Design Report
Date: 01/06/2022
Unit: FIT3077
Team Members: Abdalla Abdellatif, Youssef Shash

Design Pattern

The design pattern we have used is Observer. This is a behavioural design pattern that allows us to define mechanism to notify all subscribers when an update happens to object they are observing. Referring to the assignment requiremnts, it stated that if a booking is modified, canceled or deleted, all others like admins, admin panel, testing sites and user active profile should be notified to update their bookings too. So, we have seen no more suitable design pattern than observer to notify all observers about changes. We have applied this in our design pattern inside booking package. We created a class called CovidBookingSubject and this acts as the subject where it performs all the changes that can be done to bookings, this class uses another abstract class called CovidBookingObserver to notify it about the changes. All the classes who needs to be notified about the changes inherits from Cove Booking Observer (Adminstrator, Customer, Patient, AdminPanel) and call update method to update as shown in the class diagram.

Advantages of using observer design pattern:

-> Observer by its nature allows for the open/closed SOLID principle to be applied. For example, I can add new subscribers independently without having to change the subject code
-> we can use observes independently
-> we can establish relationship between objectsat runtime
-> any future changes to subject or observers will not affect the other

Another design pattern we used is Facade. While we were refactoring our assignment 2 design, we decided to use Facade. Facade is a structure design pattern that provides a simple interface to a complex subsystem which contains lots of moving parts. We have seen that Facade is most appropriate design pattern to use because our system is divided into subsystems like, users, bookings, covid testings, facilities. An example to show how we used facade in our design is in users package, there is an abstract class called UserFacade and it uses the User class. All other subsystems/packages have a facade class and they these sub-facaed are connected to one big facade class called CovidFacade.

The reasons for our choice is as follows:

-> we can isolate the code from the complexity of a subsystem
-> Facade create a balance between CCP (common closure principle) and CRP.

Archeticural pattern

We have used MVC architectural pattern because it has a faster development process, ability to provide multiple views, support for asynchronous technique and the modification does not affect the entire model since its follows the single responsibility principle and each class has just one thing to do. To show how we used this in our design, in booking interface, there is a CovidBookingSubject class that acts as the model and performs all the calculations, BookingView that handels all the display and have view method to display all the bookings, and BookingController that deals with both the view and the model.

SOLID principles

While applying the design patterns correctly, we have applied the SOLID principles as well. For instance, in the booking package, we have used the observer design pattern. The SOLID principle applied here is open/closed principle, where the observers are independant of each other and I can add another observer without affecting the rest of code. All the classes who needs to be notified about the changes inherits from Cove Booking Observer (Adminstrator, Customer, Patient, AdminPanel) and call update method to update as shown in the class diagram.

We have also used Single Responsibility principle while we used the MVC. Inside the testing, there is a class that does all the logic (CovidTest), a class for the display (CovidTestVeiw) and a class that handle things between view and model (CovidTestController).

2.2 Package Principles

Package Cohesion Principles

The first principle we chose is the Common Reuse Principle. This means that if classes make up a component, they should be packaged together. In our conceptual class diagram, this principle is used to group CovidTest abstract class and the subclasses RAT, PCR, together as they make up a component for COVID-19 testing and they are tightly coupled. Classes such as the User do not belong in this package as the User does not necessarily make up COVID-19 testing, even though they are the ones that get tested. Another example would be Facilities packaged. Inside the package, there are only classes that make up the component of being a COVID-19 testing facility. This also includes the CrowdLevel enumeration class to determine the busyness of a facility, the ServiceType enumeration class to determine if a facility provides only walk-in service, drive-through service, or both, and the Coordinate class to determine the latitude and longitude of a facility. In addition, Booking and NonBooking facility subclasses are included in the package. All of these classes have a higher probability of being used together, hence, they are packaged in the same package.

With that said, we did not 100% adhere to CRP. Adhering 100% to CRP causes packages to be really small as there are not a lot of classes that are being used together. As a result, we mixed Common Closure Principle when determining the groupings for the classes. CCP is similar to SRP and OCP, however, it is on a package/component level. Adhering 100% to CCP causes the packages to be really huge. More often than not, every class would be in the same package. To find the ideal packaging, we decided to find balance between the two principles. The way we decided the packaging was, if a class is related to another class, they should be in the same package. This could be said with all classes. However, we decided that if the class has a weak relationship (such as no direct relationship) with another class, they should not be in the same package. With this, we believe that we have come up with the perfect packaging which can be seen in our class diagram.

Package Coupling Principles

The first package coupling principle that we adhere to is the Acyclic Dependencies Principle, which states that the dependencies between packages must not perform cycles. Every single dependency in our conceptual class diagram does not perform a cycle. This is because we used the Interface Segregation Principle and the Dependency Inversion Principle on the class level. This means that there should not be even a single cyclic dependency (including bidirectional) between classes. In turn, there would be no cyclic dependencies between the packages.

The first package coupling principle that we adhere to is the Acyclic Dependencies Principle, which states that the dependencies between packages must not perform cycles. Every single dependency in our conceptual class diagram does not perform a cycle. This is because we used the Interface Segregation Principle and the Dependency Inversion Principle on the class level. This means that there should not be even a single cyclic dependency (including bidirectional) between classes. In turn, there would be no cyclic dependencies between the packages.

There is another principle that we did not particularly focus on, which is the Stable Abstraction Principle. We did not focus on this as we do not want to make our code inflexible. On top of that, we also did not want to create a lot of unmeaningful abstractions that could make our code more convoluted. If there are packages that adhere to this principle, it just happened by accident. We could see that by applying other principles as mentioned above, the Covid package indirectly adheres to SAP.

Refactoring

Our last assignment had so many issues, we did not have a clear design pattern applied. So as we mentioned above, we used facade this time on the assignment 2 classes. We needed to initialize all of those objects, keep track of dependencies, execute methods in the correct order, so as a result, the system had many interdependent classes, so it was hard to comprehend and maintain. So we decided to use the facade design pattern. The facade provides a simple interface to a complex subsystem which contains lots of moving parts. For example, we made a class called user facade, Instead of making your code work with dozens of the framework classes directly, we create a facade class which encapsulates that functionality and hides it from the rest of the code. The original design did not have facade at all, so we consider this step as our first refactoring.
In the customer class, it was an abstract class, we changed to interface which allowed us to inherit it to many other classes (customer, patient, admistrator, adminesterer).