Design Rationale

Introduction

As this assignment was mostly about improving the system developed in the previous assignment, we looked into various ways to incorporate new design and architectural patterns into our system. As most classes in our program were already designed based on the design principles we have learnt, we found that most of them were convenient in terms of implementing additional features to them. The new requirements that were given in the assignment specifications mostly just required us to add additional functions into existing classes, and there were no major changes in the previous code required to support these new features. In addition to that, some refactoring techniques were applied to the code to make it cleaner and more functional for this assignment. The refactoring techniques, as well as the design and architectural patterns will be explained in addition to the design principles and patterns that were incorporated in the previous assignment.

SOLID Principles

For the SOLID principles, all the principles from Assignment 2 were maintained and will be briefly explained in the following section.

1. Single Responsibility Principle

While we do have the Singleton design pattern implemented in our program, the other classes are still in a state where each of them only has a single responsibility. When this is maintained within the system, it allows for changes to be made to a single class without other classes needing to change alongside it as well, which translates to a system which has a good amount of maintainability. This was maintained in the current system as we had a clear view on what new responsibilities were required for the new extensions, so we were able to split these responsibilities to separate classes.

2. Open/Closed Principle

When designing our system, we want it to have the ability to be improved upon with greater convenience, especially for sections where there is potential for new features to be implemented. One such example is the addition of the bookingOriginator interface which is being extended by the booking class. While this interface is currently only being extended by the booking class, it can certainly be extended by other classes in our system, should the implementation be required. With that, the overall maintainability of the system is improved.

3. Dependency Inversion Principle

In the current system, certain classes like the bookingAPICollection class are low level, and the classes which interact with it are high level. High level classes should not depend on low level classes, but both should instead depend on abstractions. This led to the implementation of the bookingCollection interface, which lets the bookingAPIcollection class depend on abstraction, and ultimately follow the Dependency Inversion Principle.

Package Principles

Whilst there were changes to the package structure, we maintained the package principles from Assignment 2 even after the new extensions were added.

1. **Common Closure Principle and Common Reuse Principle**

In our diagram, we ensure that each package created within the system contains classes which have similar responsibilities. This is done so that each package is able to behave the way it is intended to when the classes within it work together. To provide an example, the testing site package contains the following classes:

- testingSite: A class representing the overall structure of the testing site
- testingSiteAPI: The class responsible for retrieving data related to testing sites from the API.
- testingSiteAPICollection: The class responsible for storing testing site data locally, as well as providing methods to retrieve data from the locally stored data.

When these classes work together, they carry out the overall functionality of handling the testing sites in the system. For this assignment, we reorganised our file structure to support the MVC software architecture. With there being a clear line for which packages represented which components in the MVC, we just needed to sub-package each of these packages into the corresponding components of the MVC architecture.

2. **Acyclic Dependencies Principle**

When designing the class diagram, it is important to ensure that there are no dependency cycles in the diagram. When a cycle is present, it would mean that one change made in a package would lead to a problem occurring in another package and vice versa. By adhering to the Acyclic Dependencies Principle and ensuring that no cycles are present, this allows our code to be more maintainable. After adding the extensions for this assignment, there was not much trouble in maintaining this principle for our packages

Design Patterns

The design patterns that were incorporated in the previous assignment are pretty much the same for this assignment. In addition to these design patterns, we managed to include some new ones, as well as some additional implementations which follow the previous design patterns that were used. These new additions will be explained below.

1. **Observer**

In addition to the observer pattern classes from the previous assignment that were implemented for when the user logs in, we also included a new listener and manager for whenever a booking is deleted, added, or modified in a system. This was used in the bookingsThread class as several panels should be notified whenever there is change to the bookings on the database. The implementation also involves multi-threading, the thread will detect for changes, and use the manager class to subscribe and notify each listener about the change. Once a change is detected, the listeners will be notified and update their tables containing the bookings accordingly.

2. **Memento**

As the system requires the functionality for the user to be able to revert their bookings to an older version of when they modified it previously, we felt that the memento design pattern would be most suitable in simulating this functionality. By implementing the memento design pattern, the system can capture save states for whenever a user decides to modify their booking. Then, that save state is stored within the system, and can be retrieved again later when a user wants to revert their booking to an older version.

3. **Singleton, Adapters and Facade**

These design patterns were maintained from our previous assignment. For this assignment, we mostly extended classes involved in these patterns, especially those that relate to bookings. The previous classes were not changed and were only extended to add new functionalities to support modifying and deleting bookings. With that in mind, this shows how the addition of these design patterns from our previous assignment supports the extendability of our program.

Architectural Patterns

Our desktop application involves a lot of user interaction with an interface in order to interact with the system as a whole. With that in mind, we decided to use a slight variant of Model View Controller (MVC) as our architectural pattern. The Model View Controller is beneficial in a sense where components are independent from one another when we want to make improvements to the system. To give an example, if a new panel needs to be implemented in the user interface, we only need to focus on the view component of the system. With this being the case, it makes the application more convenient to develop, as we can theoretically have one individual work on the code for one component, while the other works on code for the other component, without both components conflicting with one another.

As mentioned earlier, our implementation of the Model View Controller slightly deviates from the original architectural pattern. To elaborate on this, we developed our interfaces using Swing, which is a GUI widget toolkit in the Java language. Rather than using the conventional Model View Controller pattern, this toolkit combines both the View and Controller into a single component, which lets the user interact with the View and see the changes made to it directly.

Refactoring Techniques

Whilst developing the code for our system, we applied certain refactoring techniques to help the code adhere to various design principles, as well as to improve its readability and expose various bugs that are hidden within the code. Below are the explanations of the few refactoring techniques:

1. Hide Delegate and Extract Class

These two refactoring techniques were applied whilst developing the code, so that we could adhere to the Single Responsibility Principle. An example of Hide Delegate can be found in the Booking class, as it holds information about the testing site and the user tied to that booking. By using this refactoring technique, we do not need to directly acquire information about the user and the testing site from their respective classes, and instead obtain it from the booking class. On the other hand, Extract Class involves splitting a class into multiple classes so that it does not bear multiple responsibilities. The biggest example of this is the Memento design pattern implemented in our system, which involves splitting the originator, caretaker and memento components into multiple classes.

2. Extract Function/Method

After we finished developing each functionality within the code, we noticed certain lines of code were repetitive. Therefore we decided to create a singular function which is dedicated to the functionality of those lines of code. This helped to reduce redundancy within the code, whilst also improving readability and efficiency. An example can be found in the profilePanel class

3. Extract Variable

Some expressions in the code were quite complicated, which led us to declare some variables to represent portions of the expression. Doing this helped to improve the understandability of those expressions, ultimately making the code cleaner. One example of such a function can be found in the bookingCaretaker class, which is a function named getMementosJSON(), which is responsible for creating the JSON string to be passed into the API. The functionality of this code was found in multiple sections, and was therefore converted into one function.

References

Design patterns tutorials. JournalDev. (n.d.). Retrieved May 20, 2022, from https://www.journaldev.com/java/design-patterns

Design patterns. Refactoring.Guru. (n.d.). Retrieved May 20, 2022, from https://refactoring.guru/design-patterns

Svirca, Z. (2020, May 30). *Everything you need to know about MVC architecture.* Medium. Retrieved May 27, 2022, from https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1

The Model-View-Controller Architecture. O'Reilly. (n.d.). Retrieved May 25 2022, from https://www.oreilly.com/library/view/java-swing/156592455X/ch01s04.html