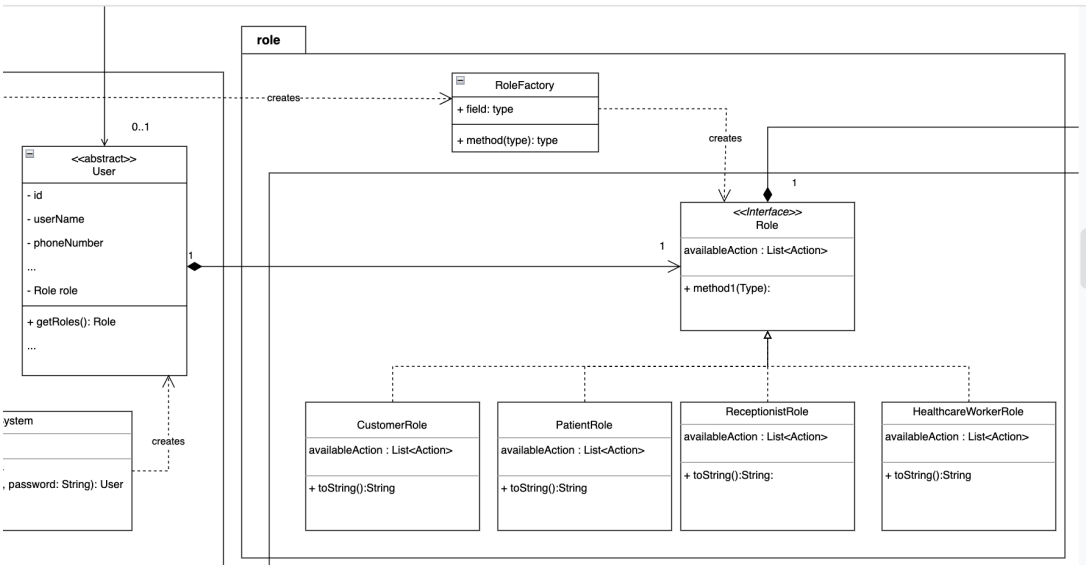
FIT3077 Assignment 2 Design Report

Link for demonstration video. Please access with Monash account: https://drive.google.com/drive/folders/1OvyT31IFwmrRd1F-Qmdu-ilbTYodZcGF?usp=sharing

Design patterns

Creational design pattern



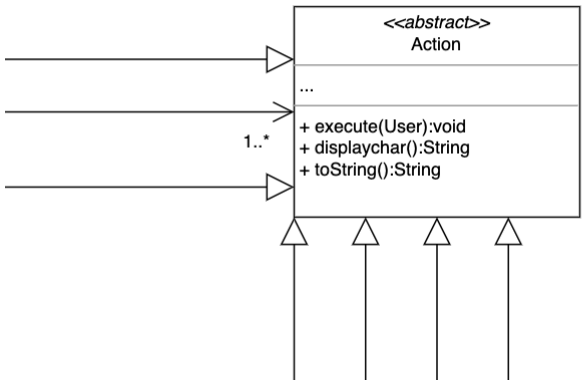
We decided to use the Factory Method pattern for the implementation of creation of user roles. Keeping in mind how User and Roles have high potential to be extended and accept new functionalities, we choose Factory Method pattern because with it, the instantiation of the Role, and eventually the User class can be customised depending on the input data. This input data can originate from the subclass (e.g. of User), for Role. With the decoupling of the instantiation logic and the object, the resulting code can just focus on the logic of the concrete class itself, and any changes in instantiation logic can just be modified in one class. This supports the Single Responsibility Principle, which states that a class should have only one reason to change, as well as Open-Closed Principle, where in this case, the instantiation logic is likely to be Open while the implementation of the class itself is Closed for modification.

We think it is also justified for us to choose Factory Method pattern for User and Roles since in this case, what we have is a parent class (User) that holds the information on how to instantiate its child classes or with roles. This fits the expectation of the usage and the pattern of Factory Method.

We use the Factory Method Pattern more than once throughout our code. We also use it along with the Strategy Pattern, in order to separate the instantiation logic of the strategy interfaces.

Initially, we were also thinking about using the Chain of Responsibility design pattern for implementing user roles and permissions. We wanted to set up a middleware in between a subsystem and the user. This middleware will contain a validation algorithm that will gate-keep Users with invalid Roles.
However, we realised that this pattern is not suitable for these types of Role implementation that are extensible. If we use Chain of Responsibility, then we imagine that the multiple handlers or methods would need to be aware of the rules regarding certain Roles. Assigning the Roles inside the the actor object itself (User) with Factory Method and delegating a list of Actions associated with that Role is much more intuitive instead. It is also easier to manage since we will find this information in one class (the factory method).

Structural design pattern

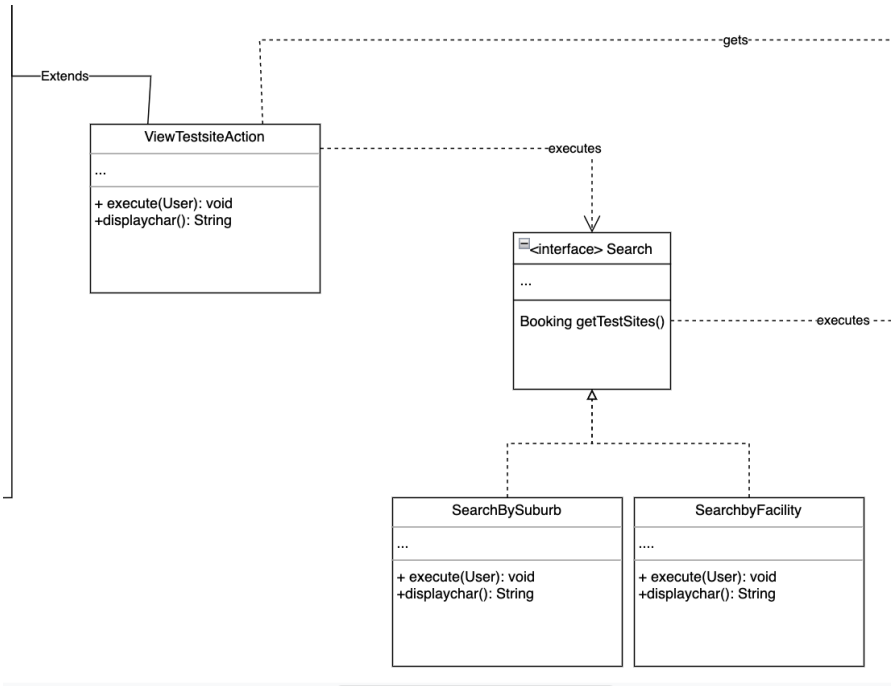


We also incorporate the Facade Design Pattern in our application. For user-friendliness, we create an abstraction of the application and isolate the complexity of the system from the user. By encapsulating the logic and relationships of the subsystems via Action classes, we hide implementation detail and only expose the functionalities that are directly useful for the client.

We have also implemented the Bridge design pattern for the implementation of Roles and Actions of the User. As there are supposedly many roles and actions that a user can perform, meaning it is easily extensible, we think it is better to separate the implementation. With this, the class may also extend in two dimensions (inheritance from User and inheritance or implementation of Roles and Actions).
We create an abstract class Role and Action. This way, User will no longer have direct dependency with the concrete class of Role as well as concrete class of Action, adhering to dependency inversion principle as the high level module does not depend on the low level module but both depend on the abstract class. As implied before, this

also allows us to create new Role or new Action with ease by extending from the High level module,which also adheres to Open-Closed Principle. Implementing the Bridge design pattern also eases the work of unit testing as the system is decoupled into smaller modules.

Behavioural design pattern



In this system, there are potentially many ways to search for a test site. However, if the enclosing class of the implementer of this search functionality needs to be aware of all these algorithms (e.g. if we use basic inheritance), we can imagine that it leads to a code smell as there would be a lot of duplicated code.

In the end, *searchTestSiteByFacility(facility)* and *searchTestSiteBySuburb(suburb)* is going to return the same object that holds the same meaning – it should be a TestSite object which has been successfully queried from the data store.

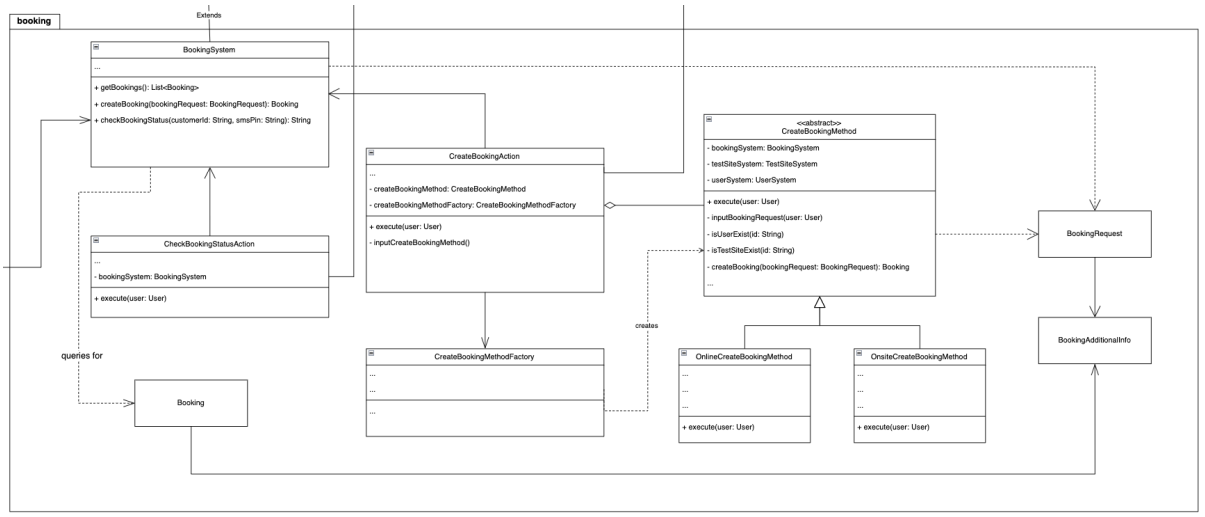
What is different here is just the way the sample methods obtain this result, in other words, only the algorithm is different. That said, what we are trying to achieve here is to separate the parts that are more likely or have varying behaviour.

This goal can be achieved with the Strategy design pattern. We are decoupling the search algorithm from the concrete class ViewTestSiteAction. The base of the search algorithm can be an interface which can be further extended into other different search algorithms.

In the same way, our Booking system also uses the Strategy pattern. The objective of Booking is to create and get a Booking object as a response; however, there are many

ways to get to that result. For example, some types of Booking may prompt for extra information, some require verification. As the variation of the logic is quite irregular, it would be even more difficult to extend when we choose to let it be coupled into the concrete classes.
Packages

The Common-Reuse Principle



The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

Classes that are commonly used together or changed together should be grouped in the same package since the classes are tightly coupled and heavily dependent on each other. For example, the classes related to the Booking functionality, i.e. dependent on the subsystem BookingSystem, like CreateBookingAction and CheckBookingStatusAction, are grouped in the same package as they are heavily dependent on each other and have to do with the domain Booking class. Grouping related classes together increases locality, hence if some classes are actually found to be connascent, the impact of a future refactor will not be as large when they are further apart in different packages. Since the affected entities would still be in the same package, it will ease the test and debugging process when required.