Design Rationale

In order to implement the covid testing and booking subsystem, we have created a standard Maven project. Maven was chosen as dependencies can easily be added, by simply reading the pom file.

For the user interface, our team has decided to use the console. The console was chosen due to its convenience and ease of implementation.

Package Principles

All the classes associated with the users (Resident, HealthCare Worker and Receptionists) have been placed in the Users package. The rest of the classes have been placed in a separate package called system.

The main package cohesion principle that we focused on following was the Common Closure Principle (CCP). This states that a change that affects a package will affect all the classes in that package and no other packages. This can be seen within our Users package where all classes within are highly cohesive, if one of these classes were changed it would also affect the other classes in the User package. Furthermore, you can see that classes outside the User package are not highly dependent on the classes within the User package. This can be viewed from our code where the System classes are not heavily dependent on imports from the User class.

By following the Common Closure Principle we were able to ensure that our components followed the Single Responsibility Principle. This allowed us to increase the maintainability of our code as changes to a single component will not affect other components outside of the package. However, in following CCP we tended to minimise the Common Reuse Principle which states that components in an application should not be too specific. In doing so packages can become difficult to use due to their specificity. However, in the long run we decided that the maintainability provided by CCP outweighed the possible negative impacts.

Another package coupling principle that we focused on following was the Stable Dependencies Principle. This principle states that packages that change more frequently should depend on packages that don't change as much. This can be seen with our users package as we only have dependencies coming into the package and not going out.This will help to increase our software's flexibility by making frequent changes much easier. It also aids in isolating functionality on which other parts of the system rely, thereby reducing the impact of changes.

Users

In this system, there are 3 main users - Residents who will be booking or taking the test, Administrators/Receptionists who will handle on-site booking and lastly HealthCare Workers who conduct the actual test on the patient. Thus, 3 classes, Resident, Receptionist and HealthCareWorker were made. As users of the same system there will be multiple methods shared by these three classes and therefore an interface, User, was made which Resident, Receptionist and HealthCareWorker implement **(Abstraction)**. As the Receptionist and Resident are able to make bookings, we have also

created another interface called BookableUser which is implemented by the Receptionist and Resident classes **(Abstraction)**. By using interfaces here, we can hide implementation details and also reduce repeated code (**Don't Repeat Yourself Principle**). While using an interface, we define the method and the signature separately. So, all the methods and classes are entirely independent. This helps to achieve **Loose Coupling**. Furthermore, by using interfaces for Users and BookableUsers, we leave our system open for extension. For example, in the event that a new user will be introduced within the app, the new class can easily simply implement the User interface and/or BookableUser interface and thus other methods and classes will not need to be modified. This makes the system open for extension but closed for modification. (**Open Closed Principle**)

Login Subsystem

The COVID Testing Registration System has a simple login system. The type of user, i.e., resident, receptionist, or health care worker will depend on the username that the user logs in with and what the particular user is saved as on the database (i.e., the web service). We studied two different ways of doing this. The first was the chain of responsibility design principle. In this principle the type of user is passed down a chain of objects until an appropriate object is found *[1].* However, with this principle we found it difficult to extract and use the instances of these objects (i.e., the users) as every type of user would immediately be instantiated in the Handler class at the start when creating the chain. This pattern also made it difficult to debug as well as maintain the code as altering an object in the chain could potentially disrupt the chain and thus the entire process. There was also the possibility that the request may fall off of the end of the chain without ever being handled *[1]*.

Thus, we decided on using the **Factory Method design pattern**. In this way, the UserFactory class can create and return a User depending on the userType that is passed to it. By doing it in this way we can easily obtain the instance of User created and coupling is reduced between the client code and the concrete products. At the same time, we preserve the Open Closed principle as it becomes much easier to introduce new users by adding to the UserFactory class. The downside of this pattern is that there is the potential for code to become excessively complicated. However, this was minimised by the fact that only one type of factory was required (factory for creating users) and thus we did not need to create an interface and multiple factories from this interface. Of course, we still did need to introduce a few more classes in implementing this pattern, however if we did not use such a design pattern, complications such as tightly coupled code and low maintainability and extensibility would definitely arise. The ability to prevent these complications from arising by using the factory design pattern far outweighs the possibility of code becoming excessively complicated. To conclude, the Factory design pattern makes our code **more robust, less coupled and easy to extend.**

Search for testing sites

Before visiting a facility, a user can view the list of testing sites in the vicinity. We first created a class called CovidTestingSite which contains all the information about each testing site such as site ID, description etc. This abides by the **Single Responsibility Principle**. Our team then implemented this search functionality in such a way that a menu is displayed to the user with the different types of searches, and the user could do a specific search by inputting a number. In situations like these involving menus and commands being sent, it is common for the **Command design pattern** to be used

and thus we analysed the pros and cons of this pattern in relation to our system. Firstly, the pattern allows us to decouple the class invoking the command from the one actually carrying out the command which would improve the flexibility of the code especially in the case where new commands need to be made *[1]*. In addition, with this pattern you can also have the option to implement undo procedures so that you can undo previous commands. However, the main issue with this pattern was that for every individual command, we create a ConcreteCommand class. This will result in many small classes and a high number of different classes and objects working together.

Our search functionality contained 6 different options for searching for different types of test sites. This would mean that we would have to create 6 different classes for relatively simple code that could easily be implemented within a method inside just one class (in our case the Search class). This seemed pretty excessive and unnecessary considering the simple commands we would be implementing.

Thus, we decided not to use this design pattern and instead keep the methods for implementing these commands in a single class (the Search class). This class contains 6 different methods for searching for a covid test site by providing a suburb name, or the type of facility (such as Drive Through, Walk-in, Clinics, GPs or Hospitals). Since all our methods were a type of searching, putting it in a single Search class which is a **Singleton class.** Using a Singleton class violates the Single Responsibility principle, but we have still used it because it allows us to complete all the functionality in a single class, which reduces the code complexity.

To display the menu to the resident, we create functions in the Resident class which would allow the user to do different types of searches by inputting a number.

<u>On-Site Booking</u>

Both receptionists and residents are capable of making bookings. When this functionality was first implemented, the Receptionist and Resident classes both created an instance of Booking which was then used to make a booking. However, this meant that there would be a strong dependency between the Booking class and the lower-level modules, Receptionist and Resident in clear violation of the **Dependency Inversion Principle**. In order to solve this issue, a new interface was created called BookableUsers for users with the ability to make bookings, which included the Receptionist and Resident users. This interface was then injected into the Booking constructor **(Dependency Injection**). In this way, Booking now depends on the higher-level module, BookableUsers, instead of the lower-level ones. In addition to this, the code can also be easily extended to accommodate more users with the ability of making bookings**(Extensibility)**.

Furthermore, the Receptionist has to make bookings for a Resident and in such a situation it is possible for there to be many strong dependencies between these two classes. However, with the factory method design principle which was discussed earlier we were able to ensure that there were **no direct dependencies** between the Receptionist and Resident when the Receptionist has to create a Resident in order to make a booking for said resident. In this way we were able to **reduce dependencies** as well as **reduce coupling** between classes.

Finally, Receptionist extends the ValidateUser class which is used to validate a User when they login. This allows the Receptionist to validate any User who wishes to make a booking using the methods available in the ValidateUser class. In this way we **do not repeat ourselves** by reusing the pre existing code **(DRY Principle)**.

On-Site Testing

The healthcare workers/administerer will conduct a brief interview of the user and based on the answer, fill a form on the system and suggest the appropriate tests. In order to implement this functionality, we have created a new class called OnSiteTesting which extends the CovidTestingSite class**(Inheritance).** The purpose of inheriting this class is because the OnSiteTesting class is a type of CovidTestingSite. In this way, we **do not repeat ourselves** by making use of the pre existing methods in the CovidTestingSite **(DRY Principle)**. In order to recommend the test type to the resident, we have created three separate text files, each containing symptoms with a different level of severity, i.e. mild symptoms, moderate symptoms and severe symptoms.

Then, we have created a class called SymptomsCollection which is responsible for reading the symptoms from their text files. This class has been created as a **Singleton class** in order to prevent other objects from instantiating their own copies of the SymptomsCollection object (as it is a collections class, only one copy is needed to be used by all clients), thus ensuring that all objects access the single instance. In doing so we can allow controlled access to the sole instance of the SymptomsCollection class and thus we can control when and how clients will access it. Furthermore, by using a Singleton we can ensure that our namespace is not polluted with global variables that store only single instances. However, by using a Singleton class we violate the Single Responsibility principle as in this case, the class controls the creation of its own instance and also is responsible for storing and managing symptoms. However, when considering the numerous advantages that the Singleton class can provide us in the current situation as described previously we decided on going ahead with the use of the Singleton design pattern.

In the OnSiteTesting class, we call the methods in the SymptomsCollection class to let the user select the symptoms they have by entering either yes or no responses. In order to take these responses from the user, we have created another class called Question. The instance of this class will be called each time we want the user to answer yes or no to a particular question. This class has also been created as a **Singleton class** for the same reason as the SymptomsCollection mentioned above.

In order to allow the HealthCare Worker to conduct the interview, a function was created in the HealthCare Worker to display the menu to the user with the option to conduct the interview, and the user could do so by inputting a number.

Home Booking Subsystem

Residents can book for home testing through the system. The Resident has to register for the testing through the system first and indicate that they are registering for home testing. In order to implement

this feature, we have first implemented a function in the Resident class that will allow the user to input numbers for each functionality. If the home testing functionality is chosen, the system will first search for relevant test sites that offer home testing, and display them to the user. To do this, we have implemented a function in the pre existing Search class. By implementing all the search functionalities in this one class(explained above in the search for test sites section), we are abiding by the **Single Responsibility Principle.** Once the user selects the test site, the system will then ask the user whether they would like to collect their RAT test kit from the chosen testing site. The function will then generate a QR code and a URL link and add all the relevant booking details to the Booking object using the api. This function has been implemented in the pre existing Booking class by calling the instance of the Question class. By doing this, we are making use of old code and preventing repeating ourselves**(DRY Principle).** For the QR code,instead of creating a real QR picture in png, we have written a function that generates a random string to mimic the QR code function. For the URL, we have written another function which generates a random string which will be added to a predefined root URL. The functions for QR code and URL generation have both been placed in the Booking class. Since all the functions needed to make a booking have been written in a single class (the Booking class), we are once again abiding by the **Single Responsibility Principle.**

Future Extensions

In the future, a facade design pattern can be used by creating a single, simplified subsystem from which our receptionist, resident and health care worker can choose which specific subsystem or functionality they would like to choose. This will help us reduce complexity of the system and minimise dependencies between the different subsystems. Due to time constraints we were unable to implement this particular feature.

Software architecture defines the model of the software, how it will function, and the problems that may arise during implementation. As a result, these patterns make it easier to make decisions and manage change in order to obtain more accurate estimates of project time. In the future, a Layered Architecture pattern can be used for our system. The reason behind doing this is that the framework helps to reduce dependency in the system as the function of each layer is separate from the other layers. Testing is also easier because of the separated components, each component can be tested individually. Lastly, the framework is simple to learn as well as implement *[2].*

References

[1] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., n.d. *Design patterns*.

[2] 2022. [online] Available at: <https://www.baeldung.com/cs/layered-architecture> [Accessed 29 April 2022].