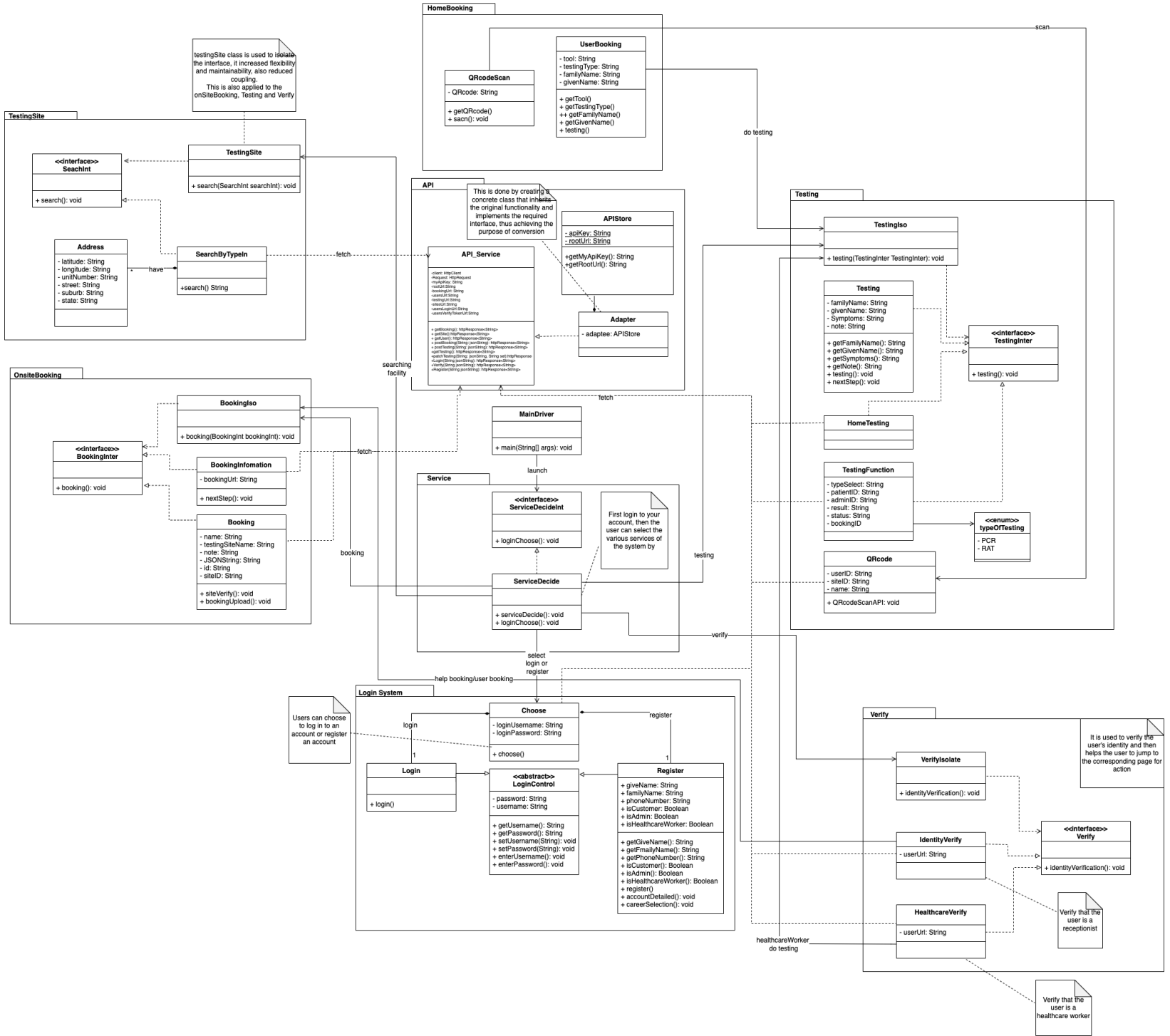
COVID Booking
& Testing System COVID



FIT3077 Assignment 2

Team 8

Name: Houxuan Zhou ID: 30066948

Yueke Zhou ID: 29606764

Design rationale

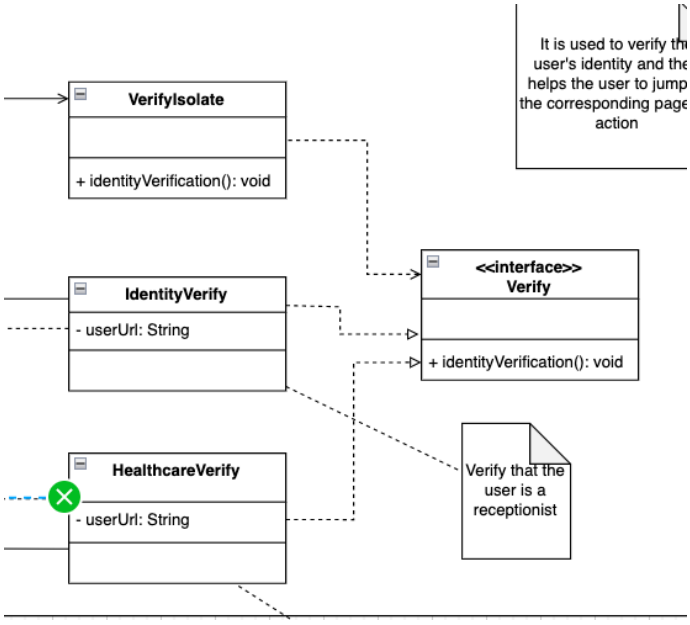
When design the system, the following SOILD principles and patterns have been applied to improve the extensibility and understandability of the whole covid- system.

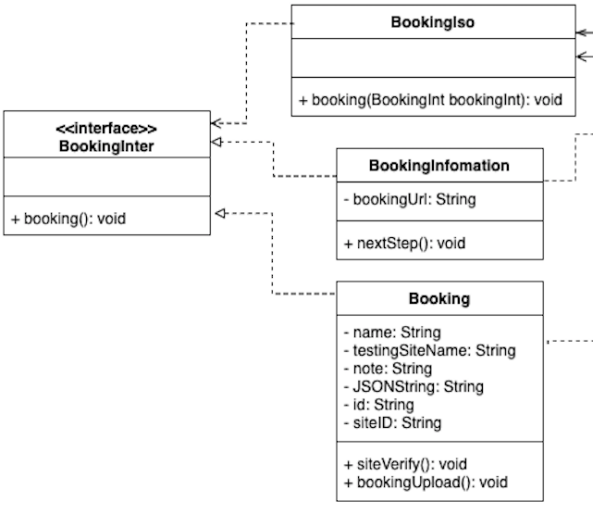
For the SOILD principle. I followed the rule of **single responsibility principle**, all the class take one responsibility, A class should only do one thing, a class should only have one reason for change, and it is important to adhere to the principle of single responsibility. For example, the service decide class will only help to sign which service you want to choose, A group of people may modify the same project, and they may modify the same class for different reasons, which can lead to conflicts. Secondly, single responsibility makes it easier to version. If you follow the SRP principle, you can tell on git that it's a commit related to storage or the database based on the file. Another 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 I follow the rule of **Open/Close principle**, in consider of future development of this program, we might need to consider the function added of the system, for each function I add an interface to extend the functionality, We can add new functions without changing the existing code of class. When the open / close principle helps us modify existing code, we reduce the risk of creating potential bugs. All classes are designed for easy future expansion. This can help that the extension of the future design. So for different function, I added interface to the different class so it helps that for different type of function added, for example, the booking interface might have add booking, delete booking, the use of interface can help that the further function add

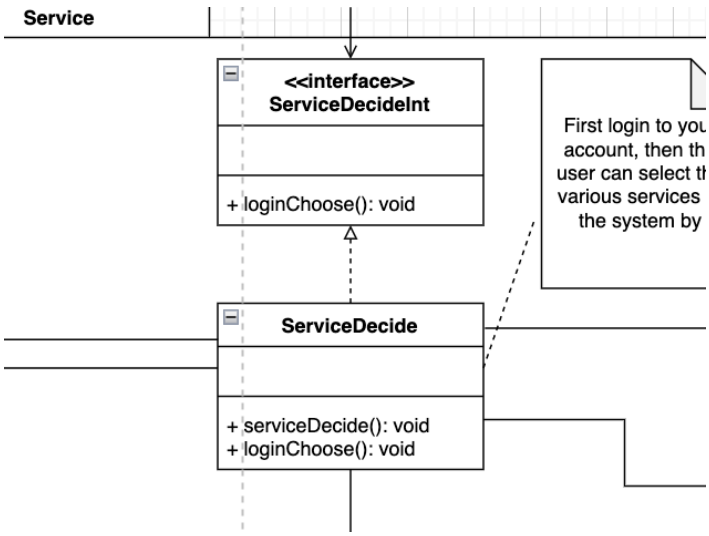
The rule of **interface segregation** principle is applied, The principle of interface isolation suggests that clients should not be forced to implement interfaces that they will not use, and should group methods in a fat interface and replace it with multiple interfaces, each serving a sub-module. So for the verify class, the booking class, and the testing class, I added a class to accept the initialization of the different classes, and passed it to the interface, this design ensure that What interfaces the client needs is what interfaces are provided, eliminating those that are not needed, which requires refinement of the interface to ensure its purity.

For the design pattern I use **Factory Pattern**, the factory pattern, The creation logic is not exposed to the client and points to the newly created object by using a common interface. When the programmer wants to create an object, just know its name. Also factory pattern has high scalability. If you want to add a product, you can only extend a factory class. And it can shield the specific implementation of the product, this helps that hide the complexity of the function, for example, the verify interface, all shows to the people is just a interface, but behind contains that multi type of verify, and also for further identity verify, this design can help further different type of verify added to the system. But this design increases the number of classes in the system exponentially, increasing the complexity of the system to some extent, and also increasing the dependency on system specific classes. Also for the Testing, I add a interface, this can help that provide a simple factor method and if there is any different type of testing, or you want to delete testing, you can use the interface to expand the function.

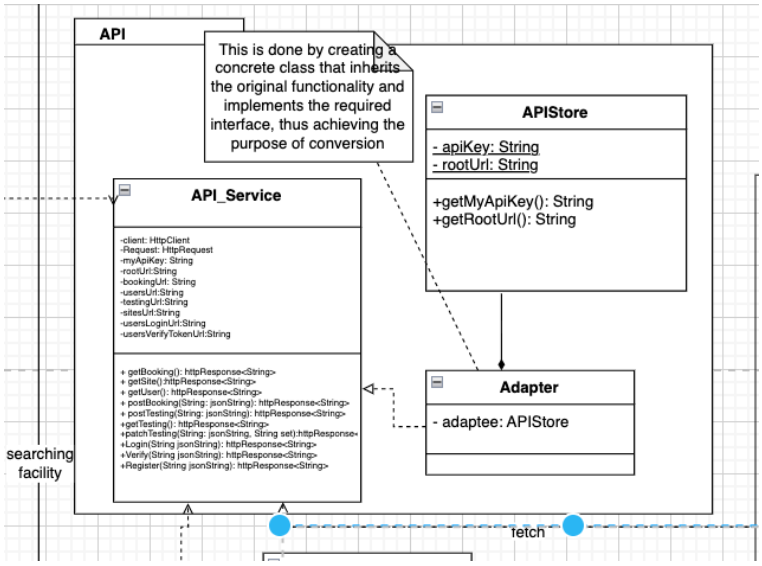




Also I used **Facade Pattern** to hides the complexity of the system and provides an interface for the client to access the system. This type of design pattern belongs to structural pattern. It adds an interface to the existing system to hide the complexity of the system. For the driver class, it will goes to the service decide class, but service decide contains a lot of interface, use the Facade pattern can increase the security, and define the entry of the system.



Also for the design I create a API class to store all the API service and returns a httpresponse body that help for future API adding or fix for the programmer.



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

Working contribution:

HouxuanZhou
34 commits (hzho0042@student.monash.edu)

Yueke Zhou
14 commits (yzho0077@student.monash.edu)