Architectural pattern

Background

At the conclusion of Assignment 2, our codebase's state was very close to following the MVC pattern despite not considering or applying any architectural patterns. Almost every class in our codebase could either be categorised as taking care of "Model" responsibilities (managing the behaviour of the data stored in the online database, and responding to requests to get or modify this data), or "View AND Controller" responsibilities (managing the display of information and interpreting user input). For example:

- Model classes: EntityManager and its subclasses, as these classes were essentially 'local' representations of the online database's data. These classes did not conduct business logic, interpret user inputs, nor directly update our console UI; instead, they responded to requests to view the data, and responded to instructions to change the data (by sending POST, PATCH, DELETE requests to the API)
- View and Controller classes: Most other classes fell into this category. Unfortunately, during Assignment 2, we did not separate the concerns of managing the display of information to the UI (View's concerns), and interpreting user input and updating the models accordingly (Controller's concerns). For example, SearchForm (no longer a class in our system) was responsible for updating the console UI with the interface for searching for sites, as well as interpreting the users' inputs on the interface.

MVC Architectural pattern - passive MVC

Due to the state of our codebase after Assignment 2 being reminiscent of the MVC pattern, this meant that, in order to apply the MVC architectural pattern to our system, we would simply need to separate the "View AND Controller" classes into two separate classes - one for the View, and one for the Controller.

We did this by implementing two abstract classes - View, and Controller. The View declared methods responsible for updating the console UI, and the Controller abstract class declared methods responsible for interpreting user inputs to the console. The classes which represented and manipulated the various data models in the online database (for example, BookingManager, Booking, UserManager, User, etc) did not need to change as they already fulfilled the responsibilities of the Model.

For example, we implemented the Admin/Receptionist view cancelled/modified bookings feature as follows:

- Model: The BookingManager and Booking classes, which are responsible for responding to the controller's requests to access modified or cancelled bookings.
- Controller: The ShowBookingsController class which extends from Controller, which is responsible for initialising a ShowBookingsView and calling its method to update the console UI, and interpreting the user's input to the console following this. This class depends on the BookingManager class (model) in order to access the Booking instances which the Admin/Receptionist should see.
- View: The ShowBookingsView class which extends from InteractiveView (which itself extends from View) is responsible for updating the console UI with the various options which an Admin/Receptionist has when viewing bookings.

Our entire system from Assignment 2 was refactored in this fashion, and the new features for Assignment 3 implemented in this fashion too. In order to best separate the concerns of the various View, Controller, and Model classes, we placed all 'View' classes into the 'view' package, all 'Controller' classes into the 'controller' package, and all 'Model' classes into the 'model' package. This gives us, as developers, a clear indication of how to implement new features and where each type of logic should be encapsulated.

In our system, the Model package is completely independent of both the Controller and View packages, thus the variant of MVC which we implemented is the passive MVC pattern. This was our intention, as our Model classes were already relatively complete and did not depend on any Controller/View classes. Furthermore, our Model classes depend on the API, which is completely independent of our system, meaning there is no way for it to notify our system when a change has been made.

Pros:

- This architectural pattern allows for the separation of concerns of the responsibilities of: updating the UI, interpreting user input, and responding to requests to access and modify the data upon which the system relies on. Allowed us to implement classes which upheld SRP, because each class was only responsible for ONE of these concerns.
- Applying this architectural pattern allowed our system to abide by the Acyclic Dependencies Principle (ADP), because the Model did not need to know about the UI (console in our case), to tell it when it had changed. Instead, Controllers were used to inform the View when the Model had changed, telling the View to update the UI.

Cons:

- By applying the passive MVC pattern, there was no way for the Model to report that its state had changed. This is unavoidable because of the use of an API which is independent of our system, because, for example, if an Admin/Receptionist is logged in on one machine and is viewing modified bookings, and another user modifies a booking on another machine, there is no way for the API to tell our system that a change has been made, thus the Admin cannot automatically see that a change has been made, without re-running the Controller's method which asks the Model to ask the API for an update of booking statuses in the system.
- A significant amount of Architectural Refactoring of our existing system was required. This involved converting each "Controller AND View" class (described in the "Background" section above), into two separate classes - a Controller, and a View.

Since applying the passive MVC pattern allowed our classes to uphold SRP, and our whole system to uphold ADP, we were willing to accept the cons (especially the first one, which is unavoidable due to the fact that our system depends on an API which is obviously completely independent of our system).

Design patterns

Facade pattern

The facade pattern simplifies the interface between client code and a more complex system, where the client only cares about certain aspects of the complex system. In our case, the facade pattern is used to create a simple interface between the BookingFormView which updates the UI to ask a user for details of a new booking they want to create, and the online database's set of operations which can be used to deal with bookings (for instance, GET and PATCH requests). In the previous assignment, BookingManager acted as a facade to provide simplified access to the API's GET and POST methods. For this assignment, we extended the BookingManager to also provide access to the PATCH method, because the updated specifications required this access. This still provides BookingFormView with a simplified interface of the complex API, by only exposing it to the API's methods which it needs access to.

Pros:

- Having already used the Facade pattern in the previous assignment, it was easy to extend the existing facade to allow access to the PATCH method.
- The coupling between the API of the online database and the BookingFormView is minimised, since the BookingFormView conducts its interactions with the API via the facade of the BookingManager.
- SRP is upheld in the BookingFormView class since the responsibility of navigating the database's API is not given to the BookingFormView, whose responsibility is to query necessary booking information from the system user.

Cons

- Using a facade has the potential to lead to 'god objects' which are coupled to many other classes throughout the system since every class which is reliant on the 'complex system' would need to access it via the facade. This turned out not to be the case for us, since BookingManager's sole purpose is to provide a facade of the API's 'booking' endpoint and only a select few View and Controller classes depended on this functionality. Thus, this downside of the facade pattern did not affect our system.

Choosing to use this design pattern was easy - there are pros (discussed above) and the major disadvantage of creating 'god objects', associated with the facade pattern, did not affect our system.

Adapter pattern

In Assignment 2, we used the adapter pattern to make the API's JSON responses compatible with the Site, and User classes. We did this by implementing an abstract JsonAdapter class from which JsonToSiteAdapter and JsonToUserAdapter extended from. This design choice upheld OCP, making it easy for us to reuse this design pattern in Assignment 3, where we implemented a JsonToBookingAdapter child class of JsonAdapter, in order to allow for Booking instances to be constructed from the API's JSON responses without violating SRP. JsonToBookingAdapter provides an easily usable interface for the Booking class, from the API response which is in the form of an ObjectNode class. This shields both the Booking class from the ObjectNode's interface, providing an easy-to-access alternative.

Pros:

- The responsibility of adapting the JSON response to data fields in the Booking class is not given to the Booking class via the implementation of the JsonAdapter interface, which's sole purpose is to do this. Thus, Booking class is able to uphold SRP, as it is not responsible for transforming API data. Since the JsonAdapter classes take on the responsibility of making JSON data usable to Booking class and no other responsibilities, this hierarchy of classes also obeys SRP.
- If the response format to the API's 'booking' endpoint changed, the Booking class would not need to change. Instead, only the JsonToBookingAdapter class would need to change. This allows the Booking class to uphold OCP, as it is closed to change - it does not need to be altered in response to change. The JsonToBookingAdapter class would be changed appropriately, and the Booking class would not need to know about this at all.

Cons:

- Hypothetically, the responsibility of transforming the API's JSON response into a Booking instance could be encapsulated within the Booking class's constructor method, reducing the need for a new class which increases the overall complexity of our system. This would, however, cause the Booking class to violate SRP.

Memento pattern

As we were required to keep track of the previous three changes to each booking, we decided to implement a Memento pattern in the form of storing the changes along with the actual booking whenever modified.

We were able to implement this pattern as our design had bookings that were being changed, hence providing the scope to have an option for version history and being able to revert to previous bookings, which in turn was feasible since we implemented the Memento pattern.

Pros:

- This behavioural design pattern allowed us to save and restore the previous state of the object without revealing the details of the implementation, as the changes were stored as part of the booking modifications, or the originator object.
- This helped avoid invading the private space of objects as the changes were stored within the originator object themselves, in this case the Booking object. This also helped produce highly encapsulated snapshots of the originator objects state, as there is no direct access to the objects fields. We chose to make use of Memento as we could produce snapshots of the originators state without violating encapsulation.
- Using Memento also simplifies the originators code by letting the caretaker maintain history of the originators state. The saved state is also kept external from other objects, helping maintain cohesion as well as giving us an easy way to recover previous versions as required by the specifications.

Cons:

- We had to make these decisions at a cost, for instance, it might consume a lot of resources if mementos are created too often, as well as being time consuming itself as the dataset gets larger. Another privacy risk is that caretakers can track the originator lifecycle and are so able to destroy obsolete mementos. Dynamic programming languages also can not guarantee the state remains untouched, but as we are posting the booking to the web API, we should not be affected by this one.

Factory method pattern

This design pattern is used when we don't know what concrete types that the code will work with until runtime, and we only know the abstraction. In our case, we have a concrete child class of UserController to control the system differently depending on what type of user logs in (Receptionist/Admin, Patient, or Administerer/Healthcare worker). The problem is that the type of user who logs in is not known until they do log in (and this is obviously during runtime). To avoid having a complex switch statement checking the type of user who logged in, we implemented this design pattern.
Pros:

- The logic for creating different types of UserController subclasses is isolated from the PreLoginController class, which is a client of UserController. In our case, the logic for creating the subclasses is encapsulated in PatientControllerFactory, ReceptionistControllerFactory and AdministererControllerFactory. We implemented a UserControllerAssigner class to assist with this subsystem, which decides the correct type of factory to return depending on the response of the API following a user login. The encapsulation of different logical processes upholds SRP.
- By having the client of the UserController classes (PreLoginController) not depend on its concrete subclasses, it is easy to implement different types of UserControllers without changing any existing code, by simply implementing a new UserController child class, and a corresponding UserControllerFactory responsible for creating it. This upholds OCP as we would not need to change any existing code if a new type of user which requires new unique functionality was introduced to the system.

Cons

- For a small number of subclasses of UserController, the need to create a hierarchy of factory/creator classes introduces a lot of unnecessary complexity to the system via the addition of a new factory class for each concrete UserController.

We were willing to accept the con of implementing this design pattern because once the pattern is implemented once, it would be incredibly easy to accommodate new types of users which require their own unique functionalities to our system - The initial overhead of setting up this design pattern would pay itself off.

Refactoring

Architectural Refactoring

Our system was completely refactored to follow the passive variant of the MVC architectural pattern. See the above section "Architectural pattern" for the details of this refactoring.

Extract class for large classes

As discussed in the "Architectural pattern" section (above), our system contained many classes which were responsible for both managing the display of information, and interpreting user inputs. This gave the code smell of large classes, and was a violation of SRP. To resolve this, we extracted two classes from these classes - one responsible for managing the display of information, and one responsible for interpreting user inputs. For example, we refactored the "PreLoginForm" class into the "PreLoginController" and "PreLoginView". This pattern was repeated throughout the entire codebase until we had implemented the MVC pattern and had got rid of the code smell.

Move method technique for duplicate code

While implementing the various subclasses of View which implemented the InteractiveView interface (refactored to abstract class), we realised that there was a lot of duplicate code in the getUserInput() method. To avoid this duplicated code, we refactored InteractiveView to be an abstract class which extends from View, which implemented the method body of getUserInput() so that this code did not need to be repeated. All classes which formerly had this duplicate code then extend from InteractiveView. We applied the technique of moving the getUserInput() method definition into an abstract class to get rid of code duplication.

Delete code for speculative generality

We previously had a lot of classes, such as QR and PIN code, methods such as getBooking() as well as heaps of variable declarations, which were not used, but implemented due to either new functionality which superseded the old ones, or speculating on what possible features we might have in the future (speculative generality). Now, while this does not change the way the code runs, this helps a lot with readability by reducing the overall complexity of the system, and thus makes future modifications easier with no redundant code. For example, we got rid of the QR code class and the MedicalTest class which were both out of the scope of our system, but previously implemented 'just in case' it became a part of our scope in future.

Extract method for long methods

In the initial implementation of the various concrete Controller classes, we had one method, controlSystem(), which was responsible for creating a View instance and using it to update the UI, and also interpreting the user input to the console. This led to the controlSystem() method being very long in every concrete child class of Controller. To get rid of this code smell, we extracted a method 'interpretUserInput()' which is responsible for interpreting the user input. The controlSystem() calls this method, instead of containing the logic to interpret the users' input, making the code more readable and thus more maintainable.

Rename classes for mysterious names

Upon architecturally refactoring our system to fit into the MVC architecture pattern, we decided to rename our classes to reflect this. Since we split many of our classes into a View and a Controller, we suffixed their names with 'Controller' and 'View' respectively, such as 'ShowNotificationsView' and 'ShowNotificationsController'. This helped to avoid having mysterious name double-ups between our view package and controller package, making it easier for us, as developers, to work with our system.