FIT3077 Assignment 3: Design Rationale

MA_Lab03Team7

(31159265) Siang Jo Yee
(31171109) Chantelle Loh Yi Wei

Assignment 3 involves adding a few features to our system: a Covid-19 Testing Registration system. Design principles are used widely across many considerations thus there will not be a specific section for them. We will discuss our design rationale for our design decision in these sections:

A. New Design Changes

Design patterns and principles

With implementing the additional feature in mind, we consider our design both from the feedback of the last assignment and the system's extensibility concerning the new features. We have used a few new design patterns, Memento and Observer patterns. Both of these patterns aid us in implementing the new feature while keeping it open for extension.

Memento

Memento pattern is used in the system for the implementation of Task 1:Booking Modifications (Residents and Admin staff). It is especially useful for the restoring booking feature as it keeps track of the past few bookings and reverts when necessary. The problem faced when implementing the feature is that to keep track of the booking history, we need to allow access(read and write access) to it for various operations. Outside classes, therefore, become very dependent on the Snapshot class (the class that keeps track), down to small changes such as changes in the field. This in return may expose the original object's state and if we restrict it too much, we cannot keep track of the history. This is why the Memento pattern helps. The previous scenario is all caused by broken encapsulation. This pattern suggests that instead of directly accessing the state of the ActiveBooking(the original object that has the change), we store the copy of the state in Memento. Other objects which want to access the booking history data can communicate with the memento instead of directly accessing the original object's state. The pros with this pattern are that now my Booking system can produce snapshots(BookingHistory in our case) of the ActiveBooking's states without breaking the encapsulation. The caretaker who stores the Memento object in our system is the BookingModifier.

There are some cons to the pattern used. It might take a lot of memory space if the system creates mementos too often and the caretaker needs to track the original object's life cycle to destroy unused Memento.

Observer

The Observer pattern is useful for Task 2 Admin Dashboard Interface as it involves a notification system with a strict condition(not all admins in the system database will be notified). This definitely suggests a Publisher and Subscriber relationship where the Observer pattern can come in handy. In our system, BookingEvent Manager act as a "Basic Publisher" which defines the Subscription logic and BookingModifier/BookingExecutor mainly acts as a "Concrete Publisher" which creates the EventManager instance and uses it to notify all the subscribers. We have a NotifyReceptionistListener which will also update accordingly to which testing site's admin. With the BookingEventListener interface(a subscriber interface), we can keep creating a new subscriber class which can implement that interface and be included in the BookingEventManager.

The pattern essentially withholds Open Closed Principle as it can allow you to introduce a new subscriber class without having to change too much/not at all on the publisher's code. This also allows us able to establish the notification system at runtime which means any kind of relations (another type of subscribing feature can be implemented as well). However, one con that remains clear is that if we have too many subscriber classes, the subscribers are notified in random order.

Software architecture

We have integrated MVC architecture into our system. The motivation behind this is to avoid cyclic dependencies between User Interface and the Model. Our system has been designed with the Extracted UI approach as we have chosen Vue Framework which has extensive libraries for us to work with better visuals. With this, we understand that there is an issue when there are multiple, simultaneous views of the same model.

The covid-19 testing registration system is to be used by many users ranging from the residents to the admins. The nature of the target system will then face a problematic scenario, which is as the model data is modified/updated by any user, an update of all views must be given by the system automatically. This brings us back to what we learned in the lecture: "How can the model trigger this if it knows nothing about UI classes?" (FIT3077 Week8 slide 27) thus a cyclic dependency between the View and the Model is inevitable. This brings us back to the MVC architecture where it introduced Controller and separates the UI aspects: presentation logic (View) and update logic (Controller). This approach is essentially the Acyclic Dependency Principle.

As we are using SpringBoot and VueJs frameworks, the integration of the architecture may differ slightly due to their respective built-in architecture. Although so, we have implemented them strictly following the MVC architecture. The view pages Vue File are grouped together under the View component of the architecture. In it, we also made modules/packages for them to group them based on what is relevant to the target user. As we know the system will be used by users of different roles thus there are 3 distinct role packages and a common file that all roles share in common. We have taken Release Reuse Equivalence Principle (RREP) and Common Reuse Principle (CRP) into consideration as we want to be able to reuse the view packages if it is needed in future extension. This helps us to be able to segregate the presentation logic for respective roles as well.

Next, in SpringBoot, we do not have a Class which creates all Controller objects as in the framework it is not necessary to call on the Controller in that fashion. Regardless, the Controller component is still the one handling the update logic of the UI.

The main business logic of the application/system is in the Model. We have made use of the package principle of Release Reuse Equivalence Principle (RREP) and Common Reuse Principle (CRP) here. There are booking-related, user-related packages etc each is made with close consideration are they reusable if placed in another system or do they have anything in common. One highlight is that BookingUser arguably can be a booking related class which may be suitable for the booking package however it is definitely not fully common with the other classes in the booking package. Also, it cannot be released together with the booking package in another system as it has dependencies with the User. Thus, the current design decision.

Refactoring technique

There are a few refactoring techniques that we used extensively as we find them useful and applicable.

1. Extract Method

   There are some methods that are too long and have many code fragments in them thus we made sure to extract them and create a private method in that class. It can be considered as a support method to the main public method in that class. It helps with the readability of the class and we have given some method a meaningful method name so that it is easier to read through at a glance. Few classes benefitted from this are NotifyReceptionistListener, HomeBookingStrategy and BookingMemento.

It also helps reduce code duplication as we are able to make use of the support method repeatedly instead of writing the same code. This satisfies the DRY principle (Do not repeat yourself)

2. Move Method

We understand that our implementation of the Facade design pattern helps us to achieve this refactoring technique. In assignment 3, there are many more dependencies as we implement more features. Our old Facade design decision from A2 has helped us a lot as we are able to filter out what is the main important function and only have them in Facade classes. By doing so, we have moved many methods as we realise from a higher point of view that some methods can exist together in certain classes and finally be called by their respective Facade. We have also made use of other refactoring techniques such as extracting variables(we have many examples across the system, making use of variables in our classes) and Inline temp etc.

Support of Initial Design

We have also realised that having a good design decision in A2 has supported us in reducing many dependencies. Creating a MainFacade class and instantiating 4 other Facades for the respective packages has allowed us to reduce dependency between classes and have them all concentrated to one higher point, the Facade.