MA_Lab04Team10

Design Pattern

Abstract Factory

We created abstract factory classes in our system for handling API calls such as Get and Post requests. This was done by creating three abstract classes which are Get, Post and Patch classes as well as an interface called APIFactory class.

The APIFactory class will be the interface for all the Factory classes that are related to making API calls. The APIFactory will contain three creational methods which are createGet(), createPost() and createPatch(). These methods return abstract API types (Get, Post and Patch) and so any classes that implement this interface are expected to implement the three creational methods.

The classes that implement the APIFactory interface are all related to the web service APIs (UserFactory, TestingSiteFactory, BookingFactory and CovidTestFactory). The advantage of creating these classes is that it streamlines the process of making API calls easier as any classes that have dependency to these factory classes are able to make use of their methods. The advantage of this design is that it hides the low-level implementation of the code and the classes only need to know the functionality of the methods. This adheres to the Open/Closed Principle as these methods are closed for any modification but are opened for extensibility.

The disadvantage of this makes it harder to add more methods in subclasses as it breaks the Open/Closed Principle. If we want to avoid this, we have to add the corresponding methods in the abstract classes but it will break the Liskov Substitution Principle because the subclasses are strengthening the precondition or weakening the postcondition.

Singleton

Our team made use of the Singleton design pattern when creating the AuthenticateSingleton class.

The AuthenticateSingleton class is to determine whether the user is authenticated in all other web pages once the user is logged in.
Considering the process is only done once, there should only be one instance of that being created. Therefore, by restricting the class to only one instance, we can ensure that there will be no other instances created anywhere in the system.

The AuthenticateSingleton class is used in controller classes such as BookingController, InterviewController, HomeBookingController, LoginController and MainController. The purpose of the singleton class is to restrict access to certain pages depending on the roles of the user (e.g. Receptionists are unable to access Interview Page).

The advantages of Singleton is that we can ensure that it is the only instance of the AuthenticateSingleton class throughout the entire system.

The entire system will gain a global access point to that instance which means that it can be called upon in any classes.

Since the constructor of the Singleton class is made private, we can prevent other objects from creating the Singleton class directly by using the ‘new’ operator.

Our team also utilises the Lazy Initialisation method where we only initialise the instance when we request for the first time, which makes it more memory efficient.

There are however disadvantages when using Singleton design patterns. First of all, creating a Singleton class violates the Single Responsibility Principle because the object creation and management of the lifecycle is being handled by the class itself.

Our AuthenticateSingleton class contains many static variables and methods which can potentially break the encapsulation of the OOP (Information leaks).

Package Principles

Acyclic Dependencies Principle

Following our class diagram, our team ensures that any dependencies in our package will not form a cycle with one another. For example, the api and domain packages are standalone packages and do not depend on other packages. The model package contains dependencies that are related to API packages and the controller package contains dependencies that are related to all of the aforementioned packages.

Through this principle, we ensure that no packages are dependent on one another and reduce the complexity of the code as well as allowing debugging to be easier.

Common Reuse Principle

Our team grouped all classes that make API calls into one API package. The API package contains an APIFactory interface as well as the three abstract classes GET, POST and PATCH class. Any classes that implement the APIFactory interface are said to be tightly coupled together and grouped into one package.

One advantage of this principle is that classes in this package can be reused in more than one place and this makes maintainability easier.

Design Principles

Open-closed principle

In the following class diagram, we made the APIFactory class as an interface to adhere to the open/close principle where we put the blueprint of our method inside the class which can be inherited by the child classes as well as allowing us to have other child factory classes we can add as an extension.

This is especially true for BookingFactory, TestingSiteFactory, CovidTestFactory and UserFactory where they have common methods which are createPost, createPatch and createGet. Furthermore, we are planning to add a PhotoFactory class in the future as an extension of the APIFactory which has common methods.

Other than that, the reason we implement an APIFactory interface class using the Open closed principle is because we want our system to be able to add features to our program in the future. For example, maybe in the future we want to add another new method to the system we can easily extend from the APIFactory interface class without breaking the old APIFactory methods.

The advantage of open closed principle is that it becomes easier to add any features or methods to the code base without changing the underlying methods that may potentially break the code or any other classes that class that particular method.

The disadvantage is that since the code can be extended, it may introduce a lot of new classes or interfaces which thus makes the entire code structure more complicated.

Single Responsibility principle

In our code we utilise this principle when we are coding all of our controller class, this is done by having each controller class handle each of the html page. For example, we have BookingController and HomeBookingController as two separate classes instead of having one Controller class that handles multiple booking html pages because we want these two classes to handle different html pages of different booking types made by the user. Therefore, this principle is applied.

Another scenario the single responsibility principle is applied when we are using abstract factory design patterns for our Get, Post and Patch api call. We separate the responsibility of Get user data from api using UserGet class, Post user data to api using PostUser class and Patch user data to api using PatchtUser class.

The advantage of using the single responsibility principle is it makes our class more cohesive because if the responsibility needs to change, all the pieces we need are already present there.

The disadvantage of using the single responsibility principle is that the class will become too small and over complicate our design.

Reference list

JournalDev. (n.d.). *Abstract Factory Design Pattern in Java.* Retrieved From https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java

Refactoring Guru (n.d.). *Abstract Factory*. Retrieved From https://refactoring.guru/design-patterns/abstract-factory

Refactoring Guru (n.d.). *Singleton*. Retrieved From https://refactoring.guru/design-patterns/singleton

Saxena, B. (2018). *How to use Singleton Design Pattern in Java.* Retrieved From https://dzone.com/articles/singleton-in-java