Design Rationale

Software Architecture

Our systems have been designed to be divided into several subsystems. Different subsystems may have multiple different classes implemented to cater different responsibilities. As we are going with MVC architecture, specifically passive MVC, we package classes based on model, view, controller, before further packaging them based on subsystems. These packages then can be easily used for the booking process as all classes inside are reused together following Common Reuse Principle. As these classes are mostly dependent on each other, this also ensures that any change or addition to the requirements relating to Booking occurs among them, meaning the change outside the package can be minimised, which adheres to the Common Closure Principle. Now that the classes have been packaged together, the package can be reused and released. The Release Reuse Equivalency Principle allows these packages to be used across multiple projects, with inclusive changes and reduction of redundant codes.

In continuation of the newest subsystems that make use of Memento and Observer design patterns, we have kept our implementations to adhere to the MVC architecture design. In short, MVC separates things into a model which is an entity and its structure in the system, a view which is an interface to connect users to the system, and a controller which controls interaction between model and view. All of our current implementations have already been separated into these three components, frames and panels as views are separated from models, and controller classes related to booking, scanning, administering, booking modification, and notification system reference the views associated and models needed accordingly. With MVC, it is much easier to split the logic behind the structure and the actual view separately thus achieving separation of concerns.

Model package is the most stable package as most of all the dependencies are incoming dependencies due to models being used by controllers to update views. While the Controller Package is the least stable. This follows the Stable Dependencies Principle, where a less stable package depends on the package with a more stable, as shown in the class diagram. In addition to this, Stable Abstractions Principle states that a component should be as abstract as it is stable, which can also be seen here as the Roles interface and roles specific classes, such as Customer, Receptionist and HealthcareWorker, which contains abstraction.

Assignment 2 Specification

Web Service Connection

To start off, our system will use data provided from the web service API. Web service related requests require additional import from libraries, however libraries used are large and complex and we do not actually need all the parts inside. Therefore, the Connection class is created using a facade pattern, such that we only import necessary methods to use, providing high-level interface and isolating our code into a single entry point. From the creation onwards, we can straight use the methods provided relating to respective functionality without the need to import and care for unnecessary functionalities. Although we use facade here, it is implemented to follow the Single Responsibility Principle, where only functionalities that are connected directly to the Web Service are accumulated into a single class.

Login Subsystems

Different roles for each user result with different GUI and menu as well as different responsibilities. To prevent excessive use of control structures in GUI and functionalities based on role, we adopted the strategy pattern. Role interface is created to expose methods to trigger algorithms relating to a specific role, concrete classes implementing roles (Customer, Receptionist, and Administerer) are the strategies to be used in the context class which is user. Using strategy pattern allows the user class to use specific methods to the prime role assigned to it, in terms of specific action expected given all the roles use similar parameters, as well as getting information such as string and menu tab needed. In relation to the GUI, the GUI creation and set up is much simpler e.g. with the help of additional MenuTab enum class, the roles classes implementing Role can directly set an enum value to indicate menu tab used and the actual creation and modification in the future is now encapsulated inside the MenuTab.

We decided to go with the strategy pattern as we want to be able to switch implementations quickly during runtime as logging in and out as different roles is possible and they differ only slightly in GUI layout and functionalities so not much changes are required. Adapter pattern was considered initially for role implementation, however we discarded the idea as adapter is better for compatibility issue when we need to “convert” an interface by making use of another interface, in our case user is already a concrete class and we simply wanted to run different role-based functionality instead of changing user’s functionality.

The application of strategy pattern in handling roles also maintains the Open/Close Principle, as such all roles have common methods to indicate allowable actions, therefore extension of new roles is allowed by implementing the role interface and making sure allowable actions are recorded inside the MenuTab class.

Search Subsystems

The convention of searching for sites should be done in the correct class where search site related methods are put in. Previously, search was created as a tab class to search for testing sites by filters and/or suburb name. This, however, is misleading as search should be method instead of class and it violates SRP as search class contains responsibility for creating new booking as well. To fix this, Search class should not exist and is converted into

BookingTab class and search methods are moved outside to the correct class. This now follows SRP correctly.

Since the search method relates to the site, it is natural to put it inside the TestingSite class. However, this would require static methods and it also violates SRP as this class of specific type may return a list of its own type. Therefore, the singleton design pattern is used here for site collection and all methods related are placed inside. This is also good as previously some of the classes (i.e. Booking) keeps on making a new TestingSite object which would create references to many objects of the same attributes and cause redundant calls to web service and excessive usage of memory. Using singleton provides reference to a single object to avoid redundant instantiation.

On Site and Home Booking and Testing Subsystems

To cater for different types of booking and administer classes, where each type is only slightly different in terms of additional logic and GUI, we extracted base attributes and functionalities into a super class for both controller and view, before customising specific type accordingly as subclasses which follows the Liskov Substitution Principle. Factory method was initially used as it allows us to reduce the work of connecting each specific class into the application itself. This promotes the loose-coupling by eliminating the need to bind application-specific classes into the code provided, which tends to have a variety of functionality. However, we decide to omit it after considering the amount of common functionalities to put in the base class is more beneficial than taking consideration of altering the subclasses for the view, as this can be easily done by constructing the corresponding view during construction of each controller. This also follows the Open Closed principle where we can introduce more booking or administration procedures.

A Covid Test can only exist after a booking is made for that test. In that sense, we can say that Covid test will be a part of booking in the future. Therefore, there must be a way to access information about a Covid Test that is connected to the booking from the booking itself, which is possible if we store the covid tests inside the booking class and create methods needed accordingly. This is usage of bridge pattern, where booking acts as a bridge for users to access the Covid Test, it acts similarly to Associations between classes as they are proposed as a “bridge” between 2 classes. To make this design pattern possible, we would need to follow the Dependency Inversion Principle, where using Dependency Injection, Covid Test is injected into the Booking constructor and stored as Booking’s attribute. Using this pattern also makes it easier to connect to the Covid Test class without needing to actually recreate or re-initialize an instance of the class just to access information, we can simply make use of the booking as a bridge.

Although several other subsystems and functionalities can be implemented using singleton, for example login and web service connection, we opted to avoid using singleton for these systems and simply create and use a new instance when needed. This is mainly to prevent over-engineering and cluttering our design which may force too many responsibilities inside a single class essentially breaking Single Responsibility Principle. In addition to this, singleton may prevent scalability, for example it will not allow creation of multiple logins which is an actual possible scenario in real life where users might login in different windows. Another important thing to mention is single entry point provided by singleton makes it hard

to test them, for example if we force it for HTTP connection, we would end up having to be extremely careful in deciding order in run as they may possibly produce different results in the connection set, thus singleton is not suitable for these cases.

Assignment 3 Specification

Booking Modifications Subsystem

As booking modifications actions are specific to certain roles (resident and receptionist), our previous strategy pattern can simply be used. Following the MVC architecture, booking modification has controllers as well as view and model reference in the controller . As actual modification for booking does not differ between resident and receptionist role, a controller and view specific for the actual modification will be applied for both roles (BookingCareTaker and BookingModificationFrame), the view differs from the modification tab leading to the frame (profile for resident and modification tab for receptionist - slight changes in logic and buttons allowed are implemented accordingly, e.g. search for booking and disable modification for lapsed booking). To indicate different status of booking after modification (including creation, deletion, cancellation, and modification), enum status class is created.

The main focus here is ability to modify and undo a booking. For reverting changes, we assume undo is only allowed at a specific time during the actual modification and by the same user. As modification and undoing requires the state of changes to be captured and restored, the memento design pattern is an excellent choice. We define bookingMemento class which is a memento to store state of changes, in this case, the testing site and time selected. Booking class acts as the originator as it is the object which state we are interested in, it produces states to save as mementos or accept mementos to apply undo. Finally a booking caretaker class which actually acts as a controller for BookingModificationFrame will save references to previous states/mementos of the originator by saving reference to the originator as well as an array of memento - methods such as undo or modification are implemented accordingly. With memento, we can restore and save the state of booking modification externally outside the booking object - thus keeping the booking object stateless.

Admin Booking Interface Subsystem

For the notification system, the admin booking interface is created for all Receptionists to receive notifications whenever certain actions, such as add, delete, cancel, or modify, are done to the booking object. Notice the keyword notify here, this is a great example to use the Observer design pattern to solve the problem as discussed above. Observer pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. In this case, Booking serves as the observable object and has reference to all receptionists which are the observers that will be notified whenever Booking is modified. The creation of Observable and Observer interfaces follows the Open Closed Principle as it provides us a way to extend a wider variety of observables/observers in the future depending on the requirements. In addition to this, Integration Segregation Principle is also adhered as only job responsibilities necessary are associated with the interfaces, which keeps it small and manageable.

For the notification feature itself, when the Booking receives any action, Receptionists working on that site and/or new updated sites are updated about these changes thus a list of admins is kept under TestingSite class who works in that particular site and it will be the site's job to keep the list updated with all admins. Upon pressing the actions button in the view class for any action made to the booking, it will trigger a booking (observable) method to notify all receptionists (observers) to update the notification list in the database. The observers will then pull the given notifications from the database.

Assumptions

- All roles have the same GUI, but certain features are only visible to certain role of users.
- Since Customer stores an ID, we assume all Customers remember their ID when filling the booking question.

- To cater for home booking, the customer will have a GUI that shows what they have booked currently and do testing as needed.
- To cater for onsite booking, the administer could enter the PIN Code of the patient and show the patient's booking status.
- Customers know the TestingSite ID for modification.
- When booking modifications occur, customers will be able to change the date, but the time will be set at the end of the day by default.
- Undo is only done at a specific time.