Design Rationale

SOLID Principles

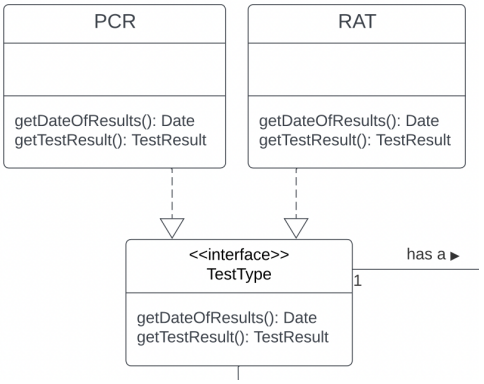
SOLID Principles have been used and justified in Assignment 1. The Single Responsibility Principle, Open/Closed Principle, Liskox Subsitution Principle, Interface Segregation Principle (ISP) and Dependency Inversion Principle (DIP) have been followed during development, and have been extended as explained below.

Creational Design Patterns

Factory Method, Single Responsibility Principle, and OCP

A factory method was used to create objects without specifying the exact class type. Replacing direct object construction calls with calls to a special factory method makes the application more abstract because the caller function does not need to know the exact type of the object returned. The internal components of the application are thus open for extension while being closed for modification (Open-Closed Principle). The 'products' returned by the factory method all implement a common interface, therefore keeping it consistent and avoiding code breakages. It is worth noting that this also saves system resources like memory and processing time since the objects are not being rebuilt each time they are required.

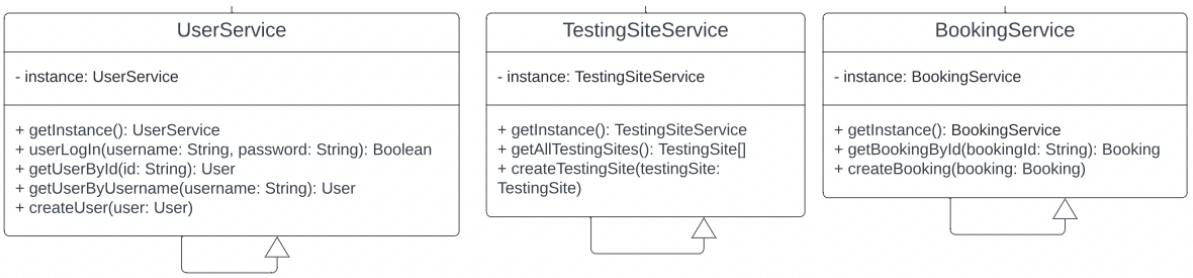
An example of this in our application is the use of the symptomsInterview() function in the User class. This factory method has a TestType return type, which refers to an interface that is implemented by both the PCR and RAT classes. The function can therefore return objects of either type without breaking the code. This also prevents tight coupling between the User class and the PCR and RAT classes. This follows the Single Responsibility Principle since the responsibility of product creation is solely handled by this function. This, along with the added benefit of the Open/Closed principle makes the application highly maintainable without increasing the complexity. However, if the scale of the application increases, the complexity may increase too due to the increased addition of subclasses



Singleton

In our application, we have implemented this for all classes in the Service package, namely UserService, BookingService and TestingSiteService. It is ensured that only one instance of these classes exists across the application, and there is a global access point to the instance and is initialized when its requested for the first time. This may seem to violate the single responsibility principle, however the alternate approach where multiple instances of each service would exist would unnecessarily increase the computational resources being used.

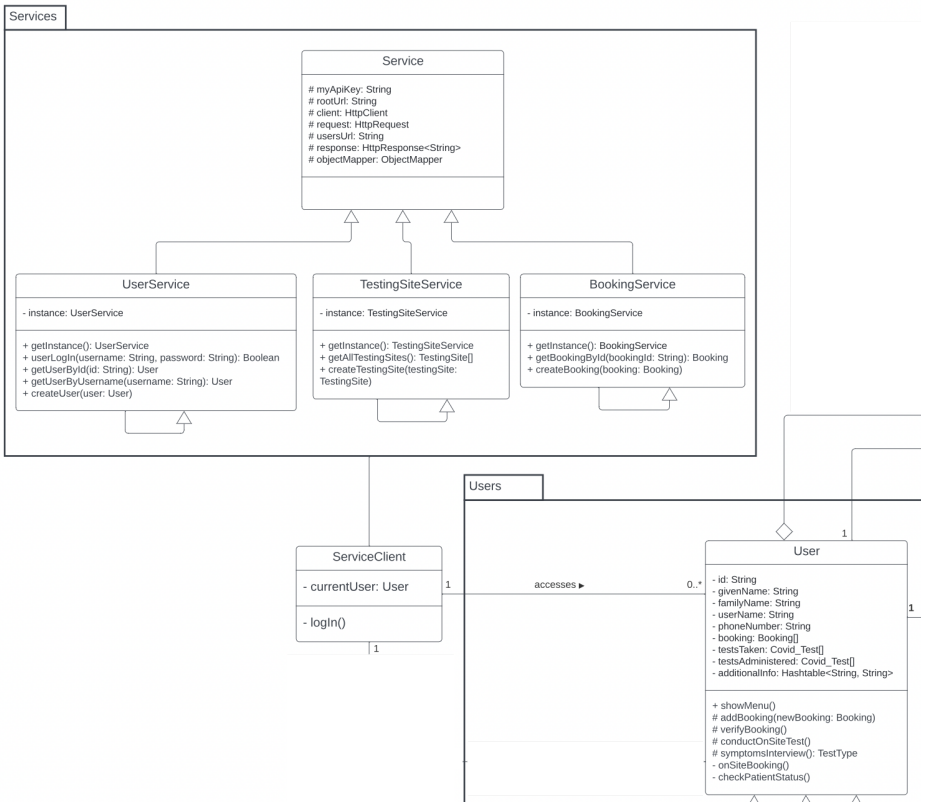
In terms of structure, this has been implemented using a static getInstance() method in the Singleton class, which returns the single instance of the class. This instance is updated every time the object is updated, and a new object is only created if the instance is null.



Structural Design Patterns

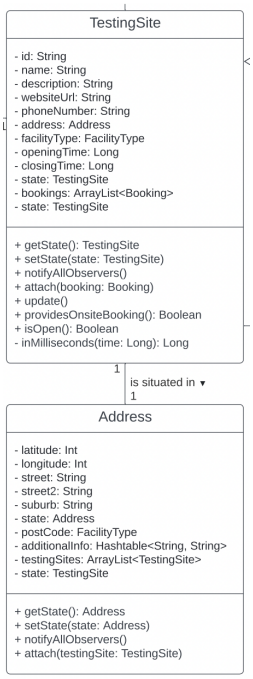
Facade

A facade class has been used in our design to provide a simplified interface to the Services package, to reduce the dependencies on the complex set of classes. The User, TestingSite and Booking classes require various classes from the Services package. Ordinarily, this would require each to declare and initialise an instance of each required service class, have a dependency on each of them and hence be tightly coupled. This would make the application hard to maintaina and extend, and also violate the Open/Closed principle. The use of the Facade class ServiceClient, enables the User, TestingSite and Booking class to solely be dependant on this one entity for any and all service functions. It is a simple interface that is provided to the complex subsystems in the application, and makes it easier to keep track of various moving parts. It is simplified as it only contains the features used by the client class, and avoids exposing unrequired functionalities to classes that do not use it. This decouples the set of classes, by wrapping all the different services together. The use of a Facade class enables the application to follow the CCP and CRP for the Services package.



Behavioural Design Patterns

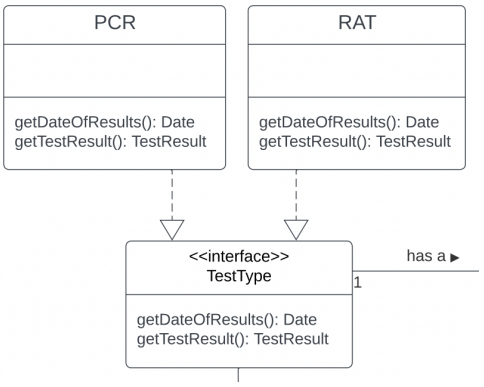
Observer



This pattern defines a one-to-one dependency between objects so when one object changes state, all of its dependencies are notified and updated automatically. In our application, we have used this for objects such as; booking being subscribed to customer so when the customer object is updated, the booking object will be notified, as well as being subscribed to the testing site object. Testing site object is subscribed to address object so that when address is changed, it will updated in testing site where the existing testing sites or new testing sites' attributes are updated. Lastly, Customer is subscribed to all booking objects in its booking list so that when that the test status needs to be changed, the booking will notify the customer object. The main benefit of this is that new subscriber classes can be added without changing the base code, this supports open/close principle. As our application is small scale and performs one functionality at a time, the order in which subscribers are notified does not matter.

Strategy

The Strategy design pattern has been used to make objects of classes in the TestTypes package interchangeable by defining a family of algorithms in the TestType interface. The PCR and RAT classes both implement the functions defined in the interface, each returning their own respective values according to their functionalities. This way, the strategy of returning different values for different test types is delegated to each individual object without exposing the different methods to the client.



Isolating the business logic of the classes from the implementation details allows for single and consistent communication, where the client only needs to be exposed to the overall context, not the specificities involved. The use of this design pattern promotes the Open/Closed principle as it enables the extension of different TestTypes and different strategies if required, while not allowing the client class to modify the specific algorithms. On the downside, in a simple application like this, it may seem to overcomplicate an otherwise simple scenario, however it is a crucial pattern that enables the application to be expanded in terms of its functionality.

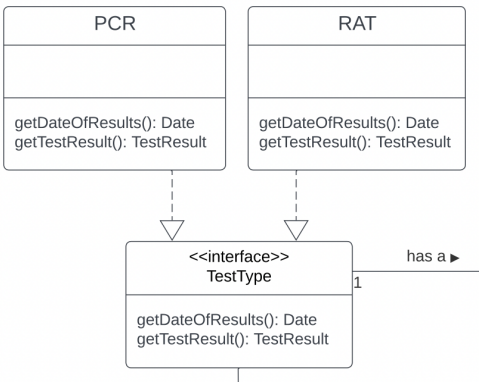
Package Cohesion Principles (followed on from Assignment 1)

The Release Reuse Equivalency Principle (RREP)

This principle states that the unit of reuse is the unit of release. In our design, packages are used in multiple places, such as the TestingSites package, the FacilityTypes package and the TestTypes package. The classes in each package will not have reused code, which is copied from one class and pasted into another, as this would be a waste of resources and cause issues when the design is extended or modified. Instead, release versions would be made whenever the design or implementation needs to be updated. This is why packages are used to group similar classes together, or those who have common attributes and methods or are used for the same purpose - so that they can be released all together as a package.

This principle is essential to increase code reusability and improves the quality of design to make it more suitable for larger and more complex systems as it promotes the concept of dividing the system into components and modules. This makes the system design more scalable for future extension.

The Common Closure Principle (CCP)



The CCP says that classes in a single component should be closed to the same kind of changes. A modification that affects a single class in a package would affect all other classes in the same package, while not having any effect on classes outside this package. It basically outlines that each package component should have a single responsibility, and classes that do not follow this responsibility should not be grouped together within this package.

An example of where this has been applied in our design is in the TestTypes package. A change in any class within the package would affect the other packages too. This component is independent of other components in the design, so there would be no side effects on any other component, as per the principle. This increases the maintainability of the system design because minimal changes would be triggered across the system. This decreases the risk of code breaks in other components, and hence reduces the need for redeployment of larger parts of the system as a result of minor changes in a single component.

Package Coupling Principles (followed on from Assignment 1)

The Acyclic Dependencies Principle (ADP)

The Acyclic dependencies principle states that ‘ the dependency graph of packages or components should have no cycle’ this means that no components should create a cycle because it reduces maintainability and makes it harder to change the component without affecting the other. Hence, in our class diagram, there are no cycles and adheres to the Acyclic dependency principle.

Morning after syndrome is a common phenomena where a functioning component can be made redundant if someone else changes to the component it relied on, which could compound at a later stage as the project gets bigger, and so, taking this occurrence into account, we avoided any such dependencies.

Future Implementation

For future implementation, we have scope to add functionalities such as more features for different type of users, this can be done just by adding more options to the menu and making relevant changes to the classes.

For the next development phase, this codebase can be implemented to MVC architecture, as it is compatible with separation between business logic and presentation layer as well as

the application being primarily used for desktop graphical user and currently our model has one to many relationship between the controller and view.

Assumptions Regarding System Workflow

- We assume that all hospitals and homes are open 24/7, clinics are open 9-6, and GP is open 11-5.
- We assume that waiting time is 15 mins per booking
- We assume that the receptionist and healthcare worker both can administer the test

References

- Design Patterns - Facade Pattern. (2022). Retrieved 27 April 2022, from https://www.tutorialspoint.com/design_pattern/facade_pattern.htm
- Design Patterns - Observer Pattern. (2022). Retrieved 23 April 2022, from https://www.tutorialspoint.com/design_pattern/observer_pattern.htm