Task 2: 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.

Architecture

The fundamental frameworks of a software system, as well as the discipline of developing such systems and structures, are referred to as software architecture. For this project, our team has decided to use the **MVC software architecture**. MVC (Model-View-Controller) is an architectural pattern that divides an application into three main logical components: the model, the view, and the controller. Each of these components is designed to handle specific aspects of an application's development. MVC is a popular web development framework for creating efficient and scalable software[1].

MVC was chosen as it was the most appropriate architecture for our application without being overly complex and not suited to the scale of our application while at the same time bringing us many benefits. Firstly, after separating our code into the 3 separate components it made development faster as developers could choose which component to focus on and this would not largely impact the other components. We also found that it greatly improved the extensibility of our code as if we wanted to add new information to display to the user we could easily add it in our View component and this would not impact our other components and business logic of the application. Similarly, we could add new code related to both Model and Controller without risking breaking other aspects of the application due to the separation of concerns. Finally, MVC made it easier to understand the different functionalities provided by the system and this organised, segregated, approach also made it easier to apply and understand changes in the future. The separation of concerns additionally made it easier to test separate parts of the application.

However, an issue we found when implementing MVC was that it was sometimes difficult to separate the View and the Controller since our application is a console based application. This meant that the refactoring and separation of code into View, Controller and Model was difficult as these elements tended to overlap with one another making it difficult to separate out. However, this refactoring was made easier by the fact that during assignment 2 we strictly followed the Single Responsibility Principle for most of our classes that frequently interacted and obtained input from the user and also preserved encapsulation within all of our methods. This meant that functionality that displayed and asked information from the user was put into a separate method from the other methods which were more geared towards the business logic and manipulation of that data. This can be seen from all of our user classes where each User had a displayMenu() and displayOptions() method which was used to display information to the user. This encapsulation made it easier to separate View related code found in these two methods from the other code.

Package Principles

For assignment 3, we updated our system in order to classify our class into more distinguished packages. The new packages we made were booking, facilities, users, utilities. As well as 3 extra packages for our MVC architecture; model, view and controller.

We chose which classes to include in each package by selecting classes that are mostly dependent on one another. This means that reusing any one class from the package would also require the use of other classes within the same package. This can be seen from our class diagram where classes within a package are highly dependent on other classes within their own package. In this way we followed the **Common Reuse Principle (CRP).**

We can also see that by using our **Stable Dependencies Principle** that the User package has many incoming dependencies but has no outgoing ones. This means that the User package is very stable and thus will not be easily changed by other packages.

Finally, we also followed the **Acyclic Dependency Principle (ADP)**. As can be seen, the dependencies between our packages do not form a cycle. If this was not followed, then our system could get very difficult to maintain especially as it grows in size due to the high dependencies between the packages.

Refactoring

Code refactoring increases productivity and maintainability, improves readability, and improves testing and QA.

The main refactoring that was done was the refactoring of software architecture, wherein we modified our existing code to incorporate the MVC architecture. **Top-down architectural refactoring** is an activity that improves structure. It is applicable to components, connectors, subsystems, and interfaces. Architectural refactoring is the deliberate removal of architectural smells without changing the scope or functionality of the code [2]. This was achieved by separating code into model, view and controller components which resulted in a more maintainable and extendable system while at the same time not changing the actual functionality of the code.

Another refactoring technique that we used was the **Move Function/Method**. This is because there were many methods related to bookings that were inside only a few user classes, however these methods tended to be used by many other users as well. Thus these methods were moved to the Booking class where they could be used by any other class simply by creating a Booking.

Apart from this, we also refactored our code by introducing a new design pattern . For this, we applied the Facade design pattern to our code. A facade is a type of class that provides a simple interface to a complex subsystem with many moving parts. When compared to working directly with the subsystem, a facade may provide limited functionality. However, it only includes the features that clients are interested in. The benefits of the facade design pattern are self-evident: The facade "hides'' the underlying software subsystems, lowering the complexity of these systems. Furthermore, the approach promotes the loose coupling principle. Changes (modifications, maintenance) are convenient and possible at any time due to the low degree of interdependence of the individual components. A loosely coupled subsystem is easier to expand [3].

The implementation of the facade pattern included a main CovidFacade in the utilities package which from here could interact with the other facades in the other packages, namely the FacilitiesFacade and the UserFacade. This allows us to easily access separate functionality for Facilities and Users through the CovidFacade. Thus, helping in hiding the more complex code in the separate packages.

There are some drawbacks to using the facade design pattern. Because of its critical role, implementing a facade is a time-consuming and difficult task, especially if it must be inserted into existing code. In general, installing a facade interface necessitates an additional indirectionstage, which increases computing time for method and function calls, memory access, and so on. Finally, the facade pattern runs the risk of making the software overly reliant on the central master interface. However, due to the advantages of this pattern mentioned above, we have decided to go through with it [6].

New Functionality

1. Booking Modifications (Residents and Admin Staff)

Residents can choose to check their booking status using pin code, modify current bookings or change their bookings to a previous booking. Receptionists can also perform the same functions on behalf of the resident. In our previous assignment, we had created a Booking class which had all the functions needed for users to make a booking, change the status of a booking using booking ID etc. By having a single class that was only responsible for the booking part of the subsystem, we abided by the **Single Responsibility Principle**.

We also created a new interface 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 depended 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)**.

Due to the above mentioned design principles that we had implemented in assignment 2, we were able to add on the new functionality without the need for any refactoring. We simply had to create new functions for modifying and cancelling bookings in the pre existing Booking class. Since the Residents and Receptionists had similar functionalities, we were able to make use of the code in the Booking class rather than having to write separate functions in the Resident and Receptionist classes. By doing this, we are abiding by the **DRY Principle**(Do Not Repeat Yourself)

2. Admin Booking Interface

The new Admin features shared much functionality with the Resident's class. For example, both residents and receptionists have the ability to search for bookings using their booking ID as well as modify them (including reverting back to a resident's previous booking). To do this, again instead of creating the same methods in the receptionist class as was used by the resident class, we instead put the code that provided the shared functionality into the Booking class and both receptionist and resident were able to call these methods. This meant we again abided by the DRY Principle (Do Not Repeat Yourself). The notifications functionality was likely to create a strong dependency between

the Resident and the Receptionist class however,i t was implemented such that the Receptionist was able to check if an update was made directly from the web service rather than the Resident. Therefore, we were able to minimise any dependencies between the two classes (**RED principle**)

Coding Practices

In order to make our code more flexible and extensible, we have implemented good coding practices. We have tried to strictly abide by the **DRY principle**. By abstracting our code into functions, we were able to reuse that code for more efficient development. In addition, avoiding code duplication made debugging easier as we didn't have to debug every instance of code that repeated in our program [4].

We also tried to ensure that our code had **low coupling and high cohesion**. Low coupling and high cohesion are two distinct but complementary concepts. Low coupling promotes separation of unrelated parts of a codebase, whereas high cohesion promotes integration of related parts of a codebase. Low coupling and high cohesion work together to ensure our applications' maintainability, scalability, and testability. High cohesion aims to maintain close relationships between units that need to know about one another. When it comes time to extend code, it helps to find related code in the same places. Low coupling, on the other hand, aims to reduce dependencies between unrelated units [4].

Lastly, we used intuitive names for variables, functions and classes and added meaningful comments to our codebase. This would help to increase the readability of our code and less time would be spent trying to understand what the code does in the future [5].

References

[1] - Medium. 2022. *Everything you need to know about MVC architecture*. [online] Available at: <https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1> [Accessed 18 May 2022].

[2] - Barrow, S., 2022. *What is Architectural Refactoring? - Lattix Inc*. [online] Lattix Inc. Available at: <https://www.lattix.com/what-is-architectural-refactoring/> [Accessed 27 May 2022].

[3] - Refactoring.guru. 2022. *Facade*. [online] Available at: <https://refactoring.guru/design-patterns/facade> [Accessed 25 May 2022].

[4] - Educative: Interactive Courses for Software Developers. 2022. *6 coding best practices for beginner programmers*. [online] Available at: <https://www.educative.io/blog/coding-best-practices#why> [Accessed 27 May 2022].

[5] - ThoughtCo. 2022. *A Quick Guide to Using Naming Conventions in Java*. [online] Available at: <https://www.thoughtco.com/using-java-naming-conventions-2034199> [Accessed 27 May 2022].

[6] - evelopment, W. and pattern?, W., 2022. *Facade pattern: unified interface for software projects*. [online] IONOS Digitalguide. Available at:

<https://www.ionos.com/digitalguide/websites/web-development/whats-the-facade-pattern/> [Accessed 27 May 2022].