**Individual Project** : CMPE 202

**Name**: Netra Amrale

**SJSU ID**: 015951793

**Application Name**: Flight Booking

**GitHub Repo Link:** <https://github.com/gopinathsjsu/individual-project-neetra>

**Index**

1. [Detailed instructions of building the project and steps to execute the same.](#inst)
2. Answers to the following questions:
   1. [Describe what is the primary problem you try to solve](#pp)
   2. [Describe what are the secondary problems you try to solve](#sp)
   3. [Describe what design pattern(s) you use how (use plain text and diagrams)](#dp)
   4. [Describe the consequences of using this/these pattern(s).](#conse)
   5. [Class diagram.](#cd)
3. [Junit test cases should be written for all the design pattern classes/methods. Screenshots of test execution and result should be included.](#test)

1. **Detailed instructions of building the project and steps to execute the same.**
2. Build the project

* Navigate to your project directory
* Execute “mvn clean install”

[Example:](https://github.com/gopinathsjsu/individual-project-neetra/blob/main/images/cleaninstall.PNG)

Text

Description automatically generated

* Execute common “mvn compile”

Example:

Text

Description automatically generated

1. Steps to run the project

* [Powershell:](https://github.com/gopinathsjsu/individual-project-neetra/blob/main/images/powershellrunn.PNG)

mvn exec:java "-Dexec.mainClass=test.RunClient" "-Dexec.args=`"path1 path2 path3 path4 `""

Example:

Text

Description automatically generated

* [Cmd](https://github.com/gopinathsjsu/individual-project-neetra/blob/main/images/CMD.PNG)

mvn exec:java -Dexec.mainClass=test.RunClient -Dexec.args="path1 path2 path3 path4 "

Example:

Text

Description automatically generated

2. **Describe what is the primary problem you try to solve**

* The primary problem is to validate user flight bookings to generate valid and invalid user flight booking entries.
* This primary problem is solved using the Chain of responsibility and Singleton design pattern.
* CoR assists to validate card numbers and generate valid flight bookings.
* Singleton Pattern provides a global instance of all flights, which aids to update it when card number validation passes for valid user flight booking.

1. **Describe what is the secondary problem you try to solve**

* The flight booking application should generate two files one of them contains valid flight bookings (Output.csv) and another invalid flight bookings (Output.txt).
* The secondary problem is to create these two different types of files.
* Factory Method Pattern helps to create two different types of file writers like CSV and text file writers. CSV file writer deserializes valid flight bookings and appends them in CSV file. Text File Writer deserialize invalid flight bookings and appends them to the text file writer.

1. **Describe what design pattern(s) you use how (use plain text and diagrams**

I am using the following design patterns:

* 1. Singleton Pattern

Diagram

Description automatically generated

* The creational singleton design pattern is used to create single instance for list of flights.
* The detail of every flight is stored in “Flight” class. Thus, details of all flights are stored in Array List of Flights.
* This Array List of flights is instantiated using SetSingletonFlights in SingletonFlights class to ensure only one instance of the flight list is created. The read-only global access to list of flights is provided using getAllFlights method.
* The global flight list object is updated when validation for a booking is passed.
  1. Factory Method Pattern

Diagram

Description automatically generated

* The result generated can have a valid booking and invalid booking entries.
* CSV Output file is created to store valid bookings.
* Invalid bookings are stored in a text file.
* Thus, we need two types of file writers to write objects either to CSV or text files.
* ‘getWriter’ method of FileWriter accepts the type of writer to generate and instantiates CSVFileWriter and TextFileWriter accordingly.
  1. Chain of Responsibility

Diagram

Description automatically generated

* **Chain of Responsibility** is a behavioral design pattern that enables to perform validation of credit card numbers.
* AmexCardValidator, DiscoverCardValidator, DigitCardValidator, MasterCardValidator and VisaCardValidator are handler that returns true or false or passes the request to the next handler.
* Chain for card number validation is set as below

A picture containing diagram

Description automatically generated

* Every validator validates credit card number in different way.
* Credit card number validation request passes sequentially along the dynamic chain of potential handlers as demonstrated above, unless one of them handles it.

1. **Describe the consequences of using this/these pattern(s).**

Consequences of using the following patterns:

1. Chain of Responsibility

* Allows controlling the order of credit card number validation.
* Each of the Card Validator classes is decoupled from another card validator class.
* New card validator can easily be inserted to handle the credit card number validation and set the chain accordingly.
* CoR seems difficult to test. Moreover, to unit test this pattern we must mock methods to verify its call as shown [here](https://github.com/gopinathsjsu/individual-project-neetra/blob/main/src/test/java/test/CardValidatorTest/CardValidatorTest.java).
* If the chain is not set properly, it may cause a cycle call, and debugging becomes difficult.

1. Factory Method Pattern

* Factory Method Patterns decouples Creator class (FileWriterFactory) from the concrete classes like CSVFileWriter and TextFileWriter enhancing the modularity of the code.
* It provides a high level of abstraction as all code is behind an abstraction. This makes it more difficult to read the code.
* When implementing factory patterns, it becomes difficult to main interface segregation principle.

For e.g.: Initially interface IFileWriter had two methods writeValidBookings and writeInvalidBookings. Thus CSVFileWriter should not implement writeInvalidBookings methods. Thus, workaround java generics are used, and the interface contains only writeObjects method.

1. Singleton Pattern

* Singleton pattern is conducive to creating one and only one instance of an array list of flights using SetsingletonFlights . Thus, all modifications are made to only this instance.
* Global read access point is created by using the method getAllFlights.
* The problem may arise for multi-threaded operations if SetSingletonFlights method is called multiple times. It may overwrite the original instance.
* SingletonFlights violate the [single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) because they control their own creation and lifecycle.
* It is difficult to unit test this pattern as it introduces a global state to the application.

You cannot completely isolate the classes dependent on the singletons. Thus, the order of tests matters a lot and cannot run multiple tests in parallel.

1. **Class diagram.**

HD image is [here](https://github.com/gopinathsjsu/individual-project-neetra/blob/main/images/Class%20Diagram0.png)

Diagram

Description automatically generated

1. **Junit test cases should be written for all the design pattern classes/methods. Screenshots of test execution and result should be included.**

* Run command “mvn test”

Text

Description automatically generated