Task 2: Design Rationale

Singleton Pattern

We have applied the singleton pattern for our ControlPanel class and all our UI page classes. For these classes, they will have a private constructor and they each have a method which creates an object instance of the respective class itself. The class provides a method to access its single object, which may be accessed without having to instantiate the class's object. The methods are declared as static which then get its static instance. Then whichever class that calls the method to create the instance can then access the components of that class. However, the singleton pattern stray from the principle of single responsibility. A singleton class is responsible for creating an instance of itself as well as other business functions.

Liskov Substitution Principle

Implementing this principle allowed us to replace objects of a superclass with objects of a subclass. For example, we have a **User** abstract class with multiple subclasses that extends it such as **Customer, Receptionist** and **Administerer** class. Each of the subclasses behave in the same way as the superclass whereby the methods of the subclasses are exactly the same as the ones from the superclass since no overriding of methods is done for any of the subclasses as only get methods are implemented in the code. Therefore, the return value of a subclass method follows the same set of rules as the return value of a superclass method. This is important when we want to expand a new user role that expands the User abstract class as it reduces repetitive code and can be easier to maintain.

Open/Closed Principle

This principle is also implemented when we design abstract parent classes with child subclasses. Therefore, we will be able to add new features without having to change the original code. For example, one of the abstract classes that we have is the **AdditionalInfo** class and it has an abstract toString method which returns a string of json format. Since we have multiple classes that have an additional info section and these additional info sections will be different for each class since they will be storing different information, we decided to create additional info classes for each such as BookingAdditionalInfo and UserAdditionalInfo class. The toString method in all these subclasses will be different and instead of creating multiple different toString methods and changing them it is better to have multiple subclasses that inherits the Additional Info superclass and override the toString method. Resilience to change is important for our system when more api models are added for different classes.

Interface Segregation Principle

In our system, we have an interface class called AdditionalInfo but with no methods and has multiple subclasses that implement it. Since a class can extend only one class but implement any number of interfaces, we have decided to use interface as when an interface is used as an instance validator, the solution is not limited in terms of inheritance. We have also thought about future implementations where the AdditionalInfo class will have many methods and thus can be split into multiple interfaces which can be implemented by all the other additional info classes such as BookingAdditionalinfo and UserAdditionalInfo class. Therefore, we can easily apply the interface segregation principle.

Common Closure Principle

By grouping our classes in packages, this allows us to ensure that CCP is upheld. As this principle states that the classes in a package should be closed together against the same kinds of changes, and that the change affecting the package affects all the classes in the package and no other packages. For example, in our system, we have added an additional info package. This package stores the interface AdditionalInfo as well as the classes that implemented it. In the future, perhaps in assignment 3, should there be new features we are required to implement for additional info, which is considered as a change to the package, such as modifying the interface for the new features, we will be able to make changes and affect all additional info classes of all the resource types (e.g testing site, user) in the package, and no other packages.

Acyclic Dependencies Principle

We have also applied ADP, by putting classes in packages in such a way where package dependencies do not form a cycle. For example, the package additional info depends on the package entity, and entity depends on the package covidsystem. But, covidsystem does not depend on additionalinfo, which means that there is no directed acyclic graph.

Single Responsibility Principle

We have implemented SRP by ensuring that each class only has one responsibility over a single section of the program's functionality, and it should encapsulate that section. As it is one of the most important principles to uphold, we have ensured that every class in our system only has a single job/task to perform. For example, for each page that we display in our GUI, it is done using a JPanel. A JPanel that handles the search testing site page (SearchTestingSitePage) will only contain methods that build the display for the page and methods that provide interaction with the user (e.g listeners). Another example is the class that handles navigation of pages. ControlPanel class only has one task, to be able to display pages properly and correctly. It does not need to care what is

in the pages, as long as it does its job by displaying it. In conclusion, SRP allows maintainability and reduces likelihood of bugs to occur.

References

Janssen. (2018). SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples. Stackify. Retrieved from https://stackify.com/solid-design-liskov-substitution-principle/

Janssen. (2018). SOLID Design Principles Explained: The Open/Closed Principle with Code Examples. Stackify. Retrieved from https://stackify.com/solid-design-open-closed-principle/

tutorialspoint.(n.d.).Design Pattern-Singleton Pattern. tutorials point simply easy learning. Retrieved from https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Edpresso Team. (n.d.). What are the SOLID principles in Java. Retrieved from https://www.educative.io/edpresso/what-are-the-solid-principles-in-java