Assignment 3: Design Rationale

Introduction

This document aims to provide the reader with the new design principles and techniques, if any, that are introduced while implementing the added functionalities in Assignment 3. The software architecture pattern and refactoring tehcniques that are employed by the team would also be discussed.

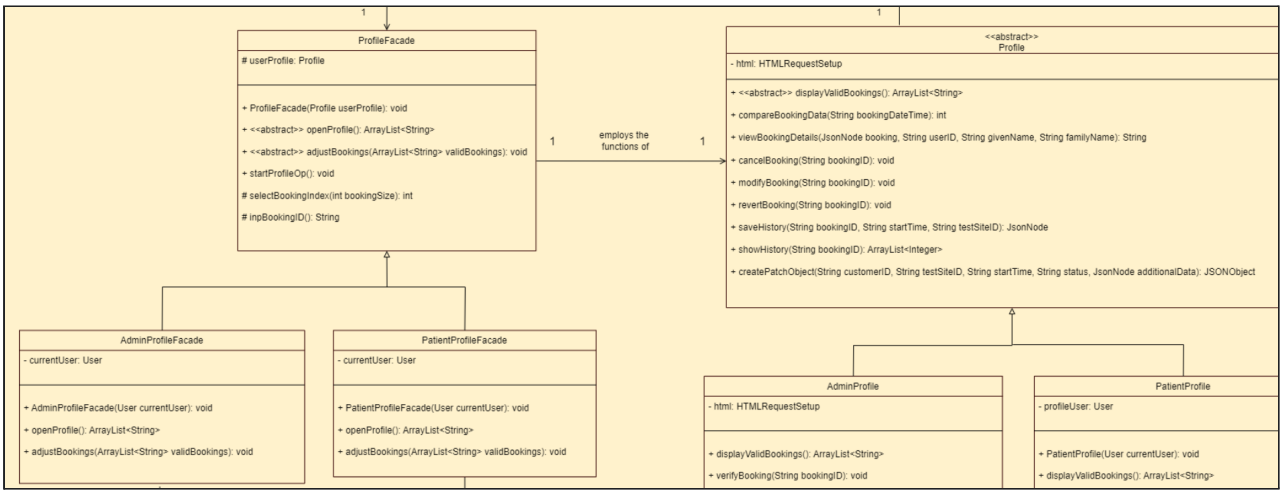
Design Patterns

New Patterns Applied

Façade

The new design pattern that was introduced into the system is façade. It is used to provide a simpler interface for the profile, booking, and test site subsystems with methods that incorporate the key features of their classes. When designing and refactoring the system, the main aspect that the team wanted to focus on is to ensure that all the methods with their associated functionalities in the CLI are uncomplicated with minimal object declarations to reduce its complexity while retaining its correctness. To that end, using the façade design pattern is perfect for this purpose as it does exactly that. The team also believes that this would allow the main class used for the CLI, the *WebApplication* class, to adhere better to the single-responsibility principle since there are less individual methods that solely call on the subsystem classes. With the case that new functionalities are added to the subsystems, new key methods can be added accordingly to the façades that incorporate them, supporting extensibility.

Aside from the advantage of isolating the code from a class's complexity, which would make things easier to understand from a user's point of view, one disadvantage associated with façades is that a god class can potentially be created, which would violate the single responsibility principle (Refactoring.Guru, 2022). However, in our system, this is somewhat circumvented by employing the usage of multiple façades, most notably through the profile subsystem. In it, we use two different façades to run the integral functionalities of the two profile types: user and administrator.



Initial Design and New Features

In terms of the previous design patterns that were used in Assignment 2 (*factory, strategy, and bridge*), no changes had to be made to them with the new Assignment 3 requirements. The reason why is because all the new requirements do not need the core functionality of the existing code to be modified in any way. For example, one of the main features that are introduced in Assignment 3 is modifying the booking data through a user's profile. The team's current implementation of the booking subsystem, which employs the usage of the strategy, does not need any adjustments to be made to its classes and methods since the new features do not require any dependencies or associations to it.

As for how the existing design pattern supports the new extensions, the team thinks that there is not much involvement for them with regards to extensibility. These extensions are primarily focused on user profiles, which was developed as an entirely separate subsystem, and booking adjustment, which would not require the strategy pattern used in the booking subsystem to be utilized as its main purpose is to differentiate the behaviour between the two different booking types. Booking adjustments are more general in application, so there is no need to incorporate it with this pattern. For the other design patterns, they are also not used as they are minimally involved in the new extensions.

To give a short description why no other design patterns are introduced other than façades, the team felt that there is no need to further complicate the code by forcefully adding new patterns when they are not required due to the already minimal involvement from the existing ones.

Design Principles

New Principles Applied

Encapsulation

One of the team's main focuses while refactoring the code for Assignment 3 is to ensure that each class is encapsulated. When looking over the code, the team felt that the design could be improved by making the fields of each class private so that they cannot be easily modified by outside sources. By doing this, the team would be reducing the amount of messy code and prevent certain snippets of code from breaking whenever the attributes change in any way, improving maintainability. The security of the code is also enhanced as it is protected from external inheritance (AppTech Systems, 2020). Other than this emphasis on maintainability and security, there is no particular scenario that forms the basis of this addition.

Previously, the team used the protected modifier for certain attributes for convenience, such as the ones in the Booking class. The simple adjustment made to attributes like this is to simply change the modifier to private. Getters and setters are only developed for the ones that need to be accessed externally and not for those that do not. This is done to also adhere to the YAGNI principle..

DRY Principle

The second design principle that is emphasized in Assignment 3 is the "Do Not Repeat Yourself" or DRY principle. Similar to the focus on encapsulation, there is no particular scenario or feature that causes this principle to be adhered to other than to improve the existing design of the system. When creating the new features, any common snippets of code are placed into their own methods to prevent unnecessary repetition. For example, if a booking is modified or reverted to a previous version, the PATCH endpoint has to be called to update the booking in the web service. From that, the appropriate JSON object with the required fields need to be made to perform the operation. Therefore, instead of including code for this creation process in every part of the code where it is required, the team developed a method called *createPatchBooking* that employs the same functions.

Doing this drastically improves the code's maintainability. Using the same example, in case there are some changes to the web service that changes the requirements of the PATCH endpoint, there would be no need to go to every part of the code that employs the old functionality to make adjustments, which may be highly prone to errors and is a huge time waster. Instead with DRY, the team can just directly modify *createPatchBooking* and the correct changes would be applied. Besides that, the readability and reusability of the code would also be improved as these methods can be used for other similar functionalities (ScienTech Easy, 2022).

Package Principles

New Principles Applied

Acyclic Dependencies Principle

Following the adherence of the package principles with an emphasis on improving cohesion from Assignment 2, one of the principles that was not mentioned in the previous assignment is the acyclic dependencies principle (ADP). This principle dictates that there should not be a dependency cycle between the packages in the system.

Based on the design philosophy that the team has followed in Assignment 2, this principle is also followed as the current system is loosely designed with the structure of a tree. All the requirements for each functionality are held within their own package with little dependencies from others, which would prevent a cycle from forming in the first place. This does not change with the added features from Assignment 3 from the addition of the *profile* package. To further elaborate, the intention the team had with taking this approach is to ensure that good maintainability is retained as much as possible. If the ADP is violated, making changes in one package would heavily affect the others, which is a sign of bad design and can waste a lot of resources to fix.

Rationale on Lack of Additions

Beyond the ADP principle, no other package principles are introduced from the lack of the suitability they have to the current design approach taken in regards to the packages. For example, the common closure principle generally promotes large packages so that any changes that are localized in the same package. There are also some depedencies between packages, most notably with the *user* package. To adhere to these principles, the main intention of the team in regards to the design to provide clean segmentation of each core functionality and its associated classes would be violated.

Therefore, since the team does not want their current approach to package design to be changed, there is a lack of new package principles introduced in Assignment 3. The team has no better justification to provide other than this.

Architecutral Pattern

Architectural Pattern Used

The software architecture pattern that is used for Assignment 3 is the "Model-View-Controller" or MVC pattern, most notably the passive variant. The reason why this pattern in particular is used is because it best supports the current design of the system with its frontend UI. To elaborate, the code developed in Assignment 2 already has the existing framework to incorporate MVC through the *WebApplication* class. This class serves as a centralized part of the system where the user inputs are processed and the functionalities are run. From that, the team believes that some simple adjustments and refactoring could be done to implement MVC.

The advantages and disadvantages of the MVC architecture are as follows:

Advantages:

- **Improved Maintainability:** This aspect is improved as the UI is separated into the three components, which allows each individual portion to be developed and maintained independently.
- **Easy to Test:** As a result of the separation of concern, each component can be tested individually.
- **Prevents Acyclic Dependencies:** Since the UI is separated into the view and controller components, the model does not need to depend on the UI to inform of updates.

*Disadvantages***:**

- **Model Independence:** While preventing acyclic dependencies, having the model be independent also prevents any change of states from being reported, at least for the passive version of the implementation.

An important thing to mention is that while the active version of the MVC can negate the disadvantages, the team felt that adding the required functionalities of an observer among other things are unnecessary because of the usage of the CLI as the frontend. The restrictive nature of the CLI display discourages the need to make those implementations, so the team decided to omit it to prevent any complications.

How It is Implemented

In its current iteration, a series of classes named *WebApplication* is used to connect all the different functionalities from their different packages to the CLI through the façade classes. As a result of this, the model can be considered as these façade classes as they are responsible for the manipulation of data. In addition, whenever a functionality is completed, a message indicating the status of the operation would also be displayed, such as the booking creation returning a success or failure message depending on the results of the operation.

As for the view aspect of the operation, a specific class called *WebApplicationView* is used. This class contains the content that showcases the available options that the user can take in regards to the functionalities they can perform with the system, not much else to it.

Lastly, the *WebApplicationController* class which serves as the controller as in its namesake, would call the required model and view objects upon instantiation. When the main method is run, the content from the view would be displayed with the user inputting an option based on the selections that are displayed. Based on the input, a specific functionality would be executed through a series of if-else statements.

That being said, the implementation of MVC is not perfect. Some of the model classes and parts of the controller still feature some statements that print sentences to the CLI. The team feels that these statements are minimal enough that they can just be integrated together with their respective code. In addition, the team would also like to reduce any unneeded dependencies or associations from being made just by putting every print statement in the view class.

Refactoring

There are already some mentions of refactoring to incorporate the new design patterns and principles above. However, the team would like to mention **some of** the specific techniques that are used to accomplish this feat.

Techniques Applied

Extract Method

When looking over the initial versions of the code for the new functionalities, one of the main things that was done is to extract any common code that is used in the system and separate it into its own method. By doing this, duplicate code is removed and the codebase's maintainaiblity is improved (Refactoring.Guru, 2022). This incorporates the DRY design principle as mentioned above with its associated advantages. As for the disadvantages, there runs the risk that the newly created method would be barely used, violating the YAGNI principle.

Extract Class

This refactoring technique can be most notably seen through the development of the MVC pattern and façades. In the initial version of the code, a lot of the methods that run the indiviudal functionalities of the different system packages are put into the original *WebApplication* class. These methods are separated into their own façade and view classes to better adhere to the single-responsibility principle.

Move Function/Method

One of the most notable things the team did while incorporating the new extensions is to move the methods more commonly used in another class to it. In this case, this is most notably seen with the functionality for booking modification, which include patching, cancelling, and deleting a particular booking. Originally, the team developed these in the Booking set of classes as the functionalities incorporate bookings some way or form. However, while developing the functionalities for the user profiles, it was decided that these methods

---

would be better placed in the profile classes as the adjustments can only occur for valid bookings that are displayed from it. This allows the code to be more internally coherent and have reduced dependencies (Refactoring.Guru, 2022).

References

AppTech Systems. (2020, February 3). *5 benefits of applying dry principle to your code*. Apptech Systems. Retrieved May 23, 2022, from https://apptechsystems.net/5-benefits-of-applying-dry-principle-to-your-code/

ScienTech Easy. (2022). Encapsulation in Java | Realtime Example, Advantage - Scientech Easy. Retrieved 23 May 2022, from https://www.scientecheasy.com/2020/07/encapsulation-in-java.html/

Refactoring.Guru. (2022). Extract Method. Retrieved 23 May 2022, from https://refactoring.guru/extract-method

Refactoring.Guru. (2022). Extract Subclass. Retrieved 23 May 2022, from https://refactoring.guru/extract-subclass

Refactoring.Guru. (2022). Facade. Retrieved 23 May 2022, from https://refactoring.guru/design-patterns/facade

Refactoring.Guru. (2022). Move Method. Retrieved 23 May 2022, from https://refactoring.guru/move-method