**FIT3077- Software Engineering: Architecture and Design**

# Assignment 2: Building High-Quality Software with Design Principles and Patterns

## Design rationale

For our UML class diagram in this assignment we carried over many but not all of the classes from the previous assignments class diagram because they were relevant considering the scenario is essentially still the same. We found it was necessary to remove some classes and add new ones to capture the requirements of the assignment and also better adhere to the design patterns we learnt in the recent weeks.

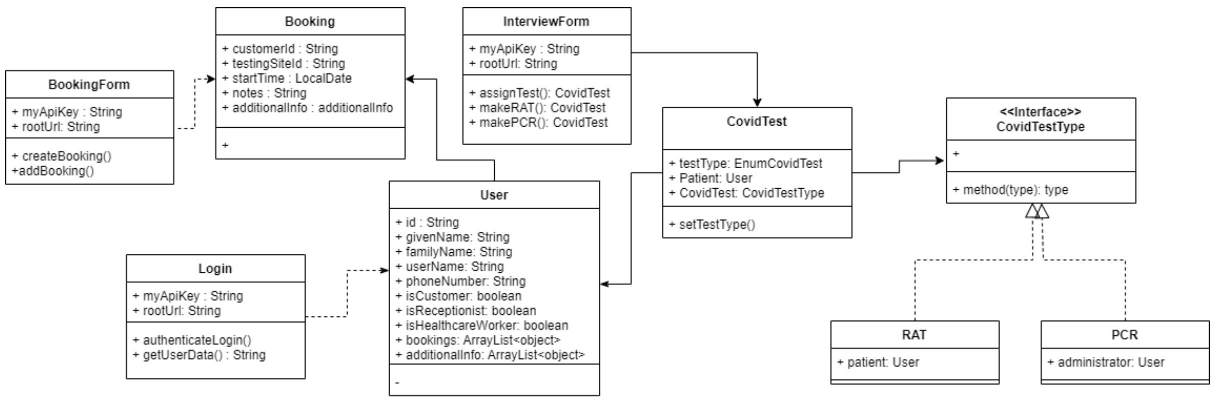
In terms of creational patterns, the Factory method pattern was in mind when creating the covidTestType interface. Although it could also been equally beneficial to make it an abstract class, the high quantity of tests consumed/carried out by both the 2 types of covid tests, RAT and PCR tests better suit interface design pattern because these 2 classes do not necessarily share the same attributes and have different implementations of their methods. The advantage is not so significant in this case but it is still good to have the option to add more functionality and classes to the interface for future scenarios like a new type of test. This also better adheres to the open-closed principle as we can add more subclasses that implement the method of the parent class differently without breaking any code. The single responsibility principle is also reinforced because each product class implements its own method that can be easily maintained and bugs isolated.
A potential disadvantage of this pattern is that perhaps more classes are needed and thus requires more memory and system resources that can be limited on some devices. A balanced design is important to achieve good performance and productivity.

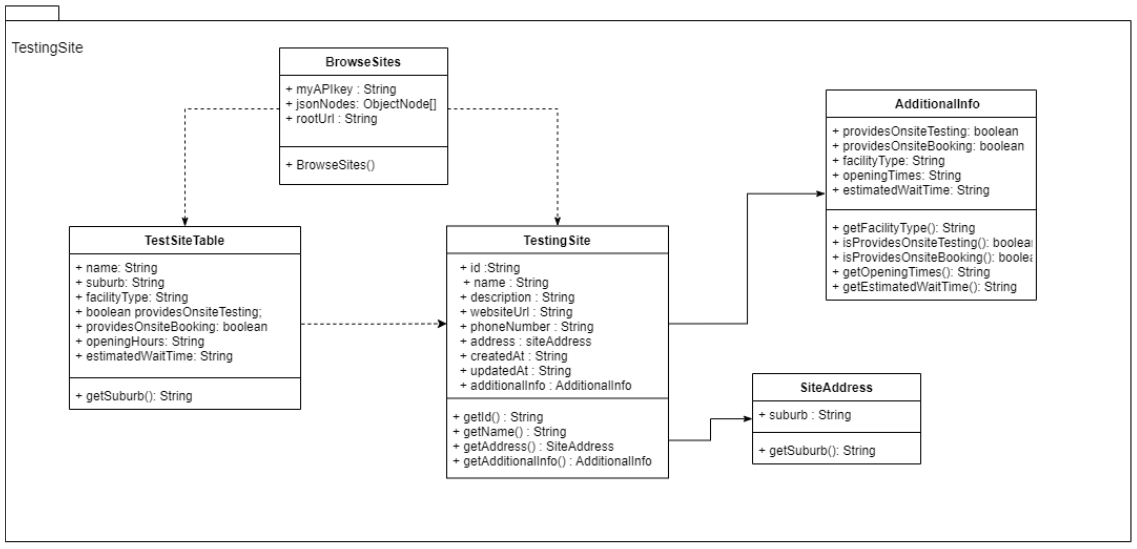
Another creational pattern that was considered was the Singleton pattern. The advantage of this pattern, although not explicitly followed by the user object in our design, is that a single object can act as a global variable that exists as one of its own kind where all classes can call it. This is useful when that object has relationships with most of the classes and the attributes and methods will be easy to access. The disadvantage of the Singleton pattern is that it violates the single responsibility principle because there may be many dependencies between the Singleton object and other classes and may mean testing and maintenance is difficult.

In terms of structural patterns, the adapter pattern was in a sense mimicked because of its advantage of allowing different objects with incompatible interfaces to collaborate. In our case although we did not manage to implement this pattern through use of interfaces, the testingSiteTable class allowed the browseSite using the testingSites object through filtering the required attributes for what we needed.

The facade pattern comes with the disadvantage of there being the possibility of a class becoming a god class that is coupled to all classes of the system which goes against the single responsibility principle.

Overall after coding, we realised that it would have been beneficial to implement more or these design principles,





## References

Learn:https://vaadin.com/docs
Create new projects:https://start.vaadin.com
Software management: https://maven.apache.org/
https://spring.io/projects/spring-boot
https://start.spring.io/
Tutorials:
Vaadin login forms
Springboot Vaadin Tutorial