# MA_Lab04Team10

## **MVC Architecture**

Our group has decided to utilise the Model View Controller Architecture for our program. The benefits of using MVC is that we get to separate the UI logic from the business logic. This allows us to easily focus on one component at a time rather than as a whole.

The models in our program are the classes that interact with the API. Because controllers never interact directly with the API. These models are the ones who will provide the necessary data back to the controller to perform the intended functionalities. The models are also responsible for maintaining the data and responding to requests from the controllers to update the data.

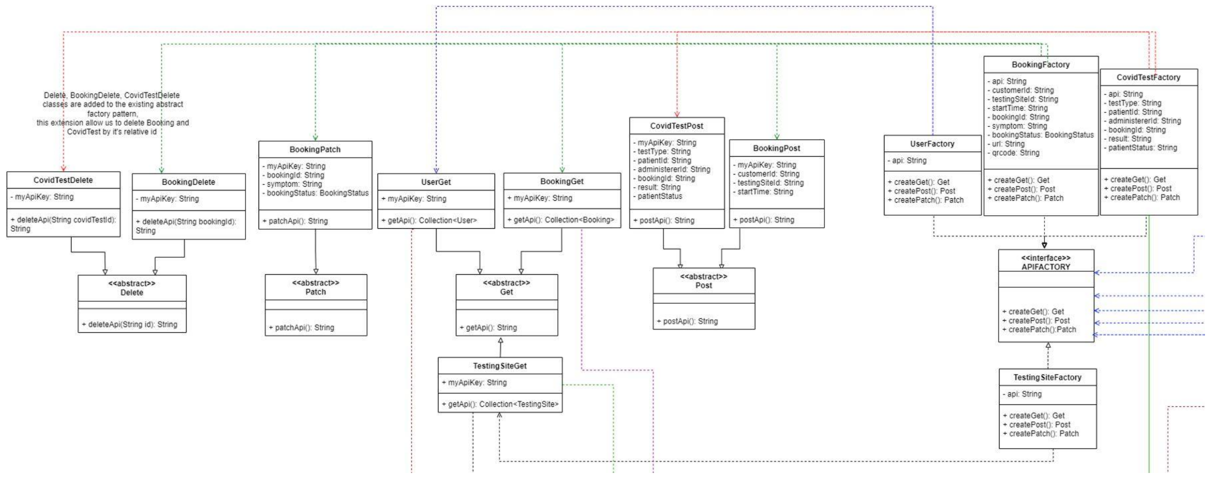
The views in our program are the HTML/CSS. These views will present the data collected from the models that call the API to the user in a graphical user interface. These views will contain the UI components that allow users to interact directly with the system and it will collect these user inputs and send it to the controller. Examples of our views include the Login Page, Admin Panel Page, Onsite Booking Page, Homebooking Page etc.

The controllers are the backbone in the entire program. They connect both the model and view component together which makes them the intermediary. Because the controllers will only perform business logic and do not need to know how to retrieve the data and present the data as those are handled by the models and views.

The advantage of using MVC is that since each component is independent on their own, integrating new requirements or modifications into our program will not take much effort as our program can easily accommodate it. Furthermore, debugging the program is made easier as the program is separated into different layers

The disadvantage of using MVC is that it introduces high complexity as it takes time to understand how the overall architecture performs.

## **Abstract Factory**



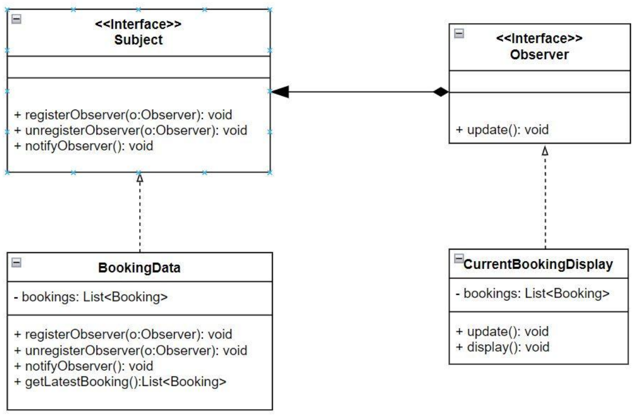
We applied the abstract factory design pattern in assignment 2. For assignment 3, we added a new class called BookingDelete which performs the Delete Request in the API. Since abstract factory isolates concrete classes from the client classes, we were able to create many different classes for each web service request such as BookingGet, BookingPost, BookingPatch and BookingDelete. By applying this design pattern, we are able to call on the respective factory classes in our controller classes to process the data and render it into the user interface classes.

The advantage of this design pattern is that it is easier to add new methods in the APIFactory class and any classes that implement the APIFactory interface will just need to override the relevant methods to perform the intended functionalities. Furthermore, it supports the Open/Closed Principle as it is open for extensibility but closed for any modifications as it may disrupt the child classes that extend from it.

The disadvantage of this design pattern is that it makes it harder to introduce new types of classes or methods as an abstract factory typically fixes the same set of classes to be created. Doing so would require changing the interface and this will affect all the subclasses and thus break the Open/Closed Principle.

**Singleton**
When we looked back at assignment 2 requirements, our program utilises the Authenticate Singleton to perform the login functionality for different user types. This design pattern has extended to assignment 3 where we needed to restrict access to certain pages for different users. This Authenticate Singleton will return the same instance to its own class whenever an instance of it is called in any of the controller classes.

**Observer**
We chose an observer design pattern to retrieve our api data because it helps to implement a subscription mechanism to the system which notifies the change in the api data we are observing. This is especially true for our real time booking webpage which is only interested in the change in booking list in the api.



<<Interface>>
Subject
+ registerObserver(o:Observer): void
+ unregisterObserver(o:Observer): void
+ notifyObserver(): void
<<Interface>>
Observer
+ update(): void
BookingData
- bookings: List<Booking>
+ registerObserver(o:Observer): void
+ unregisterObserver(o:Observer): void
+ notifyObserver(): void
+ getLatestBooking():List<Booking>
CurrentBookingDisplay
- bookings: List<Booking>
+ update(): void
+ display(): void

During our implementation, we created an Observer interface class which is used to observe the state of another object and a Subject interface class which is the object being observed.

The Subject class contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Our Subject also contains a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update. Other than that, we create a CurrentBookingDisplay class that inherits the Observer which will then perform an update action in response to the notification issue by the fit3077api publisher. Moreover, a BookingData class is created and subscribed to allow us to retrieve the latest booking data.

The main advantage of this design pattern is it handles the 1-to-many relationships well between objects, as one of the objects in the many side is updated, its dependent object will notice the change as well. Moreover, this design feature which obeys the Open-Closed principle provides code extensibility that allows us to have new subscriber classes in the future without having to change the subscriber code. Other than that, we can reuse subjects or observers independently of each other.

The disadvantage of this is that subscribers are notified in random order and also it does not scale well when there are too many subjects. An observer can be observing multiple objects as well, this will complicate the design and introduce entangle dependency. This can be solved by using MVC architecture which puts all the Subject into one package, so when there is a need we only need to update the view for that purpose. The model will handle the requests from different kinds of data. The view will show the user the kind of data they want to watch. The controller will handle the user request.

**Acyclic Dependency Principle**

Since our program adopts the MVC architecture design pattern, our program was able to separate the business logic and UI logic into their own packages. The Model package is a standalone package that handles the data logic and the Controller package will depend on data from the Model package to process the relevant data for the business logic. The Controller package will also depend on the View package to render and display the user interface. As for the dependency between Model and View package, they will have no dependency with each other as the controller package will act as the intermediary.

Hence, our program has applied the Acyclic Dependency Principle and this promotes decoupling for the packages as any modification can be done independently. Furthermore, MVC is a one way dependency which helps in eliminating the Morning After Syndrome as modifying the View package will not affect the Model or Controller packages.

**Common Reuse Principle**

Since all the API classes are grouped together under the API Package. CRP states that all classes that are reused together should belong to the same component. As our program heavily calls upon the API classes in more than one controller classes, grouping the API Classes will create a high cohesion as the classes are focused on what it should be doing. Furthermore, CRP helps us in promoting extensibility as we are able to add a BookingDelete class in the API Package which is needed for the new requirements.

**Common Closure Principle**
Based on the Common Closure Principle, our program applies the abstract factory design pattern in the API Package. This means there will be an interface called API Factory in the package. CCP states that all classes in the package should obey the Single Responsibility Principle, therefore by grouping the APIFactory interface and the classes that extend the APIFactory in one package, we are able to close the classes together under the same kind of changes.

**Single Responsibility Principle**
To avoid making our Booking controller a god class, our team decided to create a new controller called ModifyController which performs all the functionalities needed for the new requirements that are related to Booking Modifications. Our team also created two new controller classes called AdminPanelController and RealTimeBookingController. The AdminPanelController will focus on displaying all bookings in a testing site as well as handling any modifications made towards those bookings. The RealTimeBookingController will instead focus on notifying admins in the same testing site or other testing site of any updates made towards the bookings.

By separating the functionalities into two controllers instead of one big controller, we applied the Single Responsibility Principle as each controller only served one concrete purpose. Furthermore, this promotes maintainability and extensibility as all the logic is one place.

The disadvantage is that each class will be smaller and the design will be more complex as new classes are introduced to serve a singular purpose.

**Open Closed Principle**
In Assignment 2, we made an APIFactory that has createGet(), createPost() and createPatch() methods to allow the child classes to inherit those methods.

In Assignment 3, the open closed principle was applied when we extended the APIFactory to include a new method called createDelete() of type Delete. By doing so, we are able to create new factory classes that override and utilise the createDelete() method.

The advantage of this design principle is that it becomes easier to incorporate new features into the classes without modifying the core methods of the classes.

The disadvantage is that it will make the class more complex as new methods or new classes can only be introduced or extended and leaving the core methods untouched and this will complicate the entire code structure.

**Dependency Inversion Principle**
The Dependency Inversion Principle was applied in our controller classes such as ModifyBookingController when retrieving the appropriate factory classes. Our Controller class does not depend concretely on the BookingFactory class but instead on its abstraction APIFactory interface. This follows the Open/Closed Principle as we are able to add more factory classes that implement the APIFactory interface. An advantage of this principle is that it helps to promote loose coupling between the factory and controller classes.

## Refactoring Techniques

### Extract function/Method

For our program to keep displaying updated data in the UI, our controller class will need to make a request to the API and fetch the latest data back to the controller. This means our controller will have to call the respective factory classes to retrieve the relevant collection list. Instead of duplicating this long line of code in many of the controllers, we have extracted out the relevant components into their own method (e.g. getBookingList(), getTestingSiteList(), getUserList() etc.). This helps to improve code readability and obey the Don't Repeat Principle. Furthermore, code maintainability is made easier because all the code is in one method instead of all over in other places.

### Extract Class

We have applied this refactoring technique in the TestingSite class. In our TestingSite class, we have a method called getAdditionalInfo which returns the TestingSiteStatus class. The TestingSiteStatus contains all the attributes in the additionalInfo field in the Testing Site API. By refactoring it into another class, we have applied the Single Responsibility Principle as well as reducing the size of the class. This improves code readability and maintainability as we can simply debug easily into either of those classes.

## Reference List

Gudabayez, T. (2021). *Understanding SOLID Principles: Dependency Inversion.* Retrieved From https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f#:~:text=The%20Dependency%20Inversion%20Principle%20(DIP,Details%20should%20depend%20upon%20abstractions.

JournalDev. (n.d.). *Observer Design Pattern in Java.* Retrieved From https://www.journaldev.com/1739/observer-design-pattern-in-java

Refactoring Guru (n.d.). *Extract Class.* Retrieved From https://refactoring.guru/extract-class

Refactoring Guru (n.d.). *Extract Method.* Retrieved From https://refactoring.guru/extract-method

Svirca Z. (2020). *Everything you need to know about MVC architecture.* Retrieved From https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1#:~:text=%2DMVC%20is%20an%20architectural%20pattern,the%20view%20whenever%20data%20changes.

TutorialsPoint. (n.d.). *Design Patterns - Observer Pattern.* Retrieved From https://www.tutorialspoint.com/design_pattern/observer_pattern.htm