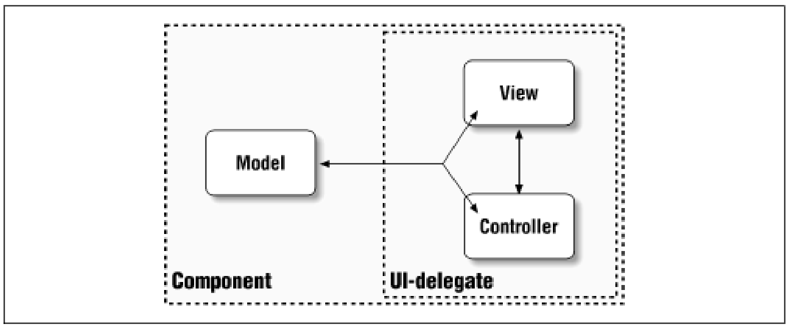
Team: MA_Lab04Team3
Task 2 - Design Rationale

Architectural Pattern

Our system is coded using Java and Swing (GUI widget toolkit for Java), and is done mostly in Apache NetBeans IDE and Visual Studio Code. We have implemented the Model-View-Controller architecture (MVC) for our system. Swing by nature, uses the MVC architecture as the fundamental design behind its components. As our system is a desktop based application, there are limitations when it comes to the implementation of the MVC architecture. Due to the nature of Swing and Netbeans, the MVC architecture we have implemented is actually a simplified/different variant of it, known as the model-delegate. The limitation is that we do not split the view and controller parts, which could potentially violate the single responsibility principle. The image below (obtained from a website, listed in References) shows that the view and controller are combined into a UI-delegate object.



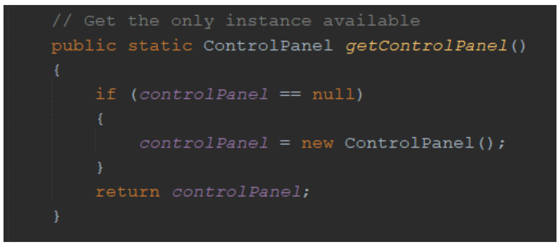
The reason why the view and controller are combined together, is due to the fact that the view and controller parts require a tight coupling. The benefit is that by combining them together, it reduces the difficulty in writing code for the view and controller parts. For instance, in one of our JPanel classes (a view), called SearchBookingPage, it can be seen that the view components exist in the same class as the controllers, whereby the view has its listeners created and its logic methods implemented in the class. We can technically say that the view and controller parts are weaved together. Another benefit is that it reduces the difficulty in debugging code, since the view and controller are located in the same class. As for our model, we believe that the classes that handle the web service API as well as the resource type classes (e.g. testing site, user) are our models, as they retrieve data from the web service and store them as Java objects.

Design Patterns

In assignment 3, we have added 2 new design patterns. The abstract factory pattern and the facade pattern. The singleton pattern we have implemented since assignment 2 still remains usable for assignment 3.

1. Singleton

For our ControlPanel class, we have applied the singleton pattern where we have created a method call getControlPanel() which is responsible for creating a single instance of the class itself and the method is also declared as static so that it can provide global access to all the other classes within the system. We have set our constructor to be private as well so that no more than one object is created & used. The part where the classes will be calling the ControlPanel instance are all the page classes in the com.mycompany.pages package. Some of the examples are the UserProfilePage where it declares a variable of type ControlPanel and assigns the variable with an instance by calling the method ControlPanel.getControlPanel(). We used this pattern because we wanted to provide a global access point for other classes to access it easily as many of our classes will be needing the class to access certain methods to navigate between pages. The image below shows where the method to create a single instance of our Control Panel is at.



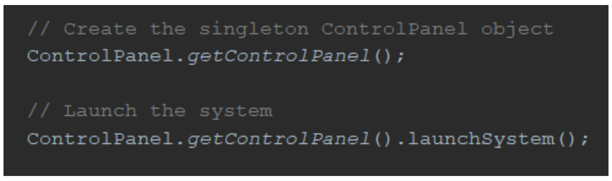
```
// Get the only instance available
public static ControlPanel getControlPanel()
{
    if (controlPanel == null)
    {
        controlPanel = new ControlPanel();
    }
    return controlPanel;
}
```

2. Abstract Factory

In our system, we have applied the abstract factory pattern to our API package classes. All the classes in the com.mycompany.api are basically the structure for this pattern. Our APIFactory class serves as the interface for the Abstract Factory class. The BookingFactory, TestingSiteFactory, UserFactory and CovidTestFactory classes serve as the Concrete Factory classes. The Get, Delete, Patch, Post classes serve as the Abstract Product classes. The BookingGet, BookingPost, BookingPatch etc. serve as the Concrete Product classes. We decided to apply this pattern to our API package classes because we realised that the web service and its methods coincidentally match the requirements for this pattern. The client code will then call the creation methods of these factory objects which will then correspond to a single product variant which has the methods to retrieve data directly from the web service. We believe that by implementing this pattern, the code becomes much cleaner and easier to track, as compared to clumping all of the methods into one single class which is hard to read and possibly debug.

3. **Facade**

   The facade pattern we have implemented into our system is located in the CovidSystemLauncher class. The CovidSystemLauncher class acts as the client to our system, as it contains the main method and runs the ControlPanel singleton object. Our system contains multiple subsystems (essentially, each page is considered as a subsystem). The ControlPanel singleton object navigates pages, which means that it handles the complex subsystems and how they interact with each other. The CovidSystemLauncher, which is the client, does not need to know how the page subsystems are navigated internally, it just needs to run and launch the system, that is all. So, by implementing this pattern, all the client has to do is to run two lines of code, as shown in the image below.



```
// Create the singleton ControlPanel object
ControlPanel.getControlPanel();

// Launch the system
ControlPanel.getControlPanel().launchSystem();
```

Design Principles

1. **Single Responsibility Principle**

   We implemented SRP by guaranteeing that each class is only in charge of a single aspect of the program's functioning. We have guaranteed that every class in our system has only one job/task to fulfil, as this is one of the most crucial concepts to keep. For example, the new class that we added for our assignment 3 other than UI is the PreviousBooking class where its only responsibility is to store details about previous bookings such as testingSite Id, timestamp, testingSite name and start time. We made sure that we always follow the SRP because SRP makes it easier to maintain and minimise the occurrence of bugs.

2. **Open/Closed Principle**

   We have initially already implemented this principle for certain classes in our assignment 2 where we design abstract classes for User and AdditionalInfo so that it can be extended if there are any new users or api introduced in assignment 3. However, for assignment 3 we have included some new abstract classes when we designed the abstract factory pattern. The abstract classes are the Product classes which are the Get, Patch, Post and Delete classes. In a way we could say that the open/closed principle is applied here as when new api methods are introduced it can be easily extended from the Product classes, so we can add new functionality without having to change the existing code.

3. **Interface Segregation Principle**
   In our system, we have implemented this principle already for our assignment 2 which is creating an interface for our AdditionalInfo class and then have multiple classes such as UserAdditionalInfo and BookingAdditionalInfo that implements it. This design has also been brought forward to assignment 3.

Package Principles

1. **Acyclic Dependencies Principle**
   We've also used ADP by arranging classes in packages in a way that prevents package dependencies from forming a loop. For example, the package additional info depends on the package entity, the package entity and package pages depend on package api but package api does not depend on any package at all. Therefore, no cyclic dependencies are occurring in our system.

2. **Common Closure Principle**
   We can verify that CCP is followed by organising our classes into packages. As stated in this concept, the classes in a package should be closed together against the same kinds of modifications, and any change affecting the package should affect all of the classes in the package and no other packages. In our system, for example, we've included a Additionalnfo package. This package contains the AdditionalInfo interface and the classes that implement it. Should we be required to implement new features for additional info in the future, which is considered 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 resource types (e.g. testing site, user) in the package, and no other packages.

Refactoring Techniques

1. **Extract Function/Method**
   This technique is used when there exists a code fragment that can be grouped together. In our ModifyBookingPage, we have a method called modifyBooking, which is used to create a start time and patch a booking with data. There are a decent amount of lines that are involved in creating

the start time, and since they serve the same purpose/goal of creating the start time, they are put together in a method. Also, this method is called in 2 locations inside the confirmChangesButtonActionPerformed method. So by storing the code fragment in a method, this reduces the duplication of code in the method that uses the code fragment. It improves readability as well. This is applied during the extension of new features in our system.

2. **Extract Variable**
   This technique is used when there exists expressions of code that are hard to read/understand. So, the expressions are separated into different variables, to greatly improve the readability. For instance, in the setDetails method of SearchTestingSitePage and SearchBookingPage, some methods might have overly long chain method calls such as object.getThis().getThat().getThose(). This can be solved by separating the method calls into variables, and using those variables to continue the method calls. This is applied during the extension of new features in our system.

3. **Extract Class**
   This technique is used when one class does the work of two. We have applied this technique in resource type classes such as TestingSite. The TestingSite class contains another object called the TestingSiteAdditionalInfo. Instead of returning the methods of the TestingSiteAdditionalInfo in the TestingSite class, we return the TestingSiteAdditionalInfo object itself. Then, to call the methods, we will just perform a call such as TestingSiteAdditionalInfo.method().

References

stackoverflow. (2011). The MVC pattern and Swing. Stackoverflow. Retrieved from
https://stackoverflow.com/questions/5217611/the-mvc-pattern-and-swing
Oreilly. (2022). The-Model-View-Controller Architecture. Retrieved from
https://www.oreilly.com/library/view/java-swing/156592455X/ch01s04.html
JournalDev. (n.d.). Facade Design Pattern in Java. Retrieved from
https://www.journaldev.com/1557/facade-design-pattern-in-java
tutorialspoint.(n.d.).Design Abstract-Factory Pattern. tutorials point simply easy learning. Retrieved from
https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
Janssen. (2018). SOLID Design Principles Explained: The Open/Closed Principle with Code Examples. Stackify. Retrieved from
https://stackify.com/solid-design-open-closed-principle/