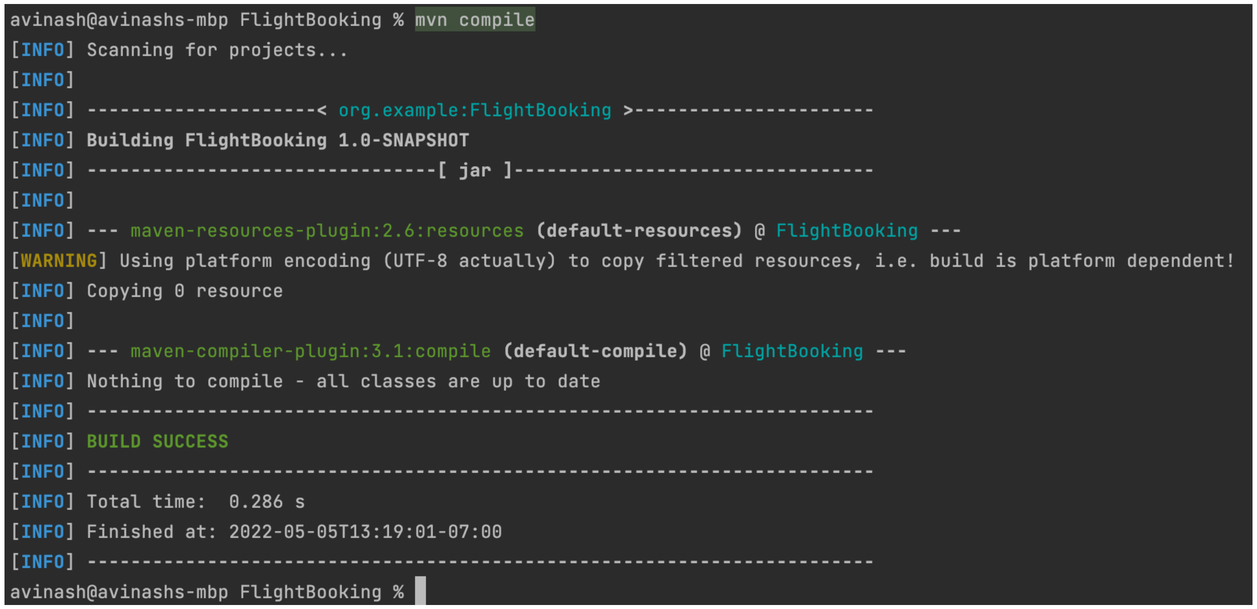
CHAPTER 1

Detailed instructions for building the project and steps to execute the same.

Have followed the steps provided in the canvas, so you can use the same steps as outlined in the canvas.

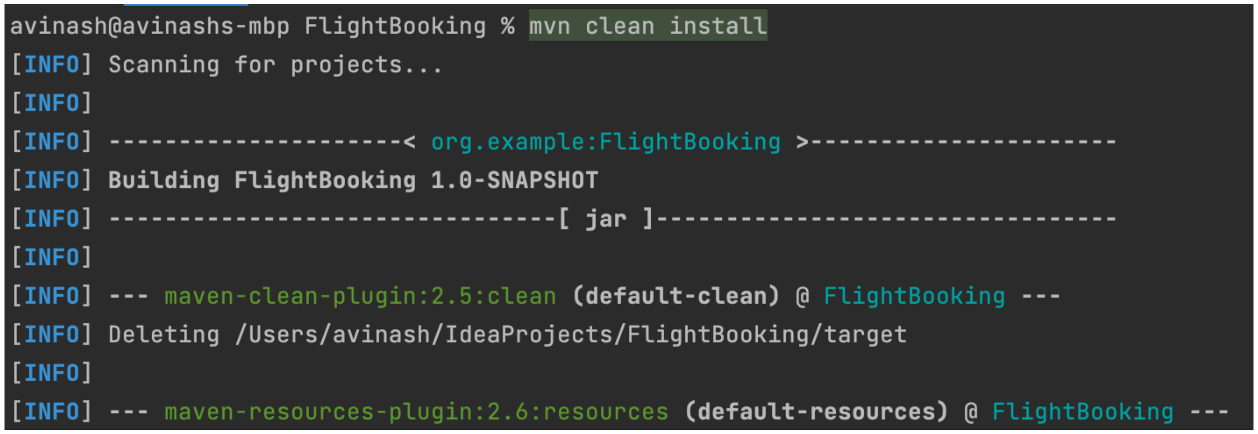
Steps to Execute:

- Open the command line where the project directory is located and execute the below steps
    - **mvn compile**



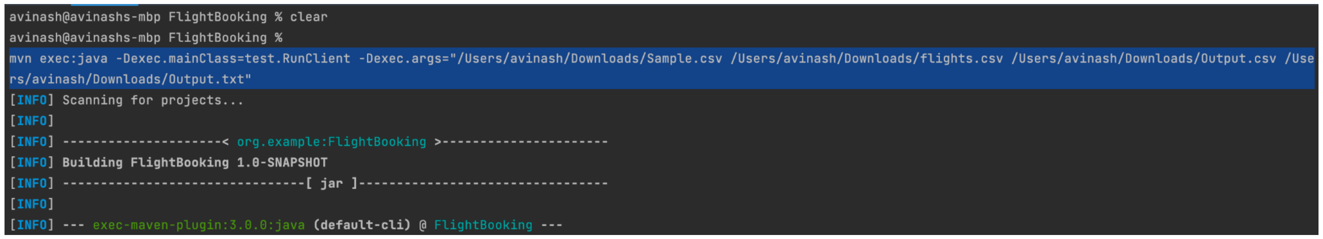
```
avinash@avinashs-mbp FlightBooking % mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.example:FlightBooking >----------------------
[INFO] Building FlightBooking 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ FlightBooking ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ FlightBooking ---
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.286 s
[INFO] Finished at: 2022-05-05T13:19:01-07:00
[INFO] ------------------------------------------------------------------------
avinash@avinashs-mbp FlightBooking %
```

    - **mvn clean install**



```
avinash@avinashs-mbp FlightBooking % mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< org.example:FlightBooking >---------------------
[INFO] Building FlightBooking 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ FlightBooking ---
[INFO] Deleting /Users/avinash/IdeaProjects/FlightBooking/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ FlightBooking ---
```

- Execute the below maven command to execute with arguments (Path to where the input file is located and path to where output file should be located) passed via command line
- **mvn exec:java -Dexec.mainClass=test.RunClient -Dexec.args="<arg1> <arg2 > <arg3> <arg4>"**



```
avinash@avinashs-mbp FlightBooking % clear
avinash@avinashs-mbp FlightBooking %
mvn exec:java -Dexec.mainClass=test.RunClient -Dexec.args="/Users/avinash/Downloads/Sample.csv /Users/avinash/Downloads/flights.csv /Users/avinash/Downloads/Output.csv /Use
rs/avinash/Downloads/Output.txt"
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< org.example:FlightBooking >---------------------
[INFO] Building FlightBooking 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ FlightBooking ---
```

- arg1 – path to the input data (Sample.csv)
- arg2 – path to flight details to populate DB (flights.csv)
- arg3 – path to Output.csv
- arg4 – path to Output.txt
-

CHAPTER 2

Describe what is the primary problem you try to solve

The primary issue I'm attempting to address is the development of a flight booking application that employs various design patterns and software engineer practices to provide a general reusable solution for the problem domain.

From User Perspective:
The application's user would enter the Flight DB information, Booking details, and output file path. The application would process the data and generate output files with confirmed booking details in Output.csv and errors in Output.txt.

Describe what are the secondary problems you try to solve

- The application should maintain an internal, static database (an inventory of flight details. This means that if we re-run the program, the changes to the data will be lost.
- We must follow good software engineer practices and design patterns such that code should be open for new changes. Have tried to follow SOLID principles wherever possible.

Have followed the steps/workflow of the application mentioned in the CANVAS.

- *Validate if the requested flight exists.*
- *If the flight exists, validate the number of seats requested for the category.*
- *After this validation, if the booking is valid, calculate the total price (NoOfSeats * price)*
- *Take the card number of the user and validate it using the given rules:*

    - *Visa card: has a length of either 13 or 16. It begins with a 4*
    - *Mastercard has a length of 16. Begins with 5 and the 2nd digit begins from 1 to 5 inclusive*
    - *Discover length 16, and the first 4 digits beginning from 6011*
    - *Amex: has a length of 15 and starts with 3. 2nd digit must be 4 or 7*
    - *Any card greater than 19 or not satisfying the above conditions is considered invalid.*
- *If the card is valid then modify the available seats for that category and flight number*
- *Then output the CSV list with booking name, flight number, Category, number of seats booked, and total price.*
- *In case, it is an incorrect request at any of the steps, generate and output a TXT file with the message "Please enter correct booking details for <booking_name>:<reason>" and include the information with incorrect information. For example, Please enter the correct booking details for John: invalid flight number.*

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

For this Flight Booking Problem domain, I have taken 3 Patterns into consideration:

- ***Singleton Pattern for maintaining a static database.***
- ***State Pattern for transition between various states.***
- ***Chain of Responsibility for Card Validation.***

Other Pattern Consideration, but not used in my project:

- *Decorator Pattern for writing files in a different format*
- *Strategy pattern for Card validation*

Singleton Pattern

To Store and Maintain an internal, static database of HashMaps for Flight Information. Also, have ensured to have a thread-safe Singleton(Double Check Synchronization).

Problem & Context

When it is necessary to have only one instance of a given class during the lifetime of an application. This could be due to a lack of resources or, more commonly because only a single instance of the class is sufficient.

Solution

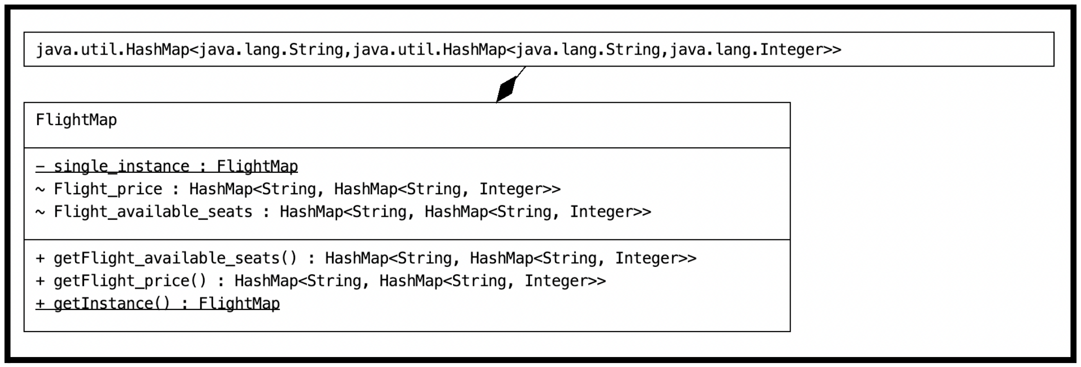
Create a class that includes a method for creating a new instance of the object if one does not already exist. If one does exist, it returns a reference to the already existing object.

My problem domain

Class in my application should only have one instance available to all clients; for example, a single database(**HashMaps containing price and seat availability for each flight and category**) shared by various parts of the program.

Class Name
FlightMap

Class Diagram


java.util.HashMap<java.lang.String,java.util.HashMap<java.lang.String,java.lang.Integer>>
FlightMap
- single_instance : FlightMap
~ Flight_price : HashMap<String, HashMap<String, Integer>>
~ Flight_available_seats : HashMap<String, HashMap<String, Integer>>
+ getFlight_available_seats() : HashMap<String, HashMap<String, Integer>>
+ getFlight_price() : HashMap<String, HashMap<String, Integer>>
+ getInstance() : FlightMap

Consequences of Using Singleton

- *Permits subclassing*
- *Controls access*
- *Enhances flexibility*

Disadvantages:

- *Violates the Single Responsibility Principle.*
- *The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.*
- *The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.*

State Pattern

For transition between various states of my application and carry out operations pertaining to that state.

Problem & Context

Useful for creating an efficient structure for a class whose typical instance can exist in a variety of states and behave differently depending on the state it is in. Some or all of an object's behavior is totally influenced by its current state in the case of such a class.

Solution

The State pattern advises separating state-specific activity from the Context class and storing it in a collection of State classes. Each of the many different states in which a Context object can exist can be mapped into a separate State class.

My problem domain

Here, I have used a common state interface called '*BookingState*' with operation as an abstract function. So, in each specific state, there would be a state-specific operation that would be carried out. I have also created a *setstate method* in the context class for state transitions and *current state* member to store the current state of the application.

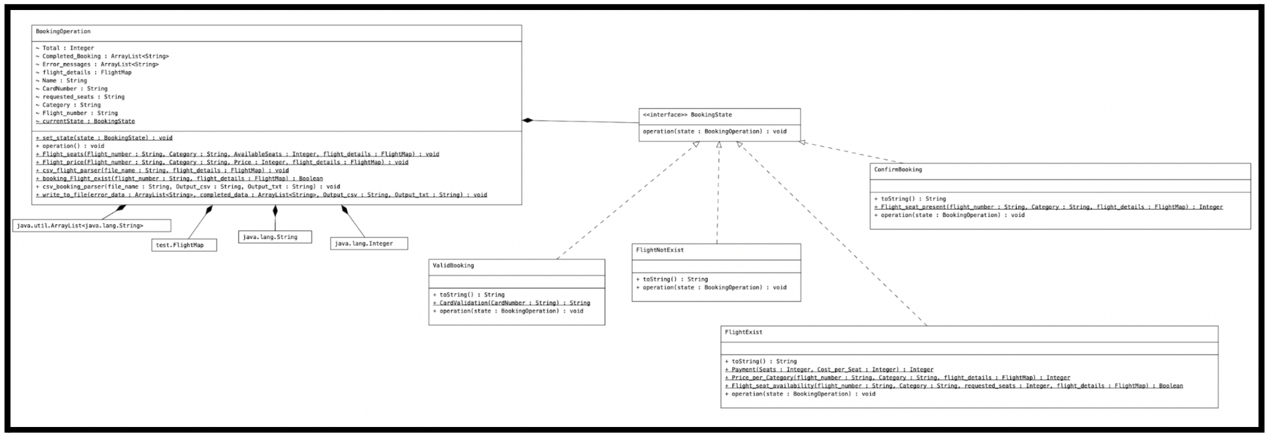
Interface

BookingState

State Classes

FlightExist
FlightNotExist
ValidBooking
ConfirmBooking

Class Diagram


BookingOperation
~ Total : Integer
~ Completed_Booking : ArrayList<String>
~ Error_messages : ArrayList<String>
~ flight_details : FlightMap
~ Name : String
~ CardNumber : String
~ requested_seats : String
~ Category : String
~ Flight_number : String
~ currentState : BookingState
+ set_state(state : BookingState) : void
+ operation() : void
+ Flight_seats(Flight_number : String, Category : String, AvailableSeats : Integer, flight_details : FlightMap) : void
+ Flight_price(Flight_number : String, Category : String, Price : Integer, flight_details : FlightMap) : void
+ csv_flight_parser(file_name : String, flight_details : FlightMap) : void
+ booking_Flight_exist(flight_number : String, flight_details : FlightMap) : Boolean
+ csv_booking_parser(file_name : String, Output_csv : String, Output_txt : String) : void
+ write_to_file(error_data : ArrayList<String>, completed_data : ArrayList<String>, Output_csv : String, Output_txt : String) : void
java.util.ArrayList<java.lang.String>
test.FlightMap
java.lang.String
java.lang.Integer
<<interface>> BookingState
operation(state : BookingOperation) : void
ConfirmBooking
+ toString() : String
+ Flight_seat_present(flight_number : String, Category : String, flight_details : FlightMap) : Integer
+ operation(state : BookingOperation) : void
ValidBooking
+ toString() : String
+ CardValidation(CardNumber : String) : String
+ operation(state : BookingOperation) : void
FlightNotExist
+ toString() : String
+ operation(state : BookingOperation) : void
FlightExist
+ toString() : String
+ Payment(Seats : Integer, Cost_per_Seat : Integer) : Integer
+ Price_per_Category(flight_number : String, Category : String, flight_details : FlightMap) : Integer
+ Flight_seat_availability(flight_number : String, Category : String, requested_seats : Integer, flight_details : FlightMap) : Boolean
+ operation(state : BookingOperation) : void

Consequences:

- *state-specific behavior is localized and is partitioned for different states*
- *makes state transitions explicit*

- *State objects can be shared.*

Disadvantages:
- *If a state machine merely has a few states or changes infrequently, applying the pattern may be overkill.*
- *Requires a significant amount of code to be written*

Chain of Responsibility

Used this Pattern for the Card Validation. Validation based on a Chain.

Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it

Problem & Context

When there is more than one object that can handle or fulfill a client request, each of these potential handler objects can be arranged in the form of a chain, with each object having a pointer to the next object in the chain.

Solution

The client can choose the sequence in which the objects compose the chain dynamically at runtime. A consistent interface should be provided by all potential handler objects. The client object, as well as any of the handler objects in the chain, does not need to know which object will actually perform the request.

My problem domain
Here, I have used a chain of responsibility for card validation in form of a chain.

When a request comes into the Chain, the first chain would try to process this request, if not it passes it over to the next item in the chain.

In Validation State under validation operation, I would Perfrom Card validation using the above mechanism.

Interface:

CardValidation

Classes:

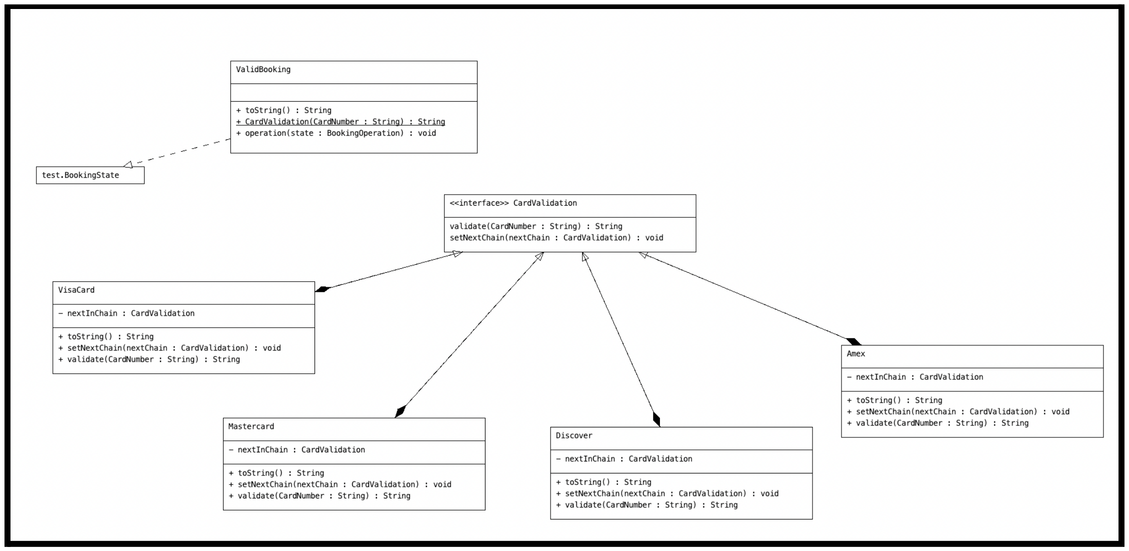
VisaCard

Mastercard

Discover

Amex

Class Diagram


ValidBooking
+ toString() : String
+ CardValidation(CardNumber : String) : String
+ operation(state : BookingOperation) : void
test.BookingState
<<interface>> CardValidation
validate(CardNumber : String) : String
setNextChain(nextChain : CardValidation) : void
VisaCard
− nextInChain : CardValidation
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
Mastercard
− nextInChain : CardValidation
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
Discover
− nextInChain : CardValidation
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
Amex
− nextInChain : CardValidation
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String

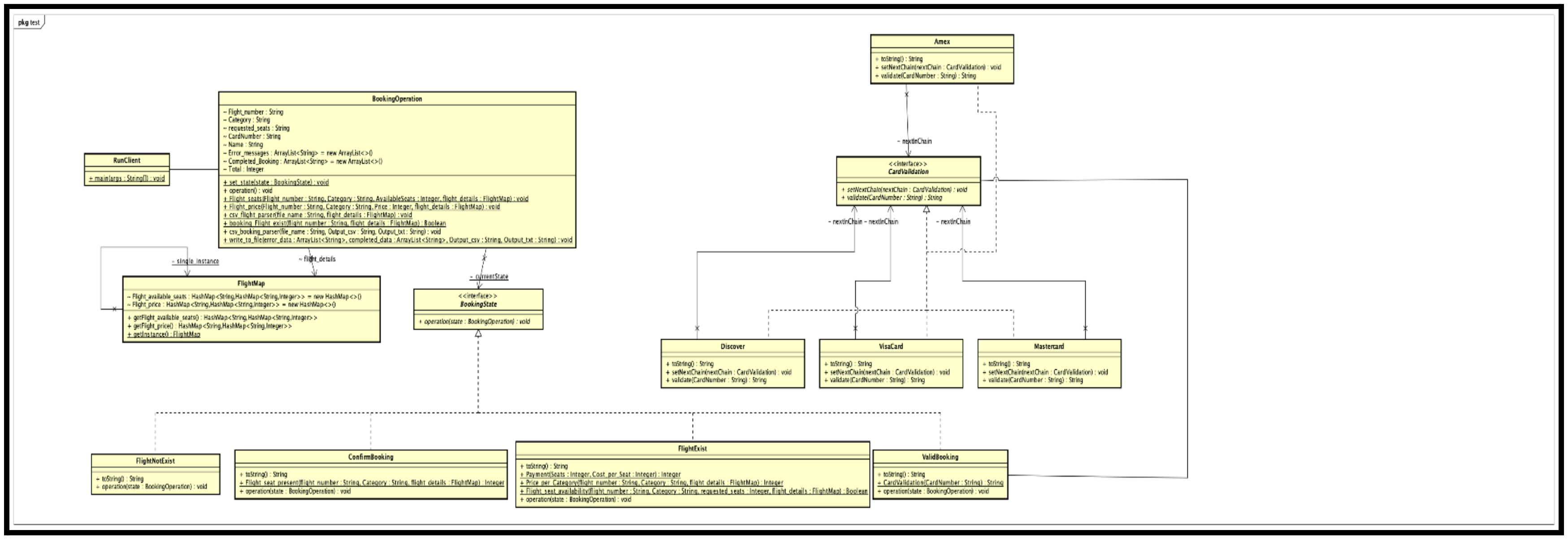
Consequences:

- *Reduced coupling*
- *Added flexibility in assigning responsibilities to objects*
- *Some requests may end up unhandled.*
- *Supports Single Responsibility Principle, Open/Closed Principle*

Disadvantages:

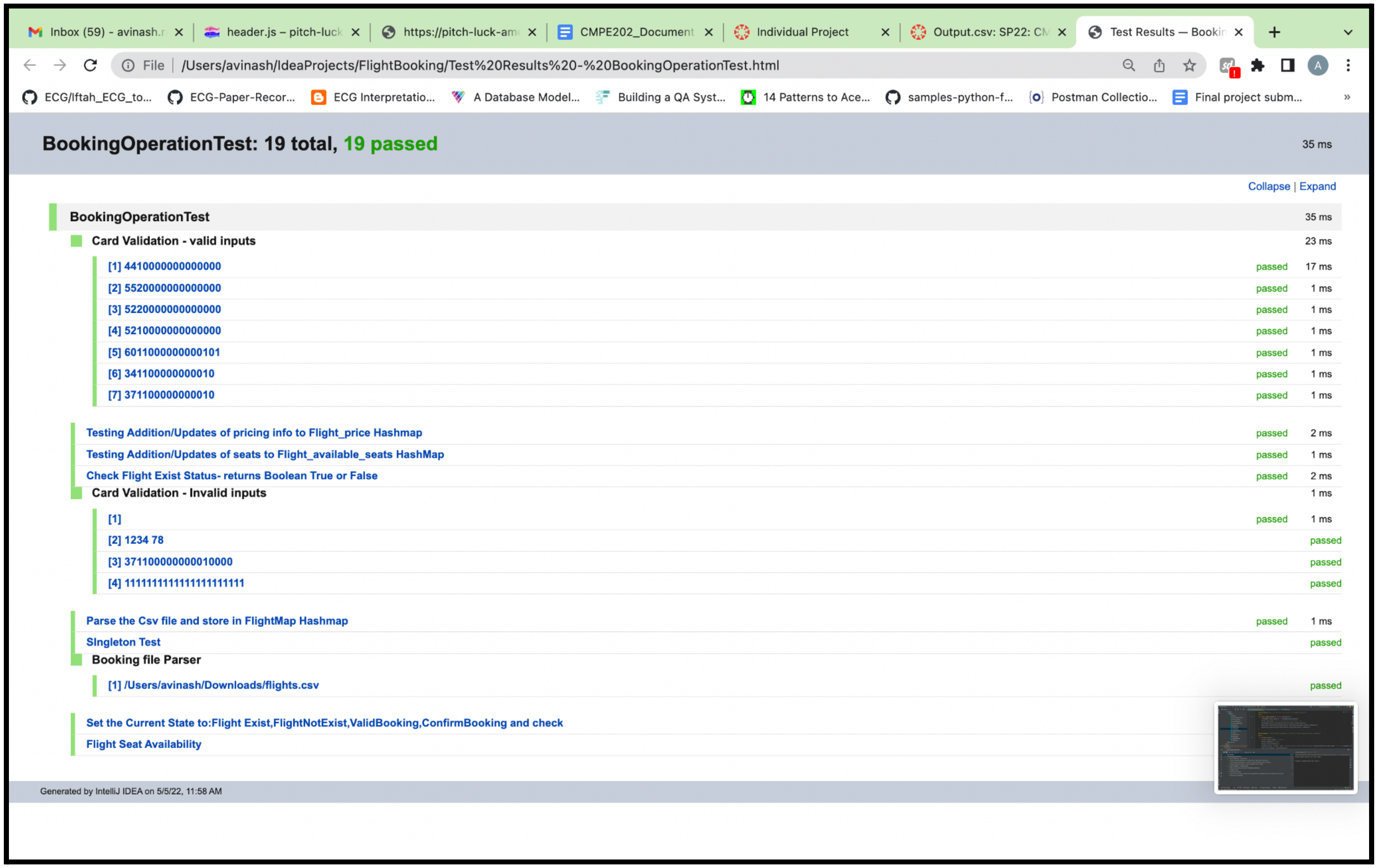
- *The system's performance will be affected, and the difficulty of debugging the code may end in a cycle call.*
- *Due to debugging, it may be difficult to observe the features of the operation.*

Entire Class Diagram of Application:


pkg test
RunClient
+ main(args : String[]) : void
BookingOperation
~ Flight_number : String
~ Category : String
~ requested_seats : String
~ CardNumber : String
~ Name : String
~ Error_messages : ArrayList<String> = new ArrayList<>()
~ Completed_Booking : ArrayList<String> = new ArrayList<>()
~ Total : Integer
+ set_state(state : BookingState) : void
+ operation() : void
+ Flight_seats(Flight_number : String, Category : String, AvailableSeats : Integer, flight_details : FlightMap) : void
+ Flight_price(Flight_number : String, Category : String, Price : Integer, flight_details : FlightMap) : void
+ csv_flight_parser(file_name : String, flight_details : FlightMap) : void
+ booking_Flight_exist(flight_number : String, flight_details : FlightMap) : Boolean
+ csv_booking_parser(file_name : String, Output_csv : String, Output_txt : String) : void
+ write_to_file(error_data : ArrayList<String>, completed_data : ArrayList<String>, Output_csv : String, Output_txt : String) : void
- single_instance
~ flight_details
- currentState
FlightMap
~ Flight_available_seats : HashMap<String,HashMap<String,Integer>> = new HashMap<>()
~ Flight_price : HashMap<String,HashMap<String,Integer>> = new HashMap<>()
+ getFlight_available_seats() : HashMap<String,HashMap<String,Integer>>
+ getFlight_price() : HashMap<String,HashMap<String,Integer>>
+ getInstance() : FlightMap
<<interface>>
BookingState
+ operation(state : BookingOperation) : void
Amex
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
- nextInChain
<<interface>>
CardValidation
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
- nextInChain
- nextInChain
- nextInChain
Discover
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
VisaCard
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
Mastercard
+ toString() : String
+ setNextChain(nextChain : CardValidation) : void
+ validate(CardNumber : String) : String
FlightNotExist
+ toString() : String
+ operation(state : BookingOperation) : void
ConfirmBooking
+ toString() : String
+ Flight_seat_present(flight_number : String, Category : String, flight_details : FlightMap) : Integer
+ operation(state : BookingOperation) : void
FlightExist
+ toString() : String
+ Payment(Seats : Integer, Cost_per_Seat : Integer) : Integer
+ Price_per_Category(flight_number : String, Category : String, flight_details : FlightMap) : Integer
+ Flight_seat_availability(flight_number : String, Category : String, requested_seats : Integer, flight_details : FlightMap) : Boolean
+ operation(state : BookingOperation) : void
ValidBooking
+ toString() : String
+ CardValidation(CardNumber : String) : String
+ operation(state : BookingOperation) : void

CHAPTER 3

JUnit Test Report



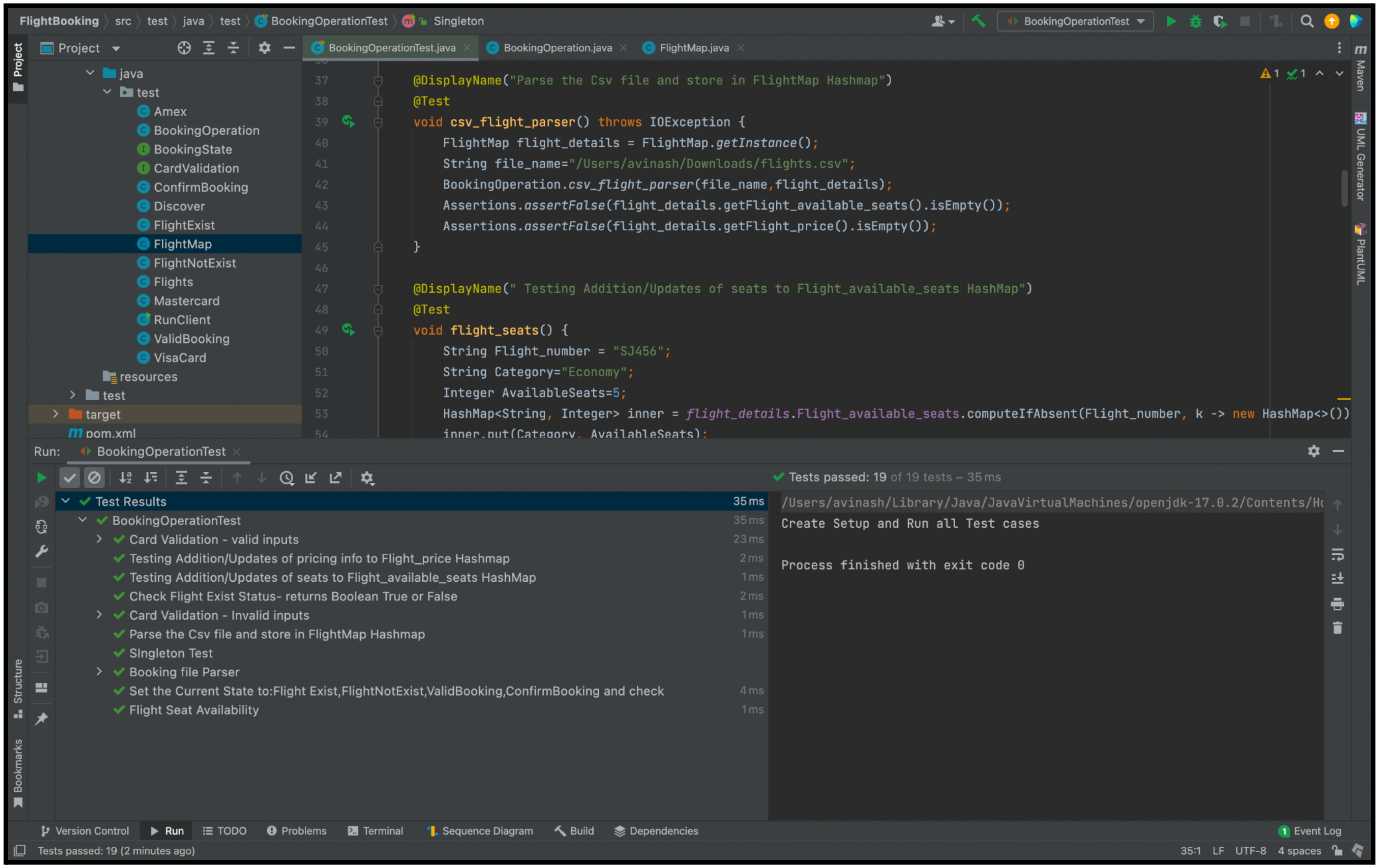
BookingOperationTest: 19 total, 19 passed — 35 ms

Collapse | Expand

```
BookingOperationTest                                                        35 ms
  Card Validation - valid inputs                                            23 ms
    [1] 4410000000000000                                        passed     17 ms
    [2] 5520000000000000                                        passed      1 ms
    [3] 5220000000000000                                        passed      1 ms
    [4] 5210000000000000                                        passed      1 ms
    [5] 6011000000000101                                        passed      1 ms
    [6] 341100000000010                                         passed      1 ms
    [7] 371100000000010                                         passed      1 ms
  Testing Addition/Updates of pricing info to Flight_price Hashmap        passed      2 ms
  Testing Addition/Updates of seats to Flight_available_seats HashMap     passed      1 ms
  Check Flight Exist Status- returns Boolean True or False                passed      2 ms
  Card Validation - Invalid inputs                                           1 ms
    [1]                                                         passed      1 ms
    [2] 1234 78                                                         passed
    [3] 3711000000000010000                                             passed
    [4] 11111111111111111111                                            passed
  Parse the Csv file and store in FlightMap Hashmap               passed      1 ms
  SIngleton Test                                                          passed
  Booking file Parser
    [1] /Users/avinash/Downloads/flights.csv                              passed
  Set the Current State to:Flight Exist,FlightNotExist,ValidBooking,ConfirmBooking and check
  Flight Seat Availability
```

Generated by IntelliJ IDEA on 5/5/22, 11:58 AM

SCREENSHOT FROM IDE:



```java
@DisplayName("Parse the Csv file and store in FlightMap Hashmap")
@Test
void csv_flight_parser() throws IOException {
    FlightMap flight_details = FlightMap.getInstance();
    String file_name="/Users/avinash/Downloads/flights.csv";
    BookingOperation.csv_flight_parser(file_name,flight_details);
    Assertions.assertFalse(flight_details.getFlight_available_seats().isEmpty());
    Assertions.assertFalse(flight_details.getFlight_price().isEmpty());
}

@DisplayName(" Testing Addition/Updates of seats to Flight_available_seats HashMap")
@Test
void flight_seats() {
    String Flight_number = "SJ456";
    String Category="Economy";
    Integer AvailableSeats=5;
    HashMap<String, Integer> inner = flight_details.Flight_available_seats.computeIfAbsent(Flight_number, k -> new HashMap<>())
    inner.put(Category, AvailableSeats);
```

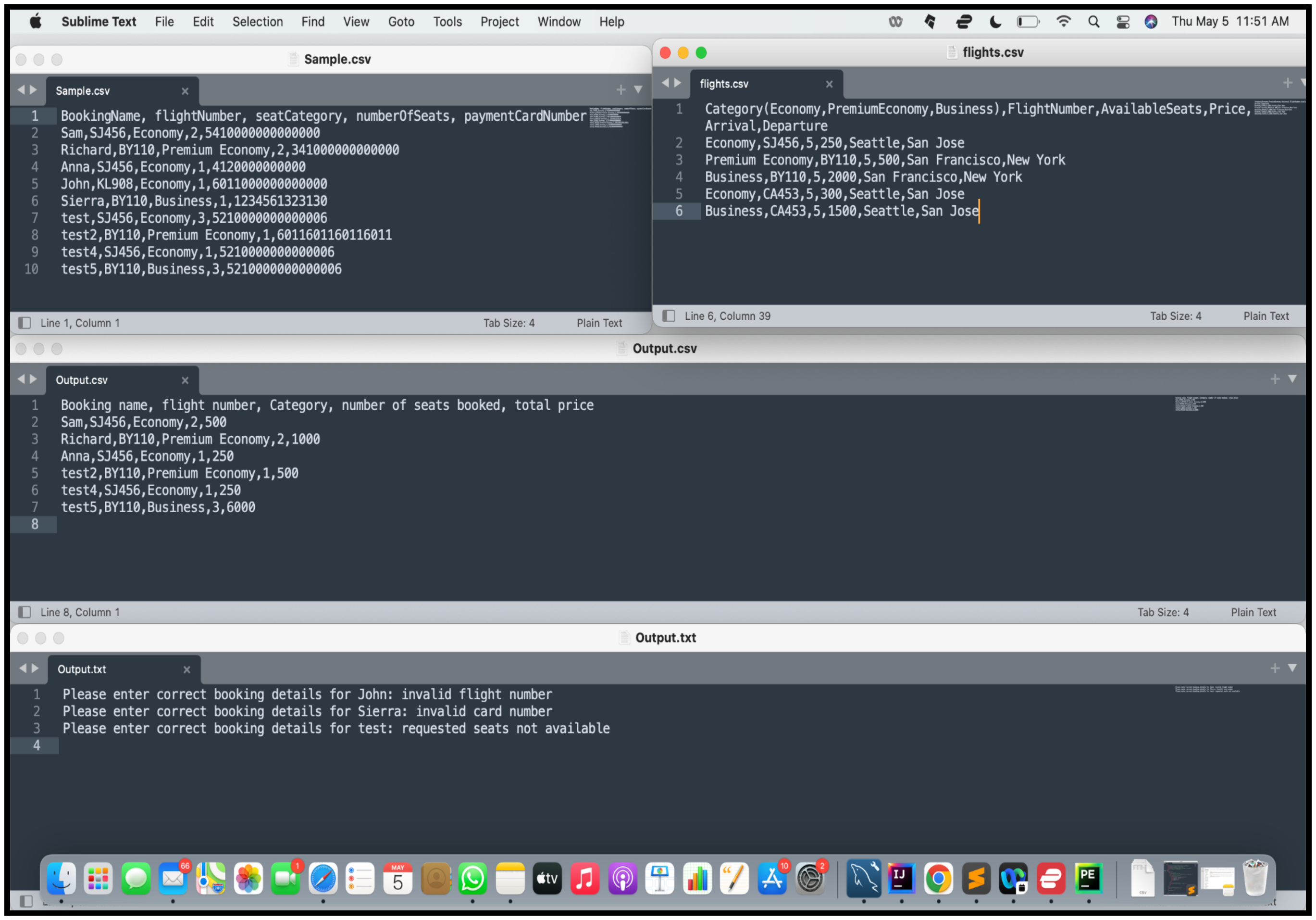
Run: BookingOperationTest

```
✓ Tests passed: 19 of 19 tests – 35 ms

✓ Test Results                                                                          35 ms
  ✓ BookingOperationTest                                                                35 ms
    ✓ Card Validation - valid inputs                                                    23 ms
    ✓ Testing Addition/Updates of pricing info to Flight_price Hashmap                   2 ms
    ✓ Testing Addition/Updates of seats to Flight_available_seats HashMap                1 ms
    ✓ Check Flight Exist Status- returns Boolean True or False                           2 ms
    ✓ Card Validation - Invalid inputs                                                   1 ms
    ✓ Parse the Csv file and store in FlightMap Hashmap                                  1 ms
    ✓ SIngleton Test
    ✓ Booking file Parser
    ✓ Set the Current State to:Flight Exist,FlightNotExist,ValidBooking,ConfirmBooking and check   4 ms
    ✓ Flight Seat Availability                                                           1 ms
```

```
/Users/avinash/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Hc
Create Setup and Run all Test cases

Process finished with exit code 0
```

Tests passed: 19 (2 minutes ago)

SAMPLE INPUT/OUTPUT SNIPPET:



Sample.csv

```
BookingName, flightNumber, seatCategory, numberOfSeats, paymentCardNumber
Sam,SJ456,Economy,2,5410000000000000
Richard,BY110,Premium Economy,2,341000000000000
Anna,SJ456,Economy,1,4120000000000
John,KL908,Economy,1,6011000000000000
Sierra,BY110,Business,1,1234561323130
test,SJ456,Economy,3,5210000000000006
test2,BY110,Premium Economy,1,6011601160116011
test4,SJ456,Economy,1,5210000000000006
test5,BY110,Business,3,5210000000000006
```

flights.csv

```
Category(Economy,PremiumEconomy,Business),FlightNumber,AvailableSeats,Price,
Arrival,Departure
Economy,SJ456,5,250,Seattle,San Jose
Premium Economy,BY110,5,500,San Francisco,New York
Business,BY110,5,2000,San Francisco,New York
Economy,CA453,5,300,Seattle,San Jose
Business,CA453,5,1500,Seattle,San Jose
```

Output.csv

```
Booking name, flight number, Category, number of seats booked, total price
Sam,SJ456,Economy,2,500
Richard,BY110,Premium Economy,2,1000
Anna,SJ456,Economy,1,250
test2,BY110,Premium Economy,1,500
test4,SJ456,Economy,1,250
test5,BY110,Business,3,6000
```

Output.txt

```
Please enter correct booking details for John: invalid flight number
Please enter correct booking details for Sierra: invalid card number
Please enter correct booking details for test: requested seats not available
```