Design Rationale

The model view controller (MVC) Software Architectural (SA) Pattern was decided and applied as early as Assignment 2 when we settled on using a web application approach for the Covid19 Booking System. One of the main deciding factors was the separation of concerns the SA pattern afforded the system as a whole [1]. It allowed us to change both the views and controller without having to worry about the model, since the model depended on neither. Same goes for changing the controller without worrying about the view. Because of the segregation of the three layers, the MVC SA Pattern allowed us to organise the web app in a way that provides high extensibility and maintainability which has proven to be helpful in the implementation of additional functionalities for Assignment 3 [2].

The MVC pattern also enabled easy development of different view components for different pages [2]. This helped limit code duplication as specialised views can be reused, with its business and data logic separated. All things considered, the main reason for the selection of MVC as the SA Pattern is because of how easy it lets you build a web application, and how it makes the maintenance and extending from said web app effortless.

The downside to this decision however, apart from the added complexity, was the fact that the model could not easily interact with the view to update its contents. This became particularly troublesome for the implementation of Task 2. Due to limitations stemming from the implementation decisions made during Assignment 2, the observer pattern was not a feasible solution to this problem. The way our model and view were integrated into the web application we designed simply didn’t allow for the updating of the view through method calls like update(). Therefore, we opted to make use of an implementation specific feature to handle the live display of updated data, i.e. notifications.

Other than the MVC SA pattern, the Memento behavioural design pattern was also implemented for this Assignment. One of the requirements was to allow the user to be able to modify and undo the bookings that were made which would require snapshots of the state of the booking instance to be saved. This scenario perfectly fits the Memento design pattern that

lets you save and restore the previous state of an object without revealing the details of its implementation [1].

Due to the restrictions of the coding language we used, which is Ruby, we opted for the Memento structure with an interface instead of a nested class (see Figure 1) since we felt that Ruby's nested class behaviour does not provide enough support. The nested classes in Ruby aren't strongly enforced as it still allows its inner classes to be instantiated without being bound to its upper level class, in other words, the outer class does not need to be instantiated to be able to instantiate the inner nested class [4].

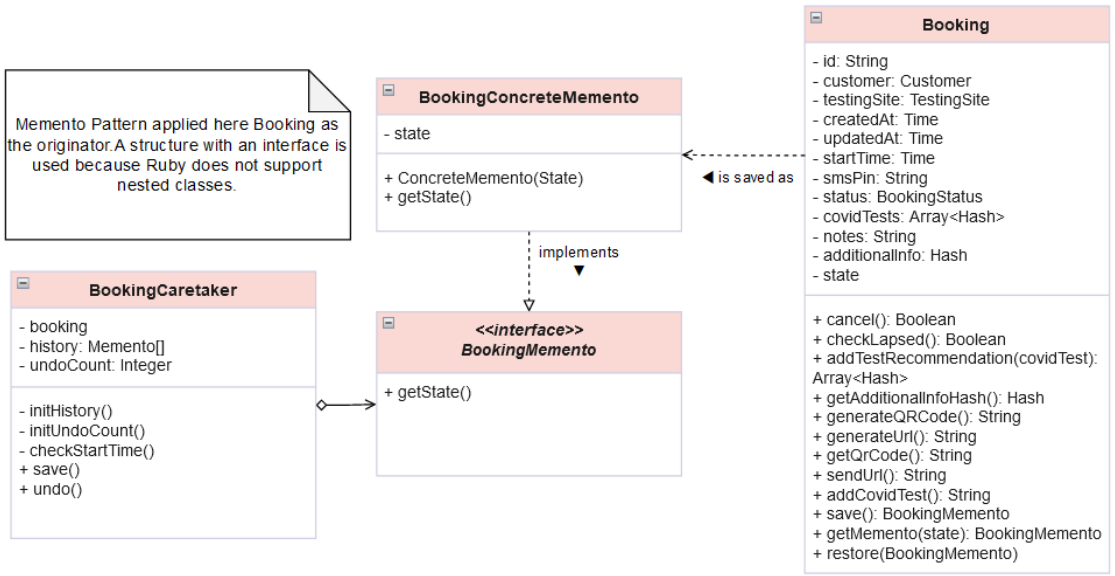

Figure 1 - Memento Design Pattern
(NOTE: not all dependencies shown)

The foremost reason for choosing to use a Memento design pattern is because of the encapsulation it provides. Snapshots of an object's state can be produced without violating its encapsulation on account of the fact that the contents of the memento aren't accessible to any other object except the one that produced it [3]. This means that the Single Responsibility Principle (SRP) would not be violated by having the Booking class also be responsible for the state saving and restoring functionalities. Instead, those responsibilities are delegated to the BookingMemento.

In implementing the new features for our application, we found our previous adherence to the Single Responsibility and Open/Closed Principles helped a lot in aiding the addition of functionalities. This can be seen most prominently in the API, Handler and Booking classes. Since the Handler classes do not interact at all with the web service, leaving that responsibility to the API classes, we were able to make updates to them without worrying about the implications to the interfacing with the web service.

Similarly, since the behaviour of single booking objects are not the responsibility of the BookingHandler class, it was easier to isolate and implement changes to single booking instance behaviours than it would've been if they were both a single class.

Furthermore, the Stable Dependencies Principle that was implemented previously made sure that the packages with classes that we changed the most, simply because they were the most volatile, had little to no incoming dependencies that we had to take into account, greatly easing extension.

In developing our application further, we realised that the Handler class was becoming more and more like a Facade. This can be seen most evidently in the BookingHandler class, where it provides a simplified interface for other classes to interact with bookings using actions that require multiple other classes to complete. The BookingHandler class handles the calling of these other classes, i.e. Booking and BookingAPI, to complete tasks like cancelling a booking. This greatly helped reduce the complexity of the client code, since it only had to rely on BookingHandler. It also reduced the number of dependencies that existed between client code and the core system, since client code would only need to rely on BookingHandler, and maybe Booking, but never BookingAPI.

No new design principles were added and no principles modified during our extension of the application. The reason for this is that very few entities were actually added to the core system during extension, and those that were didn't require refactoring of the existing system. We essentially just added on to the system using the same principles we were using before.

Despite that, we still used previous design patterns to accomplish extension of the application. The Memento design pattern that we used

implements the Single Responsibility Principle by making sure a class doesn’t store past instances of itself. In this way the Single Responsibility Principle is used in a new way in our application.

References:

[1] N. Nazar. (2022). Software Engineering: Architecture and Design [PowerPoint]. Available: https://lms.monash.edu/mod/resource/view.php?id=9880793

[2] Verma, A. "Benefit of using MVC - GeeksforGeeks." GeeksforGeeks. https://www.geeksforgeeks.org/benefit-of-using-mvc/ (Accessed 25 May 2022).

[3] N. Nazar. (2021). Object Oriented Design Principles - III [PowerPoint]. Available: https://lms.monash.edu/mod/resource/view.php?id=9880764

[4] "ruby - When to use nested classes and classes nested in modules? - Stack Overflow." Stack Overflow. https://stackoverflow.com/questions/6195661/when-to-use-nested-classes-and-classes-nested-in-modules (Accessed 25 May 2022).