FIT3077 - Software Engineering: Architecture and Design

Assignment 1: From Specifications to Design

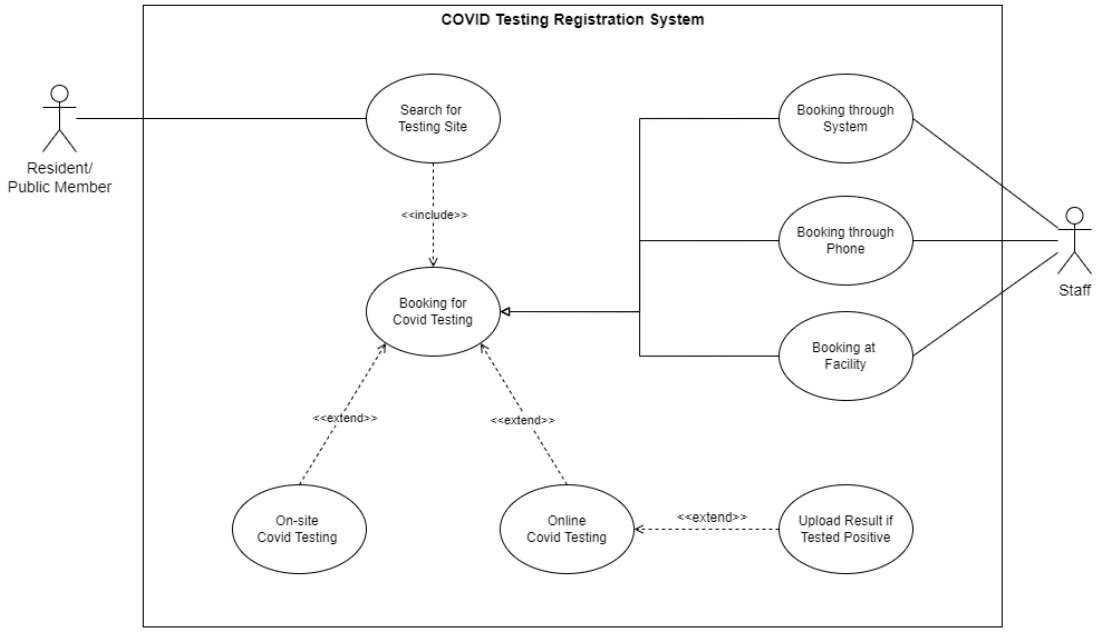
Name: Lee Chang Horng - 31108369

Chan Wai Han - 31555373

Laboratory Class: Lab04

Team: Team15

Task 1: Use Case Diagram



An original PNG photo is attached to the submission file to have a clearer view of the details use case diagram.

Task 2: Use Case Description

3. Browse Testing site near the users' location and book through phone.

UC1: Search for Testing Sites

Actors: Resident/User

Purpose: Browse or search for nearby testing sites.

Description: A resident either enable their phone's location and lets the system automatically read their location or manually search for testing sites using suburb name, postal code, or coordinates (latitudes and longitudes) and the system will suggest nearby testing sites.

Type: Primary Use Case

Precondition: Resident needs to open and login into the registration system or app before searching the testing sites.

Postcondition: Resident successfully chooses a testing site.

FLOW OF EVENTS	
ACTOR ACTIONS	**SYSTEM RESPONSE**
1. This use case begins with a user with a login and opening the app.	
	2. The system will enable the search function for the user.
3. The user will choose their searching method.	
	4. The system will suggest nearby testing sites and for each site, it will show the estimated waiting time for testing and if the testing sites provide walk-in or drive-through facilities. The system will visualize the testing sites in a map view or a list view.
5. The user can choose a testing site's location and book through different methods.	

ALTERNATE FLOWS
2a. The system will prompt users to enable phone location.
2b. The system will allow users to manually search for a particular location.
3a. The user enables the phone's location.
3b. The user manually enters a location into the search bar.
5a. The user books through the phone. <Include> UC2: Booking through Phone
5b. The user books through the system. <Include> UC3: Booking through System
5c. The user books at the facility. <Include> UC4: Booking at Facility

UC2: Booking through Phone

Actors: Resident/User (Primary), Staff (Secondary)

Purpose: Book for covid testing through the phone

Description: A resident that book covid testing by phone will be provided with a PIN code as a text message. Then, the PIN code needs to be shown at the testing facility so that the staff will generate a QR code for the resident to scan by their phone.

Type: Primary Use Case

Precondition: Resident needs to choose an available covid testing site based on their preference.

Postcondition: Resident successfully make a booking to get covid testing at the location they chose.

FLOW OF EVENTS	
ACTOR ACTIONS	**SYSTEM RESPONSE**
1. This use case begins with a user choosing to make a booking through the phone.	
	2. The system will send the user a PIN code as a text message after the test is booked by phone.
3. The user must show the PIN code at the testing facility when they arrived at the site.	
4. The staff checks the PIN code for verification.	
	4. The system generates a QR code.
5. The user scans the QR code using the mobile phone.	
ALTERNATE FLOWS	
1a. The user is able to choose to have onsite covid testing. i. The user can choose to have a PCR test. ii. The user can choose to have a RAT test.	
1b. The user is able to choose to have online covid testing (RAT test).	

UC3: Booking through System

Actors: Resident/User (Primary), Staff (Secondary)

Purpose: Book for covid testing through the system

Description: A resident that books covid testing through the system will be provided with a QR code and the resident has to show the QR code to the facility before getting Covid testing.

Type: Primary Use Case

Precondition: Resident needs to choose an available covid testing site based on their preference.

Postcondition: Resident successfully make a booking to get covid testing at the location they chose.

FLOW OF EVENTS	
ACTOR ACTIONS	**SYSTEM RESPONSE**
1. This use case begins with a user choosing to make a booking through the system.	
	2. The system will generate a QR code for the user after successfully being booked by the user.
3. The user carries the QR code to the booked facility.	
4. The staff will scan the QR code provided by the user as verification.	
	5. The system confirms the registered user with the QR code scanner which is done by the staff.
6. The user is allowed to get a covid test.	
EXPECTION FLOWS	
4a. If the user doesn't have a QR code, the testing facility will deny testing and has to register for the tests at the facility. If the facility can generate QR codes or register users, the user is able to directly register at the facility.	

4b. Alternatively, the user doesn't have a QR code and the facility cannot generate a QR code or register the user.

i. The user has to book through the phone. <include> UC2: Booking through Phone

ii. The resident/public member has to go to another facility that can register users to book for covid testing. <include> UC4: Booking at Facility.

***

For UC3: Booking through System, we assume that if the user doesn’t have a QR code and at the same time the facility cannot generate a QR code or register the user on the spot, the user can only go for another two booking options which are booking through the phone and booking at the facility that allows user to book for covid testing.

***

UC4: Booking at Facility

Actors: Resident/User (Primary), Staff (Secondary)

Purpose: Book for covid testing at the facility

Description: A resident that books covid testing at the facility must ensure the facility has a system to generate QR code and register users. The user will be asked for a phone number and a PIN code will be sent as a text message. The user will also need to scan a QR code generated by the staff.

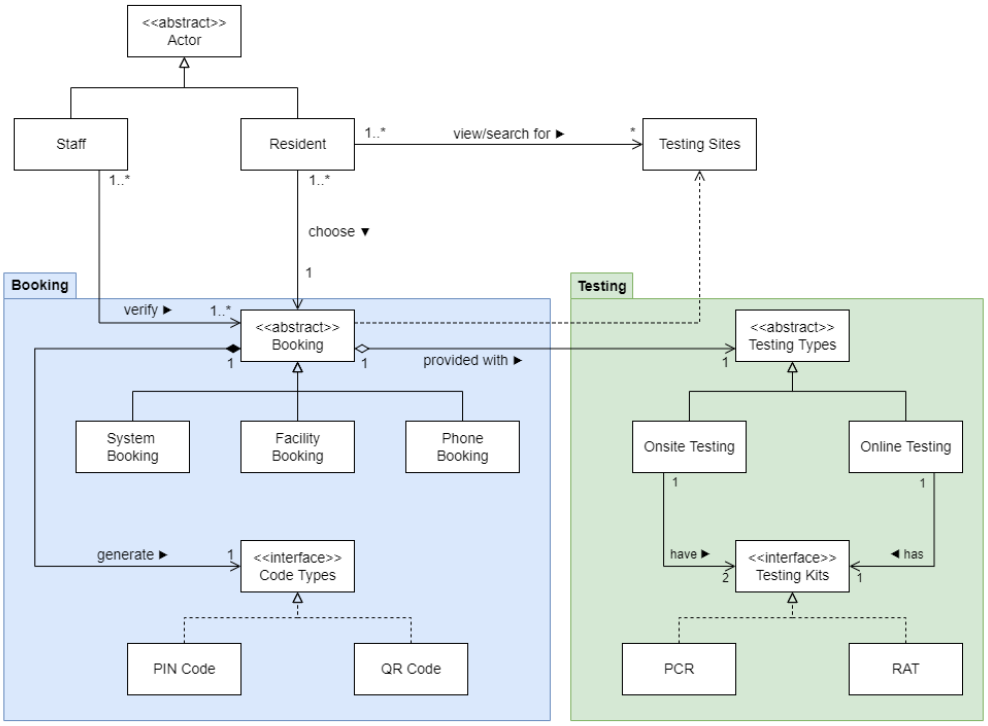
Type: Primary Use Case

Precondition: Resident needs to choose an available covid testing site based on their preference.

Postcondition: Resident successfully make a booking to get covid testing at the location they chose.

FLOW OF EVENTS	
ACTOR ACTIONS	**SYSTEM RESPONSE**
1. This use case begins with a user choosing to make a booking at the facility.	
	2. The system prompts the user to provide a phone number.
3. The user provides a phone number to register for covid testing.	
	4. The system sends a PIN code as a text message to the user.
	5. The system generates a QR code.
5. The user scans the QR code.	
6. The user is able to get a Covid test.	

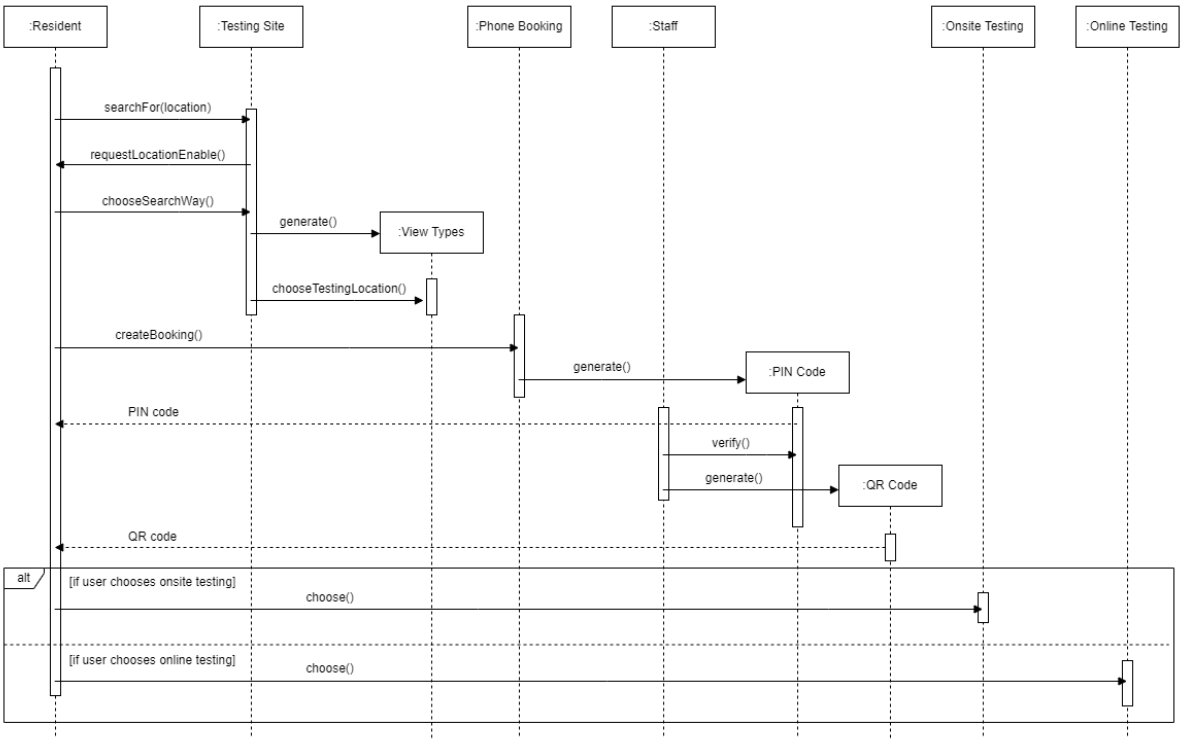
Task 3: Conceptual Class Diagram



An original PNG photo is attached to the submission file to have a clearer view of the details use case diagram.

Task 4: Sequence or Communication Diagram

Sequence diagram for Task 2



An original PNG photo is attached to the submission file to have a clearer view of the details use case diagram.

Task 5: Design Report

In this design report, we will be discussing the design of our conceptual diagram in Task 3 mainly on relationships between classes, SOLID principles and package principles.

Relationship between Classes

Aggregation

The relationship that we will be looking into is Aggregation which appears between Booking and Testing Types classes. The reason that Booking and Testing Types classes will be connected with the Aggregation relationship is that the Testing Types object will be created when the user has booked for covid testing and when the Booking object is deleted in some situations, the Testing Types object will still exist for users because the system will be recognised that the user has booked for covid testing and waiting for tested with either onsite or online testing.

Composition

Besides, the relationship between Booking and Code Types classes is Composition. This is because a Code Types object will only be created when users have confirmed a book for covid testing while after booking the system will generate either a QR code or PIN code for the users. If the Booking object has been deleted, that's mean the Code Types will also be deleted because both of these classes are linked together.

Generalisation

Moreover, the relationship we will be discussing is Generalisation which contains Inheritance and Realisation. For the Inheritance relationship, a parent class is inherited by the child class which can exceed the method and variable from the superclass and this exists in the Booking abstract class where System Booking, Facility Booking and Phone Booking inherit from it because three of them have some similar methods to implement. Besides, there is a Realisation relationship between the Testing Types interface and PCR and RAT. The relationship of Realisation exists between them is due to both PCR and RAT being bounded in the Testing Types interface which will have several same method names but different implementations for the method.

SOLID Principles

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is defined as a class having only one reason to change. In other words, a class should only be responsible for a single part of a software's functionality, with it being encapsulated by the class. In our diagram, we managed to implement this principle by creating separate classes for each of the software's functions. The different functions of searching for testing sites, booking for testing sites, and the testing types have their own classes, with them being packaged up. This helps to reduce complexity and achieve loose coupling between the classes, where unrelated codes are separated as much as possible. By separating each functionality into its own class, we are able to prevent unnecessary changes, which may break other parts of the class which we did not intend to change. Besides that, through the SRP implementation, we are able to keep our classes are cohesive, whereby related codes are compiled together in a single place. If there are responsibilities that need to be changed, we can easily do so without any worry as all the components that are related to the responsibility would be encapsulated within its particular class. Not only that, excess features or unrelated responsibilities would not be of concern when editing the particular class. Throughout the implementation of the SRP, it also enables the system to be easier to maintain and extend, which is another principle that will be discussed in detail below.

Open/Closed Principle (OCP)

The Open/Closed Principle is defined as software entities such as classes, modules, and functions that should be open for extension but closed for modification. In other words, a class should be able to be extended, but unable to be modified. We kept this in mind when creating our diagrams as we understood that during the software development process, several and frequent changes may occur due to the addition or deletion of features. Therefore, the easiest way to implement this principle was through the usage of interfaces. The Code Types class is an interface that will be implemented by the QR code and PIN code classes. We decided to use interfaces for the Code Types due to the potential of the addition of new features into the program. The Booking class interacts with the Code Types class. So if there is a new Code Type, the Booking class would not need to modify its current code as it refers to Code Type Interface instead of specific code type classes. Therefore, through

interfaces, we would ensure that the code modules are able to support the new functionalities and would be able to add the new methods easily. For example, if new types of codes other than QR codes and PIN codes were to be implemented, we would be able to add those features into our software with ease. On the other hand, interfaces ensure modifications do not happen by making it hard to add features. In other words, modifications are prevented due to the interface's nature of disallowing the instantiation of variables and the creation of objects. Besides that, interfaces do not contain concrete methods with implementation. This means that methods in interfaces technically can never be modified.

Liskov Substitution Principe (LSP)

The Liskov Substitution Principle is as defined as when we are extending a class, we should be able to pass objects of the subclass in place of objects of the parent class without breaking the client class. From the definition above, all the subclasses should be able to remain compatible with the classes that are able to work with the behaviour of the superclass. In our implementation of the conceptual diagram, we apply this concept by having an interface of Testing Kits where different types of testing can implement this interface, then from the classes of Onsite Testing and Online Testing, there is an association relationship between them where different Testing Types can have different Testing Kits. So, whenever there is any new class that implements the Testing Kits interface, both the Onsite Testing and Online Testing are able to access the subclass without changing any reference from the original code. Here, we provide some more details examples to illustrate this principle. For example, in the Onsite Testing class, there is a method called contains() where a variable is an object of Testing Kits, instead of putting this → `contains(PCT p)` or `contains(RAT r)`, we should put the object as → `contains(Testing Kits t)`, so if we pass an object that implements this interface, the client code will still work fine without changing any reference.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states that high-level classes should not depend on low-level classes as both should depend on abstractions. On the other hand, abstractions should not depend on details but details should depend on abstractions. We implemented this principle through the Code Types interface. Firstly, we identified the Code Types interface as the low-level class, as it implements basic operations, whereby, in this case, it generates the PIN code and QR code. Then, we identified the Booking class as the high-level class as it contains complex business logic that would call a low-level class to do a job. With that in mind, the DIP states that when the high-level classes interact with the low-level classes, the design should have an interface between them. This is the case for when the high-level class - Booking needs to generate the QR code and PIN code, the low-level class - Code Types is between them. Thus, moving forward, we would be able to make the high-level class - Booking dependent on the interface, rather than concrete low-level classes.

Package Principles

The Common Closure Principle (CCP) and The Common Reuse Principle (CRP)

In our conceptual diagram implementation, we have applied both the Common Closure Principle (CCP) and Common Reuse Principle (CRP) where all the classes in a component should be closed together against the same kind of changes and all the classes in a component are reused together. By applying these package principles, we have grouped all the classes of Booking and Code Types as in a package because all of them is interrelated with a tight relationship where when the Booking class is generated the Code Type classes will also be generated straight away. Therefore, by grouping them together, we easily reuse and change some components within them with less effort. Besides, packing them into a package can ensure cohesion between the classes.

The Acyclic Dependencies Principles (ADP)

The Acyclic Dependencies Principles mentioned that it is not allowed to have cycled in the component dependency package. From our conceptual diagram, we only contain two main packages which are Booking and Testing and only the Booking package is pointing to the Testing package without any cycling in between.

The Stable Abstraction Principle (SAP)

The Stable Abstraction Principle states that a component should be as abstract as it is stable. This principle was achieved through the application of the OCP principle and by adhering to the DIP principle in our diagram. As seen in our diagram, the Booking package has quite a few incoming dependencies, which means that it is a stable package, as it is responsible for others. Not only that, but the Testing package is also independent, as it does not depend on anything, which means that no other packages can make it change. In other words, the Booking and Testing packages are the most abstract as they are the most dependent upon. In a nutshell, the SAP is essentially just a restatement of the OCP and DIP but for packages.