Assignment 3 Design Rationale

Class Diagram: Located in repo, in docs folder as `FIT3077 - Assignment 3 Class Diagram.png`
Package Diagram: Located in repo, in docs folder as `FIT3077 - Assignment 3 Package Diagram.png`
Video Submission: Available at the following link (requires Monash google sign in): https://drive.google.com/file/d/1Eg_ajMqHfEy_UAwGY01e7N1cPjhIRTta/view?usp=sharing

MVC

- Model: *COVIDbookings, COVIDtests, COVIDtestingsites, users*
- Controller: *menus, menuitems*
- View: *views*

The largest system wide refactor was transforming our system into a passive MVC framework.

Our **model** classes are those such as *User*, *Booking*, *FacilityTestingSite*, and *Test*. These store the underlying data and can be modified using getters and setters. They are easily accessible now using our facade classes i.e. *UserFacade*, *BookingFacade*. Our **view** classes are our new *View* interface and *CommandLineView* class. They act solely to display information to the user, get information from the user, and construct the UI based on what the controller requests. Our **controller** classes are our *Menu* classes and our *MenuItem* classes. These oversee the control flow of our application, and pass information to the view and update and collect information from our model classes.

A major advantage of this is the separation of concerns. Our system has become far more flexible and extensible. We can change components without damaging others. For example, our UI currently exists in the console, which is achieved through the *CommandLineView* class. If system requirements changed and we needed a new UI, for example a GUI, we could create a *GUIView* class that implements our *View* interface and easily substitute this in. Because our controller and model are independent from our view, all we would need to do is on line 22 of *Main,* substitute in *GUIView* for *CommandLineView,* resulting in line 25 of *Menu* assigning the GUI interface as an attribute. Now our controller classes would seamlessly interact with the GUI view.

This would have been unachievable in our previous implementation because of entanglement. Displaying to the console was intertwined with all our *MenuItem* classes, and refactoring to GUI would have required large manual changes, sifting through many lines of code, one *MenuItem* at a time.

This is also beneficial the other way around. We are now able to change the logic of our controllers without affecting the view. This again would have been extremely strenuous in our previous implementation.

MVC also makes our system easier to test and debug, because each component functions independently of the others, and can therefore be tested independently.

Creating the view components also removed lots of repeated code across our *menuitems* package as we can now reuse components. Previously, each *MenuItem* was responsible for creating their own menus, displaying to the console, and recording user input. This functionality has now been seperated and housed into our view components, and taken out of each *MenuItem*. Instead *MenuItems* are now controller classes that interact with *View*. This has greatly decreased the rigidness of our code, and made it much easier to update in the future.

Appling MVC principles has drastically decreased dependencies across our system. This has reduced coupling and reduced dependencies on low-level classes (in-line with the Dependency Inversion Principle).

Overall the MVC framework provided a large benefit to our system, but the refactoring did take a lot of time, and it could be argued it added lots of complexity to a smaller system that was already functioning well. However we feel it was worth it, our design has been improved, our system is now more scalable, maintainable, and flexible with less coupling.

Observer Pattern

- Observers: *Receptionist*
- EventManager: *BookingEventManager*
- Listeners: *CreateListeners, ModifyListener, DeleteListener*

The Observer Pattern was utilised when implementing the admin interface. This pattern enabled us to alert receptionists of changes (events) occurring to bookings at the testing sites they worked at.

One of the reasons for implementing this pattern was that it allows for observers to be added and removed quite easily. The need for an open-ended amount of observers was quite important as the observers for each event change, since booking modifications don't always involve the same testing site and receptionists/admins as a result.

In addition to this, this pattern was used as it prevented an increase in coupling between *Booking* and *Receptionist*. This was because the actual notifying and updating was handled by the *BookingEventManager* and listener classes. If more events were to be added it would be easy to do without the need for major refactoring as well.

Furthermore, if the system did require expansion and other users such as healthcare workers needed updates on bookings at their testing sites as well, it would be really easy to implement with this pattern as observers do not need to be restricted to a specific type of worker for this pattern to operate.

A disadvantage of using this pattern was how it was prone to privacy leaks. Since, this pattern involves many references to arrays, observers and fields of the observers, it was common to run into privacy leaks. While this was resolved using deep copies of objects,

there may be cases where privacy leaks still occur and could cause unexpected results or introduce other bugs.

Facade

- Facades: *UserFacade, BookingFacade, TestFacade, TestingSiteFacade*

We decided to use the Facade pattern to help clarify access to our data and API, reduce dependencies on low-level implementation classes, and reduce repeated code. We found that in nearly all of the *MenuItem* classes, we would have to access the API directly, leading to a lot of dependencies on low-level classes. We would then have to instantiate a factory class directly, pass in raw data from the API, and often make a collection class (i.e. *BookingCollection*). This led to repeated code in several *MenuItem* classes, which made it difficult to change the way these dependencies worked without needing a lot of refactoring. It also meant that we would also have to constantly worry about keeping the data in sync with the web service manually by making PUSH requests after changing data, leading to inconsistencies if one part of the process was forgotten.

We fixed these issues by creating facade classes for each of our subsystems (*UserFacade, BookingFacade, TestingSiteFacade, TestFacade*). We used these to hide low-level implementation details such as API access, factories, and collection objects (i.e. *BookingCollection*) from the user, reducing the complexity of the system and client code. This also makes it much easier for us to add functionality to our subsystems without breaking existing client code. For example, if we needed to add functionality for adding users, or add a new type of user object, it would all be hidden behind the facade. By only interacting with the facade for a particular package, we can also ensure that our in-memory model is kept in sync with the model stored on the web service by making calls to the API behind the facade. It served to simplify access to our complex subsystems.

We decided to implement the facade as a singleton class. We decided to do this to enable two things: make our data classes available globally, and control access to a shared resource: the current version of our data. By using a singleton, we can maintain a ‘single source of truth’ for all our models in our application. Since the only way to access their data is through the facade, and since there is only a single instance of each facade, data access and manipulation is always passing through the same channel, and stays consistent.

One issue with using the facade pattern is that it can have the potential to become a ‘god class’ as it grows and grows, hiding more and more complexity. If it were to grow too big, we would be best to split the larger facade into smaller, more manageable facade classes. It is very easy to simply keep adding more and more methods to the facade, however it can end up complicating the system and muddying responsibilities. For example, we considered adding ‘pass-through methods’ for the getters and setters for all of User’s attributes. However, we decided that this was not a good idea, and would begin to turn *UserFacade* into a ‘god class’. Instead, we have methods in *UserFacade* that return the *User* instance that client code is looking for.

An issue with having the facades as singleton classes is that it violates the single responsibility principle, as on top of providing its services, the classes are also tasked with

managing their own instance. It is possible as well that there could be an issue with generating a large number of dependencies as the singleton class is available everywhere, to all parts of the program.

Other small refactoring tools

- Improved variable name declarations. Moved from names like ‘uf’ to ‘userFactory’, making our code more descriptive, legible, and easy to understand for members outside our team.
- Removed the use of magic strings to determine conditions. Created enumeration classes like *TestType* and *TestStatus*
- Shifted to using Object.equals() for comparisons over ‘==’ for strings and other types, because the latter does not have null safety

Other benefits

We achieved a good balance between package stability and abstractness. Our high level components like *menuitems* is a very unstable package, while *COVIDbookings* is stable. This is in accordance with the Stable Abstraction Principle, providing a good balance of extensibility and rigidity.

Our initial design developed in Assignment 2 supported all our new extensions. Our sub-systems (e.g. *COVIDbookings, COVIDtests*) encapsulated the components and functionality of the system well. Adding facade classes to these packages was therefore easy.

Adding new functionality to previous classes was not difficult as we tried to maintain the single responsibility principle in most classes in Assignment 2. Therefore, it was more intuitive where features needed to be programmed. For example, our use of menus in the previous assignment allowed us to add extra functionality to the ReceptionistMenu when it came to the displaying of notifications for admins.

Our implementation of the *user* package in the previous assignment allowed us to extend the functionality of individual users as well quite easily. An example of this can be seen with the *Receptionist* class which was given extra functionality with regards to setting a workplace (testing site) and methods for storing messages.

In the previous assignment we created an *endpoints* package, which had a sole purpose of making requests to the API. This was definitely one of the things that proved to be incredibly useful when extending the system. This package allowed API calls to be singled out in a particular package, even when the API URL changed, it only required a single line of code to be changed. It minimised the need to repeat code for new features as well when they needed to call the API.

Assumptions

- Patient = customer
- Facility staff = Receptionist & HealthcareWorker
- Admin = Receptionist
- User can only be a single type of user: patient, receptionist or healthcare worker
    - If an existing user is categorised as multiple types, our system assigns their type in this order: Customer -> Receptionist -> Healthcare Worker
- Each admin can only work at one testing site.
- Admins would only want to be notified of changes to bookings at their testing site when on the main menu, not in the middle of an action.
- Admin notifications can be cleared after being viewed once.
- When trying to restore a previous state of a booking, it is assumed that the user will know what time/venue was involved in that previous state. There would not be a need to display it to them.
- Note: assumptions regarding features from assignment 2 carry over as well.