Assignment 3 - Task 2: Design Rationale

Design Principle – SOLID Principles

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is defined as a class having only one reason to change. In other words, a class should only be responsible for a single part of a software's functionality, with it being encapsulated by the class. In our system's design, we ensured that most of our classes would only have one reason to change. In other words, each functionality of our system has been separated into its own classes and is entirely encapsulated within them. The SRP had been implemented since the start of Assignment 2. We decided to prioritize SRP in our system's design due to its advantage of reducing complexity, which consequently produces clean and standard code. Besides that, it helps to achieve cohesion and loose coupling between classes. This means that if our system was to have a change in functionality or responsibility, all the pieces needed will be in a particular class. This makes it easier for us to handle the system's functionalities where if we would like to change some functionality, we wouldn't easily break other parts of the class. Therefore, with SRP, our system is easier to maintain and extend. Furthermore, by implementing SRP, it allows us to implement the Open-Closed Principle, which will be discussed later down below. Essentially, the system is scalable and readable, thus it is easy to refactor or change code. However, a drawback of the SRP is that it might cause some of the classes to become god classes if we didn't implement it properly. For example, our CovidSystemFacade will be a huge class that contains all the functions which deal with the user interface. Therefore, if we have more functions to add to the interface, the class will become harder to maintain.

Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) is defined as software entities such as classes, modules, and functions that should be open for extension but closed for modification at the same time. We decided to implement through the usage of interfaces and abstract classes, as it ensures that our system is open for extension. In other words, it will be able to easily support new functionalities, such as the addition of new features and methods. In our system, we have the Action abstract class which is extended by the LoginAction and

VerifyLoginAction classes. Through the implementation of OCP, we will be able to add more types of actions, without editing any of the existing classes. Besides that, OCP allows our system to be closed for modification. This is due to the fear of breaking existing client code if we were to change the way we use existing code, such as when changing method names or signatures. Therefore, OCP was implemented by extracting each functionality or method into its own separate class with a common interface. This allows our system to add new methods for each functionality without involving other existing classes.

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) is defined as when extending a class, we should be able to pass objects of the subclasses in place of objects of the parent class without breaking the client code. We decided to implement this principle due to the usage of several abstract classes in our system. Our subclasses would remain compatible with the behaviour of the superclass. This principle is adhered to through the parameter types in the super method. Essentially, we ensured that the methods in the superclass were able to take in parameters in a more general form. For example, our users can execute different types of actions, be it a searching action, login action, or other actions. Besides that, the LSP helped us to handle exceptions in a structured manner, by ensuring that subclasses that should not throw exceptions did not do so. Furthermore, by implementing LSP, our system can easily extend new classes. On the other hand, the maintainability of the code is increased. Due to the reasons mentioned above, we had decided to keep the implementation of the design principle.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states that high-level classes should not depend on low-level classes as both should depend on abstractions. On the other hand, abstractions should not depend on details but details should depend on abstractions. The low-level classes in our system such as the Code would generate either the PIN code or QR code. Then, the high-level classes that contain the business logic would call the low-level class. For example, Booking would generate Code. However, Booking depends on the Code interface, rather than the low-level classes that implement the Code interface. DIP also helps the implementation of OCP, as we would be able to extend low-level classes using different logic

without breaking any existing classes. This design principle had been implemented in Assignment 2 and we had decided to retain it.

Design Patterns

Facade

A facade is defined as a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. We implemented Facade due to our software needing to work with multiple objects. Besides that, there is the need of initializing multiple objects, keep track of dependencies, as well as to execute the methods in the correct order, especially the printing of different menus for different users. Therefore, the Facade enables us to provide a simple interface to this complex covid system which contains multiple moving parts. In our system, the CovidSystemFacade is our Facade, as it provides convenient access to multiple parts of the subsystem's functionality. Our covid system consists of multiple functions and responsibilities. Therefore, instead of calling them directly, a facade is used to only call what is necessary. The Facade helps to simplify the interfaces in our system, as well as decouples the client from a subsystem of components. Furthermore, the Facade allows us to have a direct and easy interface to a complex system with multiple functions. However, the Facade technically violates the SRP, as it becomes a god object coupled to all classes of an application, due to its ability to call all the subsystems. In Assignment 2, we had already implemented the facade design pattern, through a class called Menu, which essentially did the same thing as CovidSystemFacade. In Assignment 3, we decide to continue to apply Facade as our design pattern because it is much easier to maintain all the interfaces in one place and handle all the user input and view.

Factory

The Factory method is defined as having an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. In other words, it is a virtual constructor. In Assignment 2, we implemented the Factory method by creating a factory method for the bookings, which creates different types of bookings. We used to have an abstract class called booking which is extended by OnsiteBooking and HomeBooking, which are different implementations of the Booking class. Besides that, we used to have

OnsiteBookingAction and HomeBookingActions, which were the factory methods, as they would create the OnsiteBooking and HomeBooking objects. However, we removed these implementations in this Assignment 3. This is because we think that having multiple action classes for booking is extra where the action classes are just with one method (execute) and this can easily be placed in the Booking class itself and this will ensure we follow SRP.

Abstract Factory

Abstract Factory is defined as a creational design pattern that lets us produce families of related objects without specifying their own concrete classes. In other words, we create a concrete implementation of the abstract factory and use the general interfaces to create the concrete object that belongs to the interfaces' family. In Assignment 2, we implemented the Abstract Factory design pattern to BookingAction and Booking which used to be extended from the Action class. Both BookingAction and Booking classes would have their own concrete classes. This helped to enable the client to work with any concrete factory variant and communicate with their objects through abstract interfaces. However, this design pattern will make our design more complicated as the system increases its functionality. Therefore, in Assignment 3, we decide to not continue applying this design pattern and move all the BookingAction classes' methods back into its Booking class as discussed above. Besides, another reason that we remove this design pattern is to implement one of the software architectures which is Model View Controller (MVC) which will be discussed below later.

Package Level Design Principles

Acyclic Dependencies Principle (ADP)

The Acyclic Dependencies Principle (ADP) is defined as allowing no cycles in the component dependency graph. Essentially, the dependencies between packages must not form cycles. At a lower level, it will not happen between classes. When components and dependencies between them are able to follow a dependency back to a component that has already been visited, then it is a violation of ADP. In this assignment, we ensured that our system did not violate the ADP where we design our system as only the users' package can access all the other subsystems such as booking, facility and testing package. From our design, we can

ensure that a cycle does not form where the other subsystems won't point back to the users which will cause a cycle.

Stable Abstraction Principle (SAP)

The Stable Abstraction Principle is defined as a component being as abstract as it is stable. The conformation to SAP leads to stable components containing many abstract classes, and unstable components containing many concrete classes. In other words, the SDP leads us to a picture of a dependency structure consisting of unstable, irresponsible packages at the top, and stable, responsible packages at the bottom, with all dependencies pointing downwards. Essentially, the packages at the bottom are very hard to change. However, they are not hard to extend. This is where the implementation of OCP is enabled, where the stable packages are highly abstract and thus easy to extend. Through SAP, the most dependent upon and thus stable packages are the most abstract. For example, the Booking class is highly abstract as it barely does anything, consisting of only accessor and mutator methods. The Booking class is extended by the HomeBookingModel and OnstieBookingModel, which do most of the complicated logic. This means that the Booking class defines the high-level policy and leaves room for various implementations of the low-level details. This also occurs in the Users class, which is extended by the Customer, HealthcareWorker, and Receptionist class, which are essentially different types of users consisting of different types of attributes to be created. The Actions class does this as well, as LoginAction and VerifyLoginAction classes extend it to perform their own type of action. This enables the OCP as different types of actions can be introduced by extending the abstract Actions class.

Software Architectural Patterns

Model View Controller (MVC)

In Assignment 3, we have implemented one of the software architecture design patterns which is Model View Controller (MVC). From the name of the software architecture pattern, we can know that Model is normally referred to as business logic whereas in our case is referred to as the API and some other system logic. Then, View will be known as the interface of the system and is about representation where in our system will be print out

some description and output. The controller will be the one who controls the interaction between Model and View. So, in our implementation, we refactor all our subsystems into each MVC such as booking, testing and facility where each of them holds its own set of Model, View and Controller. The reason that we implementing this architecture design is that the Model does not depend on the View and so, adding new Views will not affect the Model or logic in our system and this can ensure to change the UI requirements in the future we might use another type of UI to handle user interface. Then, in our main Facade class, we will be calling all the controllers in the subsystem where the controller is to control both the View and Model and we can easily handle all the subsystems without breaking the Model or View when we are changing it. Therefore, we are able to separate the storing, displaying and updating parts into three different components that can be tested individually.

Refactoring

For the refactoring part, we modify quite a huge part of Assignment 2. One of the biggest refactoring parts is that we try to remove all the actions class when users are trying to create some object. For example, in Assignment 2, if a customer wants to make a Home Booking, it will go through HomeBookingAction class then the HomeBookingAction will then create the HomeBooking class. This is quite annoying as we are introducing an extra action class in between where we can straight away create the HomeBooking class without passing by the HomeBookingAction class. Besides this example, there is still some same concept that has been applied in Assignment 2. So, in Assignment 3, we decided to remove most of the unnecessary action classes to reduce the number of classes and store all the methods in the action class back in the original class, so it is easier for us to maintain the system in the future and this also helps us in converting to MCV architecture pattern. Furthermore, we also decided to change some of the names of the classes and packages to a more suitable and related names. For example, initially, we name all the users as actors but this doesn't suit this system, so we decided to change it to users instead of actors. In conclusion, by having these refactoring in our Assignment 3, the whole system now looks much more tidy and easy to maintain as well as easier for other developers to read our code.