These option can have slightly different algorithm but with the same outcome in mind, in our case its On-site Booking vs Home Booking. Both of them achieve the same outcome where they book a test but with different requirements.

Looking at this direction, **strategy** is suitable for this situation. Thus, we make use of the Strategy structure, having a Context class, Strategy classes. In Booking terms, we have BookingContext, BookingStrategy and then the OnSiteBooking and HomeBooking will implement that BookingStrategy interface. You may refer to our class diagram for more details. With all the structure in place, we also further understand the pros of having strategy and its simple method where **interface segregation principle (ISP**) is withheld.

With concerns towards **Release Reuse Equivalence Principle (RREP) and Common Reuse Principle (CRP)**, we make good use of grouping classes up in packages. We have few packages such as api, testingsite, user, booking, login, controller, covidtest.

We will talk about all of them in general and dive deep to some main packages. In general, we understand that to uphel **RREP** and **CRP**, we need to make sure they are able to be reused elsewhere and also having something in common. Looking at all the api classes, instead of grouping them with their respective entity classes, we package them up in an api package instead. This is because there may be some senario in the future where we want to access to all the api but not in this Covid registration system. The same idea applies to the remaining packages, they are made with consideration that they have something in common and also are able to reuse it in other system. For example, we may just design a testingsite registration system for other diseases not just covid 19. In this case, we are able to make use of the whole booking system and remove the covidtest package instead.
Looking back at the first few weeks of the course, We understand that the 3 principles **RREP**, **CRP** and **CCP** cannot be fully satisfied simultaneously. During the implementation, even just simple fact of aiding user readability, we leaned towards to CRP more. We prioritise the reusability and how each package have their respective responsibilities (it may not be a singular responsibility but are in similar interest).

There are few assumptions that we made and thus we also had patch to the API.

- For receptionist, additionalInfo has testingSiteId as they are facility staff thus it is easier for them to keep track where they work at.
- For testingSite, additionalInfo has siteType to keep track the sites type so that it is possible for user to filter through.