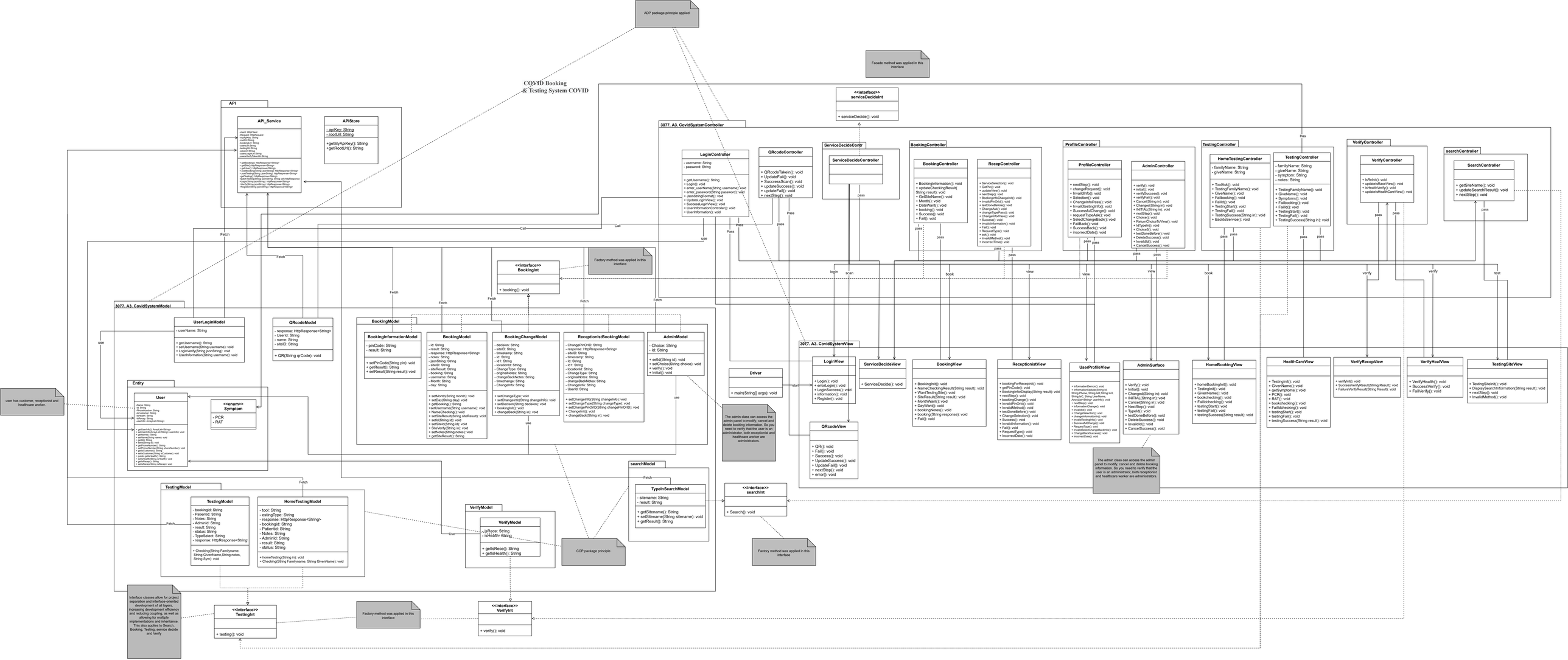

ADP package principle applied
Facade method was applied in this interface
COVID Booking
& Testing System COVID
API
API_Service
APIStore
- apiKey: String
- rootUri: String
+getMyApiKey(): String
+getRootUri(): String
<<interface>>
serviceDecideInt
+ serviceDecide(): void
3077. A3. CovidSystemController
LoginController
QRcodeController
ServiceDecideContr
ServiceDecideController
BookingController
RecepController
ProfileController
AdminController
TestingController
HomeTestingController
TestingController
VerifyController
searchController
SearchController
has
Fetch
Call
Pass
use
pass
login
scan
book
view
verify
test
Factory method was applied in this interface
<<interface>>
BookingInt
+ booking(): void
3077. A3. CovidSystemModel
UserLoginModel
QRcodeModel
BookingModel
BookingInformationModel
BookingModel
BookingChangeModel
ReceptionistBookingModel
AdminModel
3077. A3. CovidSystemView
LoginView
ServiceDecideView
BookingView
ReceptionistView
UserProfileView
AdminSurface
HomeBookingView
HealthCareView
VerifyRecepView
VerifyHealView
TestingSiteView
start
Driver
+ main(String[] args): void
The admin class can access the admin panel to modify, cancel and delete booking information. So you need to verify that the user is an administrator, both receptionist and healthcare worker are administrators.
Entity
User
<<enum>>
Symptom
- PCR
- RAT
user has customer, receptionist and healthcare worker.
searchModel
TypeInSearchModel
<<interface>>
searchInt
+ Search(): void
TestingModel
HomeTestingModel
VerifyModel
CCP package principle
Interface classes allow for project separation and interface-oriented development of all layers, increasing development efficiency and reducing coupling, as well as allowing for multiple implementations and instance. This also applies to Search, Booking, Testing, service decide and Verify
<<interface>>
TestingInt
<<interface>>
VerifyInt

FIT3077 Assignment 3

Team 8

Name: Houxuan Zhou ID: 30066948

Yueke Zhou ID: 29606764

Design rationale

Introduction:

For the Assignment 3, I follow the process: design the new design pattern,
refactoring the code from Assignment 2, implement the code into MVC and different design pattern.

Architecture design

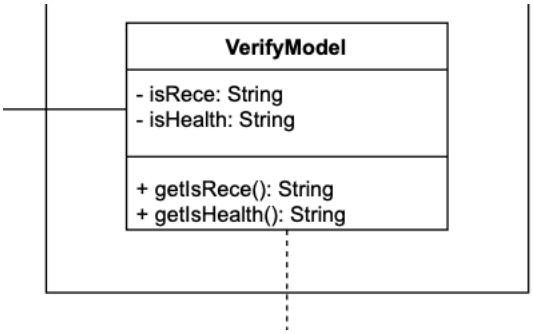
First for my Assignment 2, I didn't use any of the architectural pattern, for assignment 3, I decided to use the MCV(Model-Controller-View) architecture. The advantage is because it's a method of organizing code in a way that separates business logic, data and interface display, bringing together numerous business logics into a single component and reducing coding time by not having to rewrite business logic while needing to improve and personalize the interface and user interaction. Because due to study on the assignment 2 and 3, I realize that there is many models can be reused. And the sometimes the view need to be changed to format more aesthetic or easy to read. MVC helps me to solve the problem. because the view layer is separated from the business layer, which allows the view layer code to be changed without recompiling the model and controller code. Similarly, changes to an application's business processes or business rules only require changes to the model layer of MVC. Because the model is separated from the controller and view, it is easy to change the data layer and business rules of the application. so I can change the view simply without touch the logic model. And Controllers can be used to link different models and views to fulfil the needs of the user, providing a powerful means of constructing applications. Given a number of reusable models and views, the controller can select a model to process according to the user's needs, and then select a view to display the results to the user. So many function can be reused due to the MVC design and it decrease the code line and complexity.

Refactoring technique

For refactor my Assignment 2 to MVC architecture, I use Type generalization, Field encapsulation, extract method and rename method.

Rename method is applied for rename the method to make the MVC structure more clear. Because MVC method requires a huge amount of passing through the model to controller to view, the passing method should be more clear otherwise it will cause confused. This technique can better express its use.

For type generalization, I realize that in my assignment2, there is a lot of object or function share the same base class. So for the verify, I use verify model to get the user' s identity, in my assignment 2, I verify the health care worker and admin twice, but in Assignment 3 I create a new class to help them verify and pass the result, it save the code line, this refactoring allows for a clearer structure and increases the maintainability of the code. The same verify loop will not be used twice, even further for more identity check, there is only one class will be used.



Also because my assignment 3 have a lot of public variable, it will damage integrity of the code. the so I use **Field encapsulation**, and I provides the access method for the controller to set the data. This type of refactoring hides variables that are not relevant to external callers, reducing the coupling of the code and reducing the probability of accidental errors.

For the **extract method,** in my assignment 2, there are lots of code repeat in the different method in same class, so I extract them and build a new method, by using the method calling to reduce the workload in the class, This refactoring makes the entire program more structured and thus more readable.

Package principle

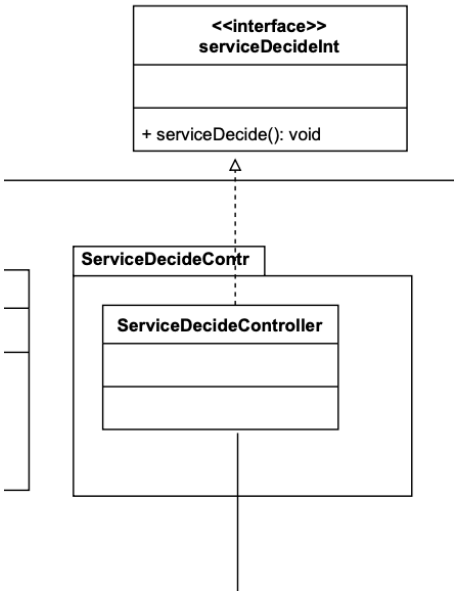
For the package principle I followed ADP(Acyclic dependences principle), There are three main package in my design and a web service, the three main package are Model, View and Controller, but the Model and view package are not connected together, it avoids acyclic dependences

Also I follow the CCP common closure principle, all the package in the Model and Controller package are divided into sub package, which means when the new function added, the package will only make change in the own package and won't affect the others.

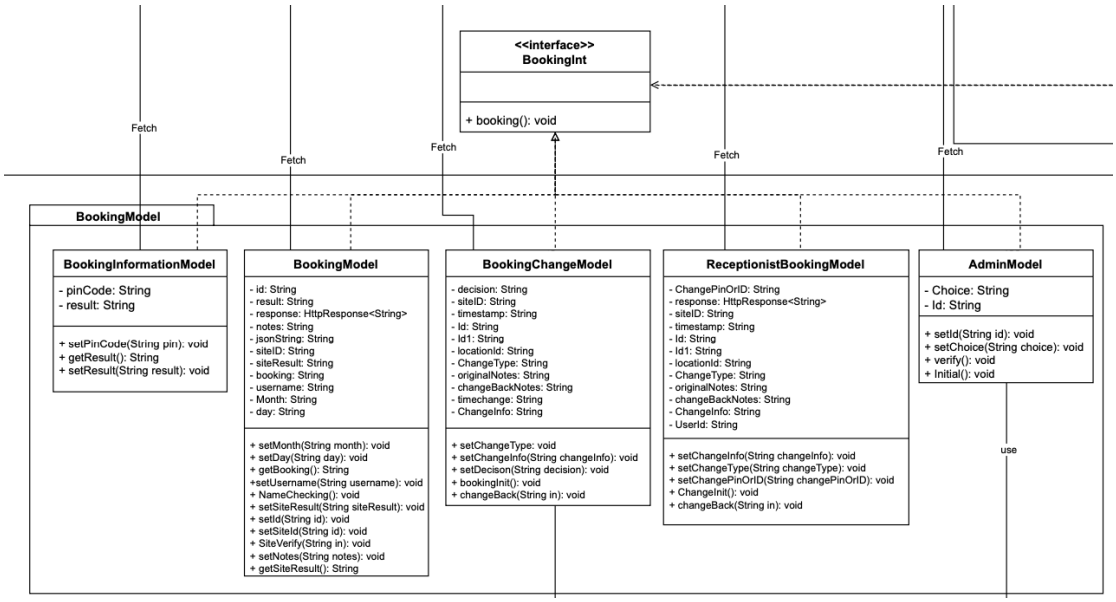
Design principle:

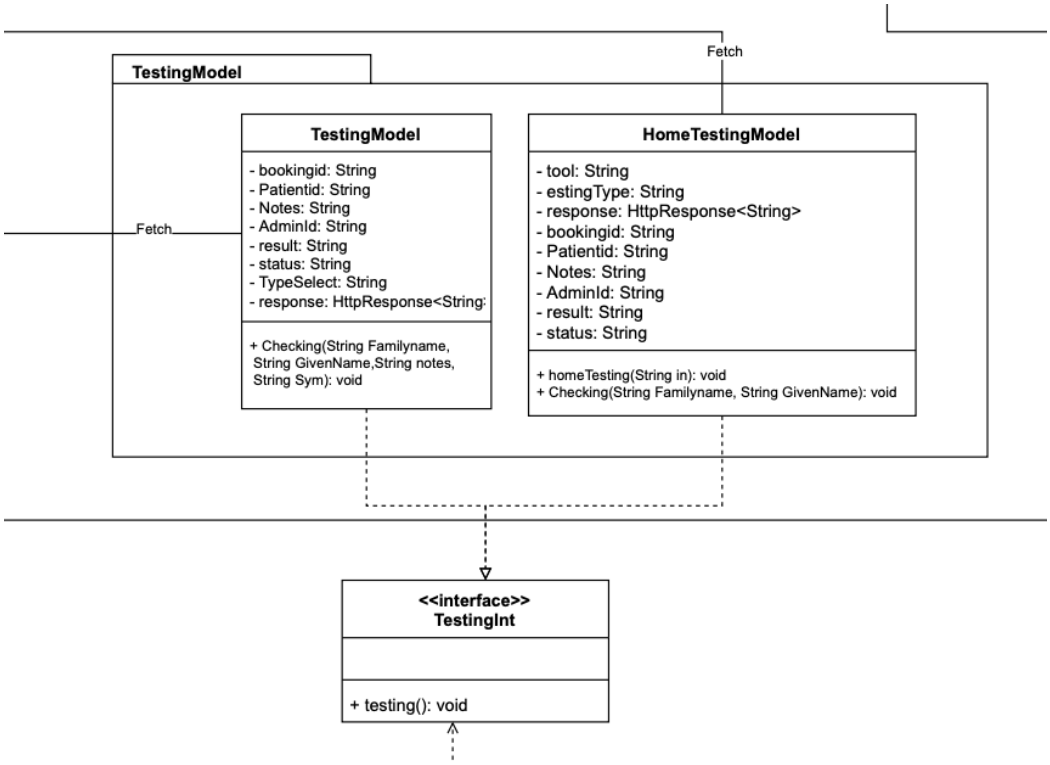
For the Assignment 3, I used the Façade principle and factory method principle.

For the **Façade principle,** I applied an interface on the service decide, and behind the service decide is different function to be choose. Facade pattern can hide the complexity of the system and to provide an interface for clients to access the system. This type of design pattern is a structural pattern. It adds an interface to an existing system to hide the complexity of the system. So it hides the complexity of different function. It helps the client not get confused by the function behind the service decide



For the **factory method**, because of the high use of the same function, I applied interface to control all the class by different function, I applied a booking interface, testing interface, verify interface and search interface. As we can see that for assignment 3 the booking interface are used many times, this is the convenient of factory method. When the programmer wants to create an object, he only needs to know its name. At the same time, the factory model is highly extensible. If you want to add a product, you can only extend a factory class. And it can shield the specific implementation of the product, which helps to hide the complexity of the functionality. So for example. In the booking model, the booking interface helps specify the method and extend to different class. The programmer only need to read the interface and develop different booking requirement.





SOILD principle

For the SOILD principle. I applied the **single responsibility principle**, each one of the class take one responsibility which mean one class do one thing and have only one reason to change. One example is merge conflicts, when the same file is modified in a team, if the SRP principles are followed, conflicts rarely occur because the file has only one reason for the change and even if a conflict arises it is easily resolved. Also the principle make the structure more clear.

I also applied the **open/close principle**, for further extend of the application, I added a lots of interface to help the programmer to develop the program. considering the future development of this program, we may need to consider features added to the system, for each feature I have added an interface to extend its functionality, we can add new features without changing the existing code of the class. For example. The search method as the A1 mentioned, it can be searched by viewing map. The extend function can fit the open/close principle, we can add the function without modify the existed classes.

References

Nazar, N. (2021). *Week 3 Object Oriented Design Principles*

Nazar, N. (2021). *Week 4 Object Oriented Design Principles - II*

Nazar, N. (2021). *Week 6 Software Design Patterns - II*

https://lms.monash.edu/course/view.php?id=135518

work contribute: