Assignment 2: Design Rationale

Introduction

In software engineering, developers work on projects with varying complexites day-by-day. With these projects, many questions could be asked during the development process regarding factors such as its maintainability and extensilibity, which brings the subject of software design to the forefront. Having good software design by using its principles and patterns can go a long way into making the jobs of those involved easier by allowing new features to be more readily added.

Therefore, this document aims to provide the principles and patterns that are adhered to by the team during the development of the web application program. Within the explanations, there would be focus towards the "why" rather than the "what" of the principles and patterns that are followed. Besides that, the advantages and disadvantages as well as an example of where the principles and patterns are applied would be provided.

SOLID and Package Principles

This section would focus on how the packages are designed within the code. Some SOLID principles may be mentioned here if they were not already explained in the section for design patterns. For example, the patterns applied in the program fufill the single-responsibility and open-closed principles as the required functionality of a class are generally abstracted into multiple separate parts, preventing a single class from taking on too many responsibilitie. In addition, any future extensions are made easier without having the need to make many modifications on the existing codebase.

Dependency Inversion Principle

DIP states that a higher level class should not depend on lower level classes. In our code design, the higher level class will invoke the lower level class to perform certain lower level operations. It is the responsibility of the higher level class to direct the lower level class to do something. For example, the customer class will invoke the method in the lower level class to perform tasks such as viewing and searching for testing sites. On the other hand, the "testingSite" class will invoke an even lower level class which is the "html" class to do the GET and POST request. This principle also applies to the user class where the user class will invoke the lower level hml class to do a POST request when creating a new account.

Package Design

When deciding on the package design that would be used in the program, the team chose to make them as cohesive as possible. Therefore, much focus was placed on fulfilling the release reuse equivalency and common reuse principles. In the code, this can be seen as the classes involved in each functionality are placed into their own packages with minimal dependencies to other ones. All the core functionalities involved in the specifications have their own respective packages. To provide some examples:

1. Booking related classes and/or interfaces are placed under the "booking" package.
2. Covid testing related classes and/or interfaces are placed under the "covid-test" package.

By doing this, the reusability of the classes are increased as the package can be easily imported for use in other programs as they are all compiled in a single location. This also makes the layout of the program appear more organized since all the classes are not lumped together; the user can easily see the functionality that each class is involved in through the labelling of the packages.

Design Patterns

This section is dedicated to showing the design patterns that are used within the codebase. The most important thing of note when using these is that they generally provide good extensibility at the expense of increased complexity with the code.

Factory

The factory design principle is used in the code by differentiating between the types of users that exist within the system: the customer, receptionist, adminsterer, and patient. Since each type of user would have different permissions and capabilities within the web application, the team believed that it would be useful to have a creator class that would define the user type that would be instantiated depending on the situation while allowing the code to be extendable without compromising the rest of the code. Rather than having the code revolve around calling a particular group of users via their constructors, using factory classes and methods would reduce coupling within the code and improve its flexibility whenever new user types are added to the system.

For the advantages and disadvantages that arise when using the factory design pattern, they comprise of:

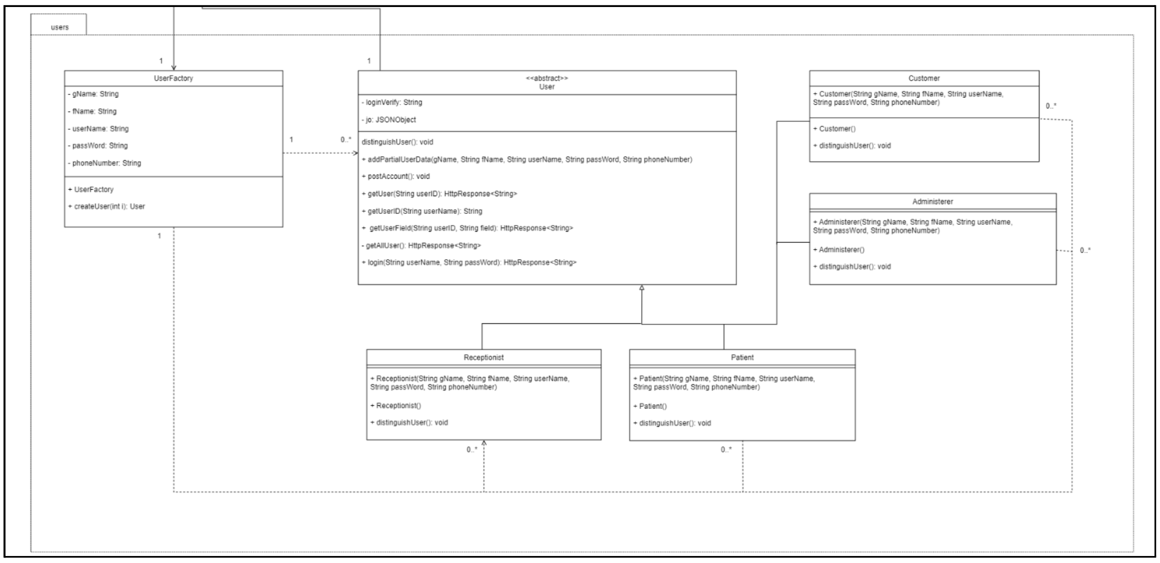
Advantages:

- **Good Accessibility:** Allows the client to instantiate objects without having the need to understand the inner workings of the object classes themselves as they do not need to call them directly (Factory Method, 2022).
- **Saves Resources:** System resources are saved by reusing existing objects rather than rebuilding them each time.
- **Reduce Coupling:** Coupling between the creator and concrete classes are reduced.
- **Adherance to Design Principles:** In particular, the single-responsibility principle and open-closed principle are fulfilled by using factory classes and methods. For the former, we would place the code responsible for object creation into one place. The latter principle is obeyed by allowing the existing code to be extended without having the need to do much modifying on what already exists from the reduced coupling.

Disadvantages:

- **Increased Complexity:** The main problem with using the factory design pattern is that the complexity of the program is increased due to the increased number of subclasses that are used. From a developer's point of view, this may cause the code to be more difficult to read through as there is more code to parse through (Factory Method, 2022)..

Diagram:


users
1
1
UserFactory
- gName: String
- fName: String
- userName: String
- passWord: String
- phoneNumber: String
+ UserFactory
+ createUser(int i): User
1
1
0..*
<<abstract>>
User
- loginVerify: String
- jo: JSONObject
distinguishUser(): void
+ addPartialUserData(gName, String fName, String userName, String passWord, String phoneNumber)
+ postAccount(): void
+ getUser(String userID): HttpResponse<String>
+ getUserID(String userName): String
+ getUserField(String userID, String field): HttpResponse<String>
- getAllUser(): HttpResponse<String>
+ login(String userName, String passWord): HttpResponse<String>
Customer
+ Customer(String gName, String fName, String userName, String passWord, String phoneNumber)
+ Customer()
+ distinguishUser(): void
0..*
Administerer
+ Administerer(String gName, String fName, String userName, String passWord, String phoneNumber)
+ Administerer()
+ distinguishUser(): void
0..*
Receptionist
+ Receptionist(String gName, String fName, String userName, String passWord, String phoneNumber)
+ Receptionist()
+ distinguishUser(): void
0..*
Patient
+ Patient(String gName, String fName, String userName, String passWord, String phoneNumber)
+ Patient()
+ distinguishUser(): void
0..*

In this diagram, some particular things of note are:

- **UserFactory class:** This class is responsible for instantiating the user objects depending on the parameters passed through the createUser method.

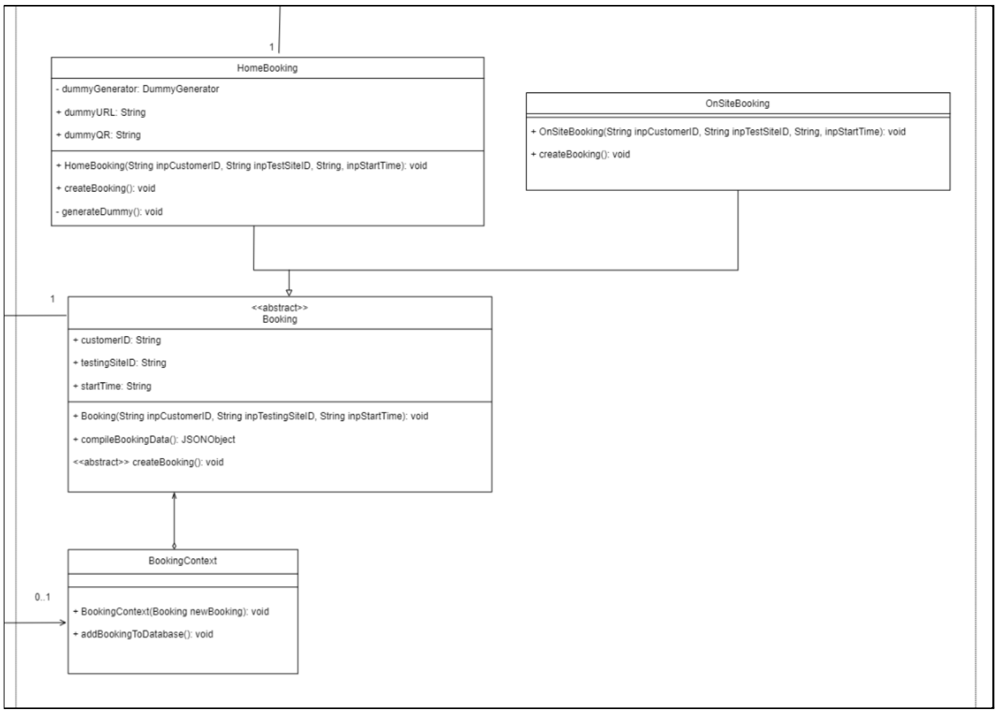
Strategy

The strategy design pattern is used through the booking functionalities within the system. One of the main features of this pattern is the capability of running different variants of an algorithm at runtime depending on which behaviour the user plans to perform (Strategy, 2022). When creating the pseudocode for the bookings, the team felt that the different types of bookings are similar enough to each other that using the strategy design pattern would be suitable. The creation of home bookings follow the same steps as on-site bookings, just with the added requirement of having additional fields set within its corresponding JSON object. By using a context class, the team can easily create either type of booking just by calling a single method by passing the appropriate parameter. This prevents a single class from tackling too many responsibilities while allowing extensions to be performed easily, fulfilling the single-responsibility and open-closed principles. In addition, this also isolates the implementations so it would be more easier to understand by end users.

Aside from these advantages, there are a few disadvantages when using the strategy pattern as well, they are:

- **Increased complexity:** The strategy design pattern may be unsuitable for simpler programs due to the added complexity from the increased number of classes and interfaces (Strategy, 2022).
- **Required awareness of strategies:** When running the code, the client needs to be aware of the different strategies to be able to properly use them (Strategy, 2022).

Diagram


1
HomeBooking
- dummyGenerator: DummyGenerator
+ dummyURL: String
+ dummyQR: String
+ HomeBooking(String inpCustomerID, String inpTestSiteID, String, inpStartTime): void
+ createBooking(): void
- generateDummy(): void
OnSiteBooking
+ OnSiteBooking(String inpCustomerID, String inpTestSiteID, String, inpStartTime): void
+ createBooking(): void
1
<<abstract>>
Booking
+ customerID: String
+ testingSiteID: String
+ startTime: String
+ Booking(String inpCustomerID, String inpTestingSiteID, String inpStartTime): void
+ compileBookingData(): JSONObject
<<abstract>> createBooking(): void
BookingContext
0..1
+ BookingContext(Booking newBooking): void
+ addBookingToDatabase(): void

In this diagram, some particular things of note are:

- **The BookingContext class:** This class would instantiate different types of bookings depending on the type of booking passed through the class's constructor. In the driver, calling the addBookingToDatabase() method would handle the rest.
- **Composition relationship between BookingContext and Booking:** BookingContext requires Booking and its child classes to be able to function. It cannot exist without the other.

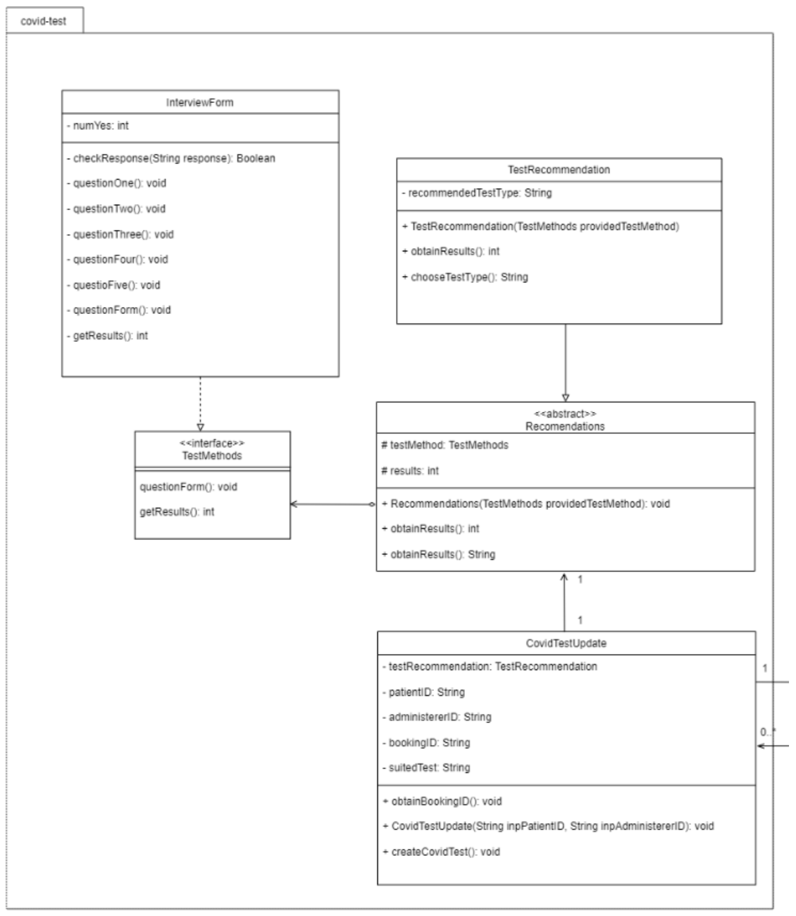
Bridge

In the team's code, the bridge design pattern is most prevalent in the functionalities for creating a covid test, notably through the "Recommendations" and "TestMethod" classes. This design pattern involves splitting a functionality into abstraction and implementation parts (Bridge, 2022). By doing this, any further extensions are made easier as we only need to add to either part rather than the whole. In contrast, if a hierarchical structure of inherited classes are used instead, there would be much more busywork as many more subclasses need to be added for each new functionality. As a result, using the bridge pattern fulfills the single-responsibility and open-closed principles similar to factory and strategy. With the current specifications, even though the classes that utilizes bridge may appear barebones, the team thought that it would be suitable regardless in case different interview formats or recommendation methods are employed in the system in the future.

Despite this, there are some disadvantages to using the bridge design pattern as well, the most notable one is:

Increased complexity: If the bridge design pattern is applied to classes that are highly cohesive, the code would be unnecessarily made more complex (Bridge, 2022).

Diagram:



In this diagram, some particular things of note are:

- **The Recommendations class:** This class acts as the abstraction part of the bridge pattern. It relies on the results from TestMethods and the classes that implement it to provide its recommendations.
- **The TestMethods class:** This class acts as the implementation part of the bridge pattern.

References

Bridge. (2022). Retrieved 27 April 2022, from https://refactoring.guru/design-patterns/bridge

Factory Method. (2022). Retrieved 27 April 2022, from https://refactoring.guru/design-patterns/factory-method

Strategy. (2022). Retrieved 27 April 2022, from https://refactoring.guru/design-patterns/strategy.