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 packages and classes and is entirely encapsulated within them. 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 means that no excess features will be involved as they would have been separated into other classes. This makes it easier for us to handle the system's functionalities. If a class has too many functionalities, we would have to change it every time there is a change in the system's functionalities. There would also be a risk of breaking 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. Not only that, but SRP also allows us to implement the Liskov Substitution Principle. However, a drawback of the SRP is that it causes the system to have many classes, due to our system having multiple functionalities. This may be due to the creation of many small modules or classes. Despite being packaged, these classes may be confusing to future developers of the system.

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 Code interface which is used by the BookingAction class. Through the implementation of OCP, we would be able to add more Code types, 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 many 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 actors are able to 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 is able to easily extend new classes. On the other hand, the maintainability of the code is increased.

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.

Design Patterns

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. We implemented the Factory Method by creating a factory method for the bookings, which creates different types of bookings. In our system, we have an abstract class Booking which is extended by OnsiteBooking and HomeBooking, which are different implementations of the Booking class. Besides that, we created OnsiteBookingAction and HomeBookingActions, which were the factory methods, as they would create the OnsiteBooking and HomeBooking objects. The factory method allows us to extend the system's internal components whenever we would want to. Besides that, it saves system resources by reusing existing objects rather than rebuilding them. Furthermore, tight coupling between the classes is avoided. On the other hand, the factory method helps us to implement the SRP due to compiling product creation code together. Besides that, OCP is adhered to as we can now introduce new types of products into the program through a factory method, without breaking any existing code. However, the Factory method makes the general code more complicated as we would need to introduce multiple new subclasses to implement this pattern.

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 our implementation, we apply this design pattern to BookingAction and Booking which are extended from the Action class. Then both BookingAction and Booking classes will have their own concrete classes. By doing this, the client will be able to work with any concrete factory variant and communicates with their objects through abstract interfaces. Therefore, this pattern allows us to follow the Single Responsibility Principle where each creation part of code does only one feature and this will avoid tight coupling between concrete factory and client code. Besides, this will also allow us to follow the Open/Closed Principle when we want to introduce new variants of concrete class without breaking the existing code. However, if the features or code become more complicated as the system goes bigger and bigger, there will be a lot of new classes and interfaces being created with causes a large number of classes.

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 ur 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 Menu 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.

Further Design in Assignment 3

After completing Assignment 2 by applying the design principles and patterns, we have come out with a brief idea of how to implement design architectures. In our implementation in Assignment 2, we have two main components which are our interface view (menu) and model (all the logical action) where whenever the model data is updated, the system view must always update automatically. However, this is very inefficient to have every view to check whether the data is updated or not and this will cause a cyclic dependency. So, to solve this problem, we introduce a new component called a controller that knows both model and view. By applying Model View Controller, the model will still manage the data of the system, the view will manage the display of the output and the controller will interpret with the users' input. After the controller is interpreted by users, it will inform the model and view to change as appropriate to follow Acyclic Dependency Principle.