FIT3077 DESIGN REPORT

Assignment 2

Cleshan Warusavitarne
Ethan Fang

Design Patterns

Singleton

The Singleton design pattern was utilised throughout the development of the program. Singleton pattern was shown in classes OnSiteTesting, Login, OnSiteBooking, HomeBooking and TestingSites. Singleton ensures that only one instance of an object is created, which in this instance was the instance of the following classes said above. This makes our constructor private to ensure no other class creates the object and we can call the methods by first getting an instance of the singleton class through the static method getInstance(). This allows us to then call the other methods in those classes.

Advantages

- Singleton improves code connectivity preventing the ability of other objects from instantiating their own copies of the class, ensuring that all objects are accessed by the single instance
- By creating a global access point, it allows us to share resources such as the API data and avoid instantiation overhead. This allows us to save efficiency and not waste memory in needing to initialise a new object every time we call a new Api, and instead use the one instance that we already have created.

Disadvantages

- However, singleton violates the *Single Responsibility Principle*. This is because classes should be able to control the number of instances it can create.
- By creating a single instance in a class, it could make it very difficult in the future to maintain the code during future unit testing. This is because it introduces a global state to the application as you are providing a global access point to that instance to other classes such as the controller classes in this scenario
- This also promotes tight coupling between classes. Relating back to the maintenance, this could be an increasing problem if the refactoring of code becomes much more difficult since a change in one class such as the Login class, will have effects on other components such as the LoginSignUpController class.

Factory method

The factory method is a way in which there is an interface for creating objects in a superclass but allows the subclasses to alter the type of objects that will be created. The factory method was utilised through our booking abstract class and our generator interface which is an interface for all the generable items such as URL, QRCode and also the PinCode.

Through this, the subclasses of the booking class which are the homebooking and the onsite booking can both generate the above items. The reason this was chosen was due to the following below:

Advantages

- You avoid tight coupling between the creator and the concrete products. Which in this case, the creator would be the booking class and the concrete product would be the generable items.
- *Single Responsibility Principle.* Since the product creation (booking class) is in one section of the program it makes the code easier to support and also follows the single responsibility principle.
- *Open-Closed Principle.* The open-closed principle states that your classes should be open for extension but closed for modification. Through the use of these factory methods for the generable items, we can easily introduce new generable products into the program without breaking the existing code.
- The implementation only interacts with parent abstract classes or interface and therefore can only work with classes that implement or extend those abstract classes. This, therefore, allows for greater cohesion between the classes.

Disadvantages

- One of the disadvantages of this is that the code may become more complicated as we need to add lots of subclasses to one class or interface to implement the pattern.
- Another disadvantage includes that by adding a new product it will require the extension of the interface which means all of its derived concrete classes must change.

The factory method was ultimately used therefore in our design between the booking system and the generable items as a way to ensure that if in the feature more generable items would be created, it is simple through the addition of another concrete product.

Strategy

The strategy method is a behavioural design pattern that makes a set of algorithms that can be used interchangeably. This was used through our onsite testing class which links to a Covidtype Test interface in which the onsite could either choose to use a Rat Test or a PCR, making the choice interchangeable.

Advantages

- Open/Closed Principle. One of the reasons this was chosen was that it follows the open/closed principle in which we could introduce new strategies without having to change the context. For example, if in the future they added more types of tests it could easily be added to the CovidTest Interface and thus also be added to the context.
- Dependency Inversion Principle- Another reason that this was chosen was that it follows the dependency inversion principle which states that high level classes shouldn't depend on low-level classes. By Having the high-level class of onsite testing being dependent on the interface instead of the low-level classes (RAT and PCR) it follows this rule. In doing this it removes the direct dependency on the details resulting in decoupling and also easier reuse.

Disadvantages

- The application must be aware of the difference between strategies to be able to select a proper one. In our scenario, the application must know when to give a rat test and when to give a PCR.
- The application Context class also creates multiple Strategy objects in order to be able to call the classes method. In this situation, the application needs to create not only two strategy class objects in PcrTest and RatTest but also creates the context class object in OnSiteTesting. This makes the application have tighter coupling and harder to maintain in the future.

Package Principles

Acyclic Dependency Principle

The basis of this principle states that there are no cycles in the component dependency graph.

As seen through the class diagram there are no visible cycles in the application. Cycles begin to make all packages dependent upon all other packages in the system. This will result in high coupling which is not efficient for the application. As such to ensure this would not occur specific interfaces such as the generator interface were created to avoid this issue.

Common Closure Principle

The common closure principle was demonstrated throughout our program. This was to ensure that our program was made in a way that changes in one package would not affect other components. An example of this would be the booking package has no relation to the on site testing package as they are separate. The pros of making the design this way is the maintainability of the system. This allows it to be more convenient and easier to debug in the future, allowing less cost and time spent refactoring the packaging and program. However, one of the cons would be for future implementation in which by using this principle the packages would become very large. As such changes in one component in the package will result in many changes in the whole package.

Release reuse equivalency

The release reuse equivalency principle was used throughout the program and the class diagram. Classes were assigned into different families depending on their relationship with the package. For example, the classes OnSiteTesting and OnSiteTestingController are associated into the same family in the package OnSiteTesting. Each of the classes shared its own single responsibility, illustrating effectively how the code in each package is not being reused and copied.

References External Libraries

Spring boot initializer
https://start.spring.io/

Javase
https://jar-download.com/artifacts/com.google.zxing/javase/3.3.0/source-code

Json
https://jar-download.com/artifacts/org.json/json/20200518/source-code