CMP5332 OBJECT ORIENTED PROGRAMMING IN JAVA

BIRMINGHAM CITY University

FLIGHT BOOKING SYSTEM

Submitted By

Nabin Oli

Student id: 23189629

Contents

Introduction 5
Entities/Classes involved 5
Main.java 5
Commands package 7
AddBooking.java 7
AddCustomer.java 9
AddFeedback.java 10
AddFlight.java 10
ApplyPromocode.java 12
CancelBooking.java 13
Command.java 14
DeleteCustomer.java 15
DeleteFlight.java 15
Help.java 16
ListCustomer.java 17
ListFlights.java 18
LoadGUI.java 19
ShowCustomer.java 19
ShowFlight.java 20
ShowFlights.java 21
UpdateBooking.java 22
VIPSeatAllocation.java 23
Models package 24
Booking.java 25
Customer.java 26
Feedback.java 27
Flight.java 27
FlightBookingSystem.java 28
Data 29
Bookings.txt 29
customers.txt 30
feedbacks.txt 30

Flights.txt 31
Data Manager 31
BookingDataManager.java 31
CustomerDataManager.java 32
DataManager.java 33
FeedbackDataManager.java 34
FlightBookingSystemData.java 36
FlightDataManager.java 37
Graphical User Interface(GUI) 39
Loading Screen 39
Landing Frame 39
AddCustomerWindow 40
DeleteCustomerWindow 40
View CustomerWindow 41
View Flights 41
AddFlightWindow 42
..... 42
Deleteflight 42
IssueBookingWindow 42
UpdateBookingWindow 43
Cancel Booking 43
AdminLoginExitWindow 43
Testing 44
Bookingtest.java 44
CustomerTest.java 44
FlightBookingSystemTest.java 45
FlightTest.java 45
Additional Features and Enhancement 45
VIP Seat Allocation 45
Apply Promocode 46
Feedback form 46
Conclusion 46

This page is intentionally left blank

Flight Booking System

Introduction

The file that you are reading in your screen is the report of Flight Booking System developed by me , Nabin Oli using Java. I have followed Object Oriented Paradigm using Java for making this program. This program utilizes several OOP concepts including interface, Abstraction, encapsulation, classes, object and more. Except OOP, it also has utilized other features like file handling, several data structures and other. For knowing about how I utilized these concepts for making Flight Booking System, You have to follow me on this report, I'll show you aesthetic GUI(Graphical User Interface) , code, and files, images.

Entities/Classes involved

For making this system possible, I have used five classes

1. Customer
2. Flight
3. Booking
4. Feedback
5. FlightBookingSystem

Customer class is the blueprint or template for generating object customers in this system. Whenever new data are given to the system, It first takes to command package where is addcustomer package which helps in making new object of customer, later this new object is stored in the FlightBookingSystem object. There is one FlightBookingSystem object storing/operating on information about Customer, Flight, Booking, Feedback and related to these classes.

This was the small story of my system. Lets start the report of my Flight Booking System in detail.

Main.java

As other program, My file also starts with Main.java. In general, it has three functions

```java
 */
public class Main {

    /**
     * The main method of the Flight Booking System.
     * It initializes the system, handles user input, and executes commands.
     *
     * @param args Command-line arguments (not used)
     * @throws IOException                If an I/O error occurs
     * @throws FlightBookingSystemException If there's an error related to the Flight Booking System
     */
    public static void main(String[] args) throws IOException, FlightBookingSystemException {

        // Load the Flight Booking System data
        FlightBookingSystem fbs = FlightBookingSystemData.load();

        // Create a BufferedReader object to read user input
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // Display welcome message and instructions
        System.out.println("Flight Booking System");
        System.out.println("Enter 'help' to see a list of available commands.");

        // Main loop for user interaction
        while (true) {
            System.out.print("> ");
            String line = br.readLine();
            if (line.equals("exit")) {
                break;
            }

            try {
                // Parse the user input and execute the corresponding command
                Command command = CommandParser.parse(line);
                command.execute(fbs);
            } catch (FlightBookingSystemException ex) {
                System.out.println(ex.getMessage());
            }
        }

        // Exit the program
        System.exit(0);
    }
```

code

1. Load the data for flight booking system which is previously stored in computer
2. Take inputs for which operation users wats to run
3. Initialize and run CommandParser which parses the command and validate which input is user giving

Output

```
Flight Booking System
Enter 'help' to see a list of available commands.
> help
Commands:
        listflights                             print all flights
        listcustomers                           print all customers
        addflight                               add a new flight
        addcustomer                             add a new customer
        showflight [flight id]                  show flight details
        showcustomer [customer id]              show customer details
        addbooking [customer id] [flight id]    add a new booking
        cancelbooking [customer id] [flight id] cancel a booking
        editbooking [booking id] [flight id]    update a booking
        showflights                                                     shows details of flight
        deleteflight [flightId]                         Delete flight
        deletecustomer  [CustomerId]                    Delete customer
```

When entered help as a input it showed me available options/Commands.

So, this is the place where you will go if you would like to run my program. As I was mentioning about Command package above, Now I'll shift my gear towards describing and touring the functionalities of my commands package one by one.

Commands package

Commnds package contains total of 19 commnds for Command Line Interface. Those commands are listed below:

1. AddBooking
2. AddCustomer
3. AddFeedback
4. AddFlight
5. ApplyPromocode
6. BookingWithName
7. CancelBooking
8. Command
9. DeleteCustomer
10. DeleteFlight
11. Help
12. ListCustomer
13. ListFlights
14. LoadGUI
15. ShowCustomer
16. ShowFlight
17. ShowFlights
18. UpdateBooking
19. VIPSeatAllocation

AddBooking.java

The AddBooking class in the flight booking system implements a command to add a new booking for a customer on a specified flight and date. When the command is executed, it retrieves the customer and flight details based on provided IDs and checks if the flight has available capacity. If the flight is full, an exception is thrown; otherwise, the booking is issued and the system data is updated accordingly. The class ensures the booking process adheres to system constraints and persists the changes by saving the updated data. This command encapsulates the logic necessary for adding bookings in a structured and error-checked manner. Additionally, the class provides clear documentation and an example usage scenario, highlighting how to create and execute the AddBooking command. By implementing the Command interface, it integrates seamlessly with the rest of the flight booking system's command architecture. This design promotes modularity and ease of maintenance, allowing for future extensions or modifications to the booking process. Furthermore, the use of exceptions for error handling ensures that users receive immediate feedback if their booking cannot be processed.

```java
public class AddBooking implements Command {

    private final int customerId;
    private final int outboundFlightId;
    private final LocalDate bookingDate;

    /**
     * Constructs an AddBooking command with the specified customer ID, outbound flight ID,
     * and booking date.
     *
     * @param customerId      The ID of the customer making the booking
     * @param outboundFlightId The ID of the outbound flight to be booked
     * @param bookingDate     The date on which the booking is made
     */
    public AddBooking(int customerId, int outboundFlightId, LocalDate bookingDate) {
        this.customerId = customerId;
        this.outboundFlightId = outboundFlightId;
        this.bookingDate = bookingDate;
    }

    /**
     * Executes the AddBooking command within the provided FlightBookingSystem instance.
     * Retrieves the customer and outbound flight based on IDs, checks if the outbound flight
     * has capacity for additional passengers, and issues the booking if possible.
     * If successful, the updated system data is stored using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the booking is to be added
     * @throws FlightBookingSystemException If the customer or flight does not exist, or if the flight is at full capacity
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        Customer customer = flightBookingSystem.getCustomerByID(this.customerId);
        Flight outboundFlight = flightBookingSystem.getFlightByID(this.outboundFlightId);
        LocalDate bookingDate = this.bookingDate;

        if (outboundFlight.getPassengerCount() >= outboundFlight.getCapacity()) {
            throw new FlightBookingSystemException("Flight is at full capacity. Cannot issue booking.");
        }

        flightBookingSystem.issueBooking(customer, outboundFlight, bookingDate);
        System.out.println("Booking added successfully.");
```


```java
            throw new FlightBookingSystemException("Flight is at full capacity. Cannot issue booking.");
        }

        flightBookingSystem.issueBooking(customer, outboundFlight, bookingDate);
        System.out.println("Booking added successfully.");
```


```java
        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

Output

```
> addbooking
Customer ID: 1
outbound Flight ID: 24
inbound Flight ID: 24
bookDate: Departure Date ("YYYY-MM-DD" format): 2004-12-12
Booking date must be in the future.
> addbooking
Customer ID: 1
outbound Flight ID: 24
inbound Flight ID: 24
bookDate: Departure Date ("YYYY-MM-DD" format): 2024-12-12
Booking added successfully.
>
```

AddCustomer.java

The AddCustomer class is a command within the flight booking system that facilitates adding a new customer. It implements the Command interface, requiring execution within a FlightBookingSystem instance. When executed, the command creates a new customer identified by their name, phone number, and email address. It assigns a unique ID to the customer by finding the maximum existing ID in the system and incrementing it. The new customer is then added to the system, and the updated data is stored using FlightBookingSystemData. This command ensures that new customers are integrated into the system seamlessly, and any issues during the process are handled with appropriate exceptions. The class's design promotes modularity and ease of use, allowing for straightforward integration and future expansion of customer management functionalities.

```java
public class AddCustomer implements Command {

    private final String name;
    private final String phone;
    private final String email;

    /**
     * Constructs an AddCustomer command with the specified name, phone number, and email address.
     *
     * @param name  The name of the customer
     * @param phone The phone number of the customer
     * @param email The email address of the customer
     */
    public AddCustomer(String name, String phone, String email) {
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    /**
     * Executes the AddCustomer command within the provided FlightBookingSystem instance.
     * Generates a unique ID for the new customer, creates a Customer object with the provided
     * name, phone number, email, and adds it to the system using flightBookingSystem.
     * The updated system data is then stored using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the customer is to be added
     * @throws FlightBookingSystemException If there is an issue adding the customer to the system
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        int maxId = 0;
        for (Customer customer : flightBookingSystem.getCustomers()) {
            if (customer.getId() > maxId) {
                maxId = customer.getId();
            }
        }

        Customer customer = new Customer(++maxId, name, phone, email, false,false);
        flightBookingSystem.addCustomer(customer);
        System.out.println("Customer #" + customer.getId() + " added.");

```

```java
        Customer customer = new Customer(++maxId, name, phone, email, false,false);
        flightBookingSystem.addCustomer(customer);
        System.out.println("Customer #" + customer.getId() + " added.");

        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> addcustomer
Customer name: Nabin Kripa Oli
Customer phone: 9809090
Customer email:
nabin@gmail.com
Customer #27 added.
```

AddFeedback.java

The AddFeedback class is a command designed to add feedback to a booking within the flight booking system. Implementing the Command interface, it operates within a FlightBookingSystem instance, where it associates feedback with a specific booking and customer ID, along with a message. Upon execution, the command adds the feedback to the system and updates the data storage using FlightBookingSystemData. This ensures that feedback is captured and persisted accurately, allowing the system to maintain up-to-date customer reviews and comments. The class's design provides a straightforward mechanism for integrating user feedback into the flight booking system, promoting better customer service and data management.

```java
public class AddFeedback implements Command {

    private final int bookingID;
    private final int customerID;
    private final String message;

    /**
     * Constructs an AddFeedback command with the specified booking ID, customer ID, and feedback message.
     *
     * @param bookingID The ID of the booking
     * @param customerID The ID of the customer providing feedback
     * @param message The feedback message
     */
    public AddFeedback(int bookingID, int customerID, String message) {
        this.bookingID = bookingID;
        this.customerID = customerID;
        this.message = message;
    }

    /**
     * Executes the AddFeedback command within the provided FlightBookingSystem instance.
     * Adds the feedback to the system using the specified booking ID, customer ID, and feedback message.
     * The updated system data is then stored using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the feedback is to be added
     * @throws FlightBookingSystemException If there is an issue adding the feedback to the system
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        flightBookingSystem.addFeedback(bookingID, customerID, message);
        System.out.println("Feedback added for booking #" + bookingID + " by customer #" + customerID + ".");

        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> givefeedback
Enter your feedback message: Worse service
Enter your booking ID: 12
Enter your flight ID: 12
Feedback added for booking #12 by customer #12.
```

AddFlight.java

The AddFlight class is a command designed to add a new flight to the flight booking system, implementing the Command interface and requiring execution within a FlightBookingSystem instance. The command is instantiated with detailed flight information, including flight number,

origin, destination, departure date, capacity, and price. When executed, it generates a unique ID for the flight, creates a Flight object with the provided details, and adds it to the system. The flight data is then stored using FlightBookingSystemData, ensuring the system is updated with the new flight. This class facilitates the addition of flights by managing ID generation and data storage, promoting efficient and accurate flight management within the system. It also includes example usage to illustrate how to create and execute the command, enhancing usability and clarity.

```java
public class AddFlight implements Command {

    private final String flightNumber;
    private final String origin;
    private final String destination;
    private final LocalDate departureDate;
    private final int capacity;
    private final double price;

    /**
     * Constructs an AddFlight command with the specified flight details.
     *
     * @param flightNumber   The flight number of the flight
     * @param origin         The origin of the flight
     * @param destination    The destination of the flight
     * @param departureDate  The departure date of the flight
     * @param capacity       The capacity of the flight (number of seats)
     * @param price          The price of the flight
     */
    public AddFlight(String flightNumber, String origin, String destination, LocalDate departureDate, int capacity, double price) {
        this.flightNumber = flightNumber;
        this.origin = origin;
        this.destination = destination;
        this.departureDate = departureDate;
        this.capacity = capacity;
        this.price = price;
    }

    /**
     * Executes the AddFlight command within the provided FlightBookingSystem instance.
     * Generates a unique ID for the new flight, creates a Flight object with the provided
     * flight details, adds it to the system using flightBookingSystem, and stores the updated
     * data using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the flight is to be added
```

```java
    /**
     * Executes the AddFlight command within the provided FlightBookingSystem instance.
     * Generates a unique ID for the new flight, creates a Flight object with the provided
     * flight details, adds it to the system using flightBookingSystem, and stores the updated
     * data using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the flight is to be added
     * @throws FlightBookingSystemException If there is an issue adding the flight to the system
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        int maxId = 0;
        if (flightBookingSystem.getFlights().size() > 0) {
            int lastIndex = flightBookingSystem.getFlights().size() - 1;
            maxId = flightBookingSystem.getFlights().get(lastIndex).getId();
        }

        Flight flight = new Flight(++maxId, flightNumber, origin, destination, departureDate, (int) price, capacity, false);
        flightBookingSystem.addFlight(flight);
        System.out.println("Flight #" + flight.getId() + " added.");

        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> addflight
Flight Number: 123
Origin: RKKM
Destination: MKKR
Enter Date (yyyy-MM-dd): 2027-12-12
Enter capacity: 120
Enter price: 22000
Flight #50 added.
>
```

ApplyPromocode.java

```java
 */
public class ApplyPromocode implements Command {

    private final int bookingID;
    private final String promocode;

    /**
     * Constructs an ApplyPromocode command with the specified booking ID and promotional code.
     *
     * @param bookingID The booking ID of the flight to apply the promocode
     * @param promocode The promotional code to apply
     */
    public ApplyPromocode(int bookingID, String promocode) {
        this.bookingID = bookingID;
        this.promocode = promocode;
    }

    /**
     * Executes the ApplyPromocode command within the provided FlightBookingSystem instance.
     * Retrieves the booking with the specified ID, applies the promotional code to its associated flight,
     * and stores the updated data using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the promocode is to be applied
     * @throws FlightBookingSystemException If there is an issue applying the promocode to the flight
     * @throws IOException                  If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        Booking booking = flightBookingSystem.getBookingByID(bookingID);
        if (booking == null) {
            throw new FlightBookingSystemException("Booking with ID " + bookingID + " does not exist.");
        }

        booking.applyPromocode(promocode);
        System.out.println("Promocode '" + promocode + "' applied to booking #" + bookingID);

        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

The ApplyPromocode class is a command designed to apply a promotional code to a flight booking within the flight booking system, implementing the Command interface. This command, instantiated with a booking ID and a promotional code, retrieves the specified booking, applies the promo code to its associated flight, and updates the system data using FlightBookingSystemData. Upon execution, it ensures that the promotional discount is correctly applied to the booking's flight details and then stores the updated system state. The class provides clear functionality for modifying bookings with promotions, enhancing the system's capability to manage discounts efficiently and accurately. The example usage illustrates how to create and execute the command, making it easy for users to integrate promotional code applications into their booking processes.

OUTPUT

```
Enter 'help' to see a list of available commands.
> applypromocode
Enter booking ID:
12
Enter promocode:
nabinOpensFlightCompany20
Applying the promocode....
Promocode 'nabinOpensFlightCompany20' applied to booking #12
>
```

CancelBooking.java

The CancelBooking class is a command designed to cancel a specific booking for a customer on a specified flight within the flight booking system, implementing the Command interface. It requires execution within a FlightBookingSystem instance, where it uses the cancelBooking method to remove the booking based on the provided customer ID and flight ID. Upon execution, the system updates the booking data to reflect the cancellation and then stores the modified data using FlightBookingSystemData. This class facilitates the orderly cancellation of bookings, ensuring the system's state remains accurate and up-to-date. Example usage demonstrates how to create and execute the command, making it straightforward for users to manage booking cancellations within the system.

```java
public class CancelBooking implements Command {

    private final int customerId;
    private final int flightId;

    /**
     * Constructs a CancelBooking command with the specified customer ID and flight ID.
     *
     * @param customerId The ID of the customer whose booking is to be cancelled
     * @param flightId   The ID of the flight for which the booking is to be cancelled
     */
    public CancelBooking(int customerId, int flightId) {
        this.customerId = customerId;
        this.flightId = flightId;
    }

    /**
     * Executes the CancelBooking command within the provided FlightBookingSystem instance.
     * Cancels the booking of the specified customer on the specified flight by invoking
     * the cancelBooking method on flightBookingSystem. After cancellation, the updated booking
     * data is stored using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the booking cancellation is to be executed
     * @throws FlightBookingSystemException If there is an issue cancelling the booking in the system
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        flightBookingSystem.cancelBooking(customerId, flightId);
        System.out.println("Booking has been cancelled successfully.");
        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> cancelbooking
Enter customer id:
12
Enter flight id:
24
Canceling the booking....
Booking has been cancelled successfully.
>
```

Command.java

```java
 */
public interface Command {

    /**
     * A help message that provides a summary of available commands and their descriptions.
     * This message is intended to guide users on how to interact with the flight booking system.
     */
    public static final String HELP_MESSAGE = "Commands:\n"
        + "\tlistflights                                print all flights\n"
        + "\tlistcustomers                               print all customers\n"
        + "\taddflight                                   add a new flight\n"
        + "\taddcustomer                                 add a new customer\n"
        + "\tshowflight [flight id]                      show flight details\n"
        + "\tshowcustomer [customer id]                  show customer details\n"
        + "\taddbooking [customer id] [flight id]       add a new booking\n"
        + "\tcancelbooking [customer id] [flight id]    cancel a booking\n"
        + "\teditbooking [booking id] [flight id]       update a booking\n"
        + "\tshowflights                                 shows details of flight\n"
        + "\tdeleteflight [flightId]                     Delete flight\n"
        + "\tdeletecustomer [CustomerId]                 Delete customer\n"
        + "\tloadgui                                     loads the GUI version of the app\n"
        + "\thelp                                        prints this help message\n"
        + "\tbookwithname                                Book your flight with destination\n"
        + "\tgivefeedback                                Give feedback\n"
        + "\tallocatevipseat                             Allocate VIP or emergency seat"
        + "\texit                                        exits the program";

    /**
     * Executes the command within the provided FlightBookingSystem instance.
     * Implementing classes define specific operations to be performed within the system,
     * such as adding a flight, cancelling a booking, etc.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is to be executed
     * @throws FlightBookingSystemException If there is an issue executing the command in the system
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException;

}
```

The Command interface defines a blueprint for commands that can be executed within a flight booking system, encapsulating various operations such as adding flights, managing bookings, and displaying customer details. Implementations of this interface must provide an execute method, which performs the specific action within the context of a FlightBookingSystem instance and can throw FlightBookingSystemException and IOException for execution and data storage issues, respectively. The interface also includes a HELP_MESSAGE constant, summarizing all available commands and their functionalities to guide users in interacting with the system. Example usage illustrates how a command, like AddFlight, can be instantiated and executed to modify the system's state, enhancing the system's functionality and usability.

DeleteCustomer.java

The DeleteCustomer class implements the Command interface and is designed to delete a customer from the flight booking system by marking them as deleted, rather than physically removing them from the system. Upon execution, the command identifies the customer by their ID, sets the 'deleted' flag to true, and updates the system's data. If the customer ID does not exist, a FlightBookingSystemException is thrown. After marking the customer as deleted, the system's data is stored using FlightBookingSystemData. This approach ensures that the customer's data remains accessible for historical purposes, while indicating their deletion status. Example usage demonstrates creating and executing the command to remove a customer from the system.

```java
public class DeleteCustomer implements Command {

    private final int customerID;

    /**
     * Constructs a DeleteCustomer command object with the specified customer ID.
     *
     * @param customerID The ID of the customer to be deleted
     */
    public DeleteCustomer(int customerID) {
        this.customerID = customerID;
    }

    /**
     * Executes the delete customer command within the provided FlightBookingSystem instance.
     * Marks the customer with the specified ID as deleted by setting the 'deleted' flag.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an issue with deleting the customer (e.g., customer not found)
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        Customer customerToDelete = flightBookingSystem.getCustomerByID(this.customerID);
        if (customerToDelete == null) {
            throw new FlightBookingSystemException("Customer with ID " + customerID + " not found.");
        }

        customerToDelete.setDeleted();  // Soft delete by setting the flag
        System.out.println("Customer #" + customerToDelete.getId() + " marked as deleted.");
        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> deletecustomer
Enter customer id: 12
Customer #12 marked as deleted.
>
```

DeleteFlight.java

The DeleteFlight class is a command that removes a flight from the flight booking system. It implements the Command interface, enabling its execution within a FlightBookingSystem instance. When executed, the command searches for the flight by its flight number and, if found, removes it from the system, then updates the data store via FlightBookingSystemData. If the flight does not exist in the system, a FlightBookingSystemException is thrown. The command also prints a confirmation message indicating the deletion of the flight. Example usage demonstrates how to create and execute this command to remove a flight identified by its flight number.

```java
 * @see Command
 */
public class DeleteFlight implements Command {

    private final int flightNumber;

    /**
     * Constructs a DeleteFlight command object with the specified flight number.
     *
     * @param flightNumber The number of the flight to be deleted
     */
    public DeleteFlight(int flightNumber) {
        this.flightNumber = flightNumber;
    }

    /**
     * Executes the delete flight command within the provided FlightBookingSystem instance.
     * Removes the flight with the specified flight number from the system.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an issue with deleting the flight (e.g., flight not found)
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        Flight flightToDelete = flightBookingSystem.getFlightByID(this.flightNumber);
        if (flightToDelete == null) {
            throw new FlightBookingSystemException("Flight with number " + flightNumber + " not found.");
        }

        flightBookingSystem.removeFlight(flightToDelete);
        System.out.println("Flight #" + flightToDelete.getId() + " deleted.");
        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> listflights
Flight #24 - FL024 - RUKUM to LUKUM on 12/12/2024
Flight #41 - 200 - DDL to LDD on 12/12/2024
Flight #42 - 2004 - KMT to RMK on 09/09/2024
Flight #44 - 44 - MTV to BKP on 06/06/2025
Flight #45 - 2022 - LMK to KML on 01/12/2024
Flight #46 - 12 - KTM to MKT on 12/12/2025
Flight #47 - 1222 - KTR to RTK on 12/12/2029
Flight #48 - 1200 - KTTTM to RKKKM on 12/12/2025
Flight #49 - 1400 - TMK to MTK on 12/12/2029
Flight #50 - 123 - RKKM to MKKR on 12/12/2027
10 flight(s)
> deleteflight
Enter flight ID: 41
Flight #41 deleted.
>
```

Help.java

The Help class is a command designed to display the help message containing a list of available commands and their descriptions within the flight booking system. It implements the Command interface, allowing it to be executed within the context of a FlightBookingSystem instance. When the execute method is called, it prints the predefined HELP_MESSAGE from the Command interface to the console, providing users with guidance on how to interact with the system. Example usage illustrates creating and executing the command to view the help message, assisting users in navigating and utilizing the flight booking system's features effectively.

```java
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.model.FlightBookingSystem;

/**
 * The Help class represents a command to print the help message of available commands in the flight booking system.
 * It implements the Command interface, allowing it to be executed within the context of a FlightBookingSystem instance.
 *
 * <p>Upon execution, the command prints out the predefined HELP_MESSAGE from the Command interface.
 * This message includes a list of available commands and their descriptions for user reference.
 *
 * <p>Example usage:
 * <pre>{@code
 * // Create a new Help command instance
 * Command helpCommand = new Help();
 *
 * // Execute the command within a FlightBookingSystem instance
 * helpCommand.execute(flightBookingSystem);
 * }</pre>
 *
 * @see Command
 */
public class Help implements Command {

    /**
     * Executes the help command within the provided FlightBookingSystem instance.
     * Prints the predefined HELP_MESSAGE to the console.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) {
        System.out.println(Command.HELP_MESSAGE);
    }
}
```

ListCustomer.java

The ListCustomer class is a command designed to list all customers within the flight booking system. It implements the Command interface, making it executable within a FlightBookingSystem instance. Upon execution, the command retrieves all customers from the system and iterates through them, printing short details of each customer to the console, while excluding those marked as deleted. It also outputs the total count of customers listed. This functionality helps users to view all active customer details in the system. Example usage shows how to instantiate and execute the command to display customer information.

```java
public class ListCustomer implements Command {

    /**
     * Executes the list customer command within the provided FlightBookingSystem instance.
     * Retrieves a list of all customers and prints their short details to the console.
     * Excludes customers marked as deleted.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an error while accessing or processing flight booking system data
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Customer> customers = flightBookingSystem.getCustomer();
        for (Customer customer : customers) {
            if (!customer.getDeleted()) {
                System.out.println(customer.getDetailsShort());
            }
        }
        System.out.println(customers.size() + " customer(s)");
    }
}
```

OUTPUT

```
Enter 'help' to see a list of available commands.
> listcustomers
Customer #1 - Nabin Oli - 980000
Customer #2 - Nabraj Kripa - 9800010
Customer #3 - Nabu Khadka - 980800
Customer #4 - Rabu Oli - 9809990
Customer #5 - Rina Shah - 980002
Customer #6 - Rajan Singh - 980003
Customer #7 - Sita Thapa - 980004
Customer #9 - Shyam Acharya - 980006
Customer #10 - Gita Lama - 980007
Customer #11 - Bibek Sharma - 980008
Customer #14 - Saraswoti Pokhrel - 980011
Customer #15 - Bikash Adhikari - 980012
Customer #16 - Shova Shrestha - 980013
Customer #17 - Prakash Maharjan - 980014
Customer #18 - Nisha Tamang - 980015
Customer #19 - Dinesh Shrestha - 980016
Customer #20 - Kamal Pandey - 980017
Customer #21 - Sunita Bhandari - 980018
```

ListFlights.java

The ListFlights class is a command designed to list all flights in the flight booking system, implementing the Command interface to be executable within a FlightBookingSystem instance. Upon execution, this command retrieves all flights from the system, iterates through them, and prints the short details of each flight to the console, excluding those marked as deleted. It also prints the total number of flights listed, providing a summary of all active flights. Example usage illustrates how to instantiate and execute the command to display flight information. This functionality ensures users can view all available flight details in the system.

```java
public class ListFlights implements Command {

    /**
     * Executes the list flights command within the provided FlightBookingSystem instance.
     * Retrieves a list of all flights and prints their short details to the console.
     * Excludes flights marked as deleted.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an error while accessing or processing flight booking system data
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Flight> flights = flightBookingSystem.getFlights();
        for (Flight flight : flights) {
            if (!flight.getDeleteStatusFlight()) {
                System.out.println(flight.getDetailsShort());
            }
        }
        System.out.println(flights.size() + " flight(s)");
    }
}
```

OUTPUT

```
> listflights
Flight #24 - FL024 - RUKUM to LUKUM on 12/12/2024
Flight #41 - 200 - DDL to LDD on 12/12/2024
Flight #42 - 2004 - KMT to RMK on 09/09/2024
Flight #44 - 44 - MTV to BKP on 06/06/2025
Flight #45 - 2022 - LMK to KML on 01/12/2024
Flight #46 - 12 - KTM to MKT on 12/12/2025
Flight #47 - 1222 - KTR to RTK on 12/12/2029
Flight #48 - 1200 - KTTTM to RKKKM on 12/12/2025
Flight #49 - 1400 - TMK to MTK on 12/12/2029
9 flight(s)
```

LoadGUI.java

The LoadGUI class is designed to display the graphical user interface (GUI) for the Flight Booking System, implementing the Command interface to execute within a FlightBookingSystem instance. Upon execution, this command sets up a JFrame with a white background and specific layout to accommodate GUI elements. It includes loading images from specified paths and placing them into JLabels. A JProgressBar is also created to show loading progress, updating incrementally via a Timer. The command organizes components into two panels, panelNorth and panelSouth, and uses a Timer to simulate the loading process, updating the progress bar at intervals. Once the progress reaches 100%, the JFrame is disposed, and a new MainWindow is instantiated to continue with the main application. Example usage demonstrates how to instantiate and execute this command to show the loading interface before the main application window appears.

ShowCustomer.java

The ShowCustomer class is a command that retrieves and displays the details of a specific customer from the Flight Booking System, implementing the Command interface to function within a FlightBookingSystem instance. Upon execution, the command takes the customer ID provided during its construction and uses the getCustomerByID method of the FlightBookingSystem to retrieve the relevant customer object. It then calls the showDetails method on this customer object to display their detailed information. This command is straightforward, designed to provide a detailed view of a customer's information based on their ID. Example usage demonstrates how to create and execute this command to view customer details within the system.

```java
 */
public class ShowCustomer implements Command {

    private int id;

    /**
     * Constructs a ShowCustomer command object with the specified customer ID.
     *
     * @param id The ID of the customer whose details are to be displayed
     */
    public ShowCustomer(int id) {
        this.id = id;
    }

    /**
     * Executes the show customer details command within the provided FlightBookingSystem instance.
     * Retrieves the customer information by their ID and displays their details using the showDetails() method.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an error while accessing or processing flight booking system data
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        Customer customer = flightBookingSystem.getCustomerByID(id);
        customer.showDetails();
    }
}
```

OUTPUT

```
Enter 'help' to see a list of available commands.
> showcustomer
Enter customer id: 12
Customer ID: 12
Name: Anita Maharjan
Phone Number: 980009
Bookings:
  Booking ID: 12
  Outbound Flight: FL024 From: RUKUM To: LUKUM Date: 2024-12-12 Price: 400.0
>
```

ShowFlight.java

```java
public class ShowFlight implements Command {

    private int id;

    /**
     * Constructs a ShowFlight command object with the specified flight ID.
     *
     * @param id The ID of the flight whose details are to be displayed
     */
    public ShowFlight(int id) {
        this.id = id;
    }

    /**
     * Executes the show flight details command within the provided FlightBookingSystem instance.
     * Retrieves the flight information by its ID and prints its long details using the getDetailsLong() method.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an error while accessing or processing flight booking system data
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        Flight flight = flightBookingSystem.getFlightByID(id);
        System.out.println(flight.getDetailsLong());
    }
}
```

The ShowFlight class is designed to retrieve and display detailed information about a specific flight within the Flight Booking System, implementing the Command interface to be executed in the context of a FlightBookingSystem instance. When executed, the command takes the flight ID provided during its construction and uses the getFlightByID method of the FlightBookingSystem to fetch the flight object. It then prints the long details of the flight using the getDetailsLong method from the Flight class. This command is useful for displaying comprehensive information about a flight, making it easier to view detailed flight specifics based on its ID. Example usage shows how to instantiate and run this command to view flight details.

OUTPUT

```
> showflight
Enter flight no. :41
Flight ID: 41
Flight Number: 200
Origin: DDL
Destination: LDD
Departure Date: 12/12/2024
Passenger Details:

>
```

ShowFlights.java

The ShowFlights class is a command designed to display a list of flights with their brief details from the Flight Booking System. By implementing the Command interface, it is able to be executed within the context of a FlightBookingSystem instance. When executed, the command retrieves all flights using the getFlights() method of the FlightBookingSystem instance. It iterates through each flight, checking if the flight is not marked as deleted via the getDeleteStatusFlight() method. For each flight that is not deleted, it prints the flight's short details using the getDetailsShort() method. After listing all the non-deleted flights, the command prints the total number of flights displayed, providing a comprehensive overview of the current flight status in the system. Example usage demonstrates how to create and execute this command to view flight information.

```java
 * @see Flight
 */
public class ShowFlights implements Command {

    /**
     * Executes the show flights command within the provided FlightBookingSystem instance.
     * Retrieves the list of flights, iterates through each flight, prints its short details if not deleted,
     * and then prints the total number of flights displayed.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If there is an error while accessing or processing flight booking system data
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Flight> flights = flightBookingSystem.getFlights();
        for (Flight flight : flights) {
            if (!flight.getDeleteStatusFlight()) {
                System.out.println(flight.getDetailsShort());
            }
        }
        System.out.println(flights.size() + " flight(s)");
    }
}

```

OUTPUT

```
> showflights
Flight #24 - FL024 - RUKUM to LUKUM on 12/12/2024
Flight #42 - 2004 - KMT to RMK on 09/09/2024
Flight #44 - 44 - MTV to BKP on 06/06/2025
Flight #45 - 2022 - LMK to KML on 01/12/2024
Flight #46 - 12 - KTM to MKT on 12/12/2025
Flight #47 - 1222 - KTR to RTK on 12/12/2029
Flight #48 - 1200 - KTTTM to RKKKM on 12/12/2025
Flight #49 - 1400 - TMK to MTK on 12/12/2029
Flight #50 - 123 - RKKM to MKKR on 12/12/2027
10 flight(s)
>
```

UpdateBooking.java

The UpdateBooking class is a command designed to update the flight number and booking date of an existing booking within a Flight Booking System. By implementing the Command interface, it is executable within a FlightBookingSystem instance. Upon execution, the command retrieves the booking using its ID with the getBookingByID() method. If the booking is not found, a FlightBookingSystemException is thrown. If the booking exists, the command updates its flight number and booking date to the new values specified during the object's construction. The class constructor requires the booking ID, the new flight number, and the new booking date. Example usage demonstrates how to create and execute this command to update booking details. This command is essential for maintaining accurate and current booking information within the system.

```java
     */
    public UpdateBooking(int bookingID, String newFlightNumber, LocalDate newBookDate) {
        this.bookingID = bookingID;
        this.newFlightNumber = newFlightNumber;
        this.newBookDate = newBookDate;
    }

    /**
     * Executes the update booking command within the provided FlightBookingSystem instance.
     * Retrieves the booking by its ID, updates its flight number and booking date to the new values,
     * and throws an exception if the booking is not found.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the command is executed
     * @throws FlightBookingSystemException If the booking with the specified ID is not found
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        Booking booking = flightBookingSystem.getBookingByID(bookingID);
        if (booking == null) {
            throw new FlightBookingSystemException("Booking not found.");
        }

        booking.setFlightNumber(newFlightNumber);
        booking.setBookingDate(newBookDate);
    }
}

```

OUTPUT

```
Enter 'help' to see a list of available commands.
> editbooking
Enter bookingID:
12
Enter newFlightNumber:
41
Enter new booking date (YYYY-MM-DD):
2024-12-12
Updated Successfully!
>
```

VIPSeatAllocation.java

The VIPSeatAllocation class is a command designed to allocate a VIP seat to a customer on a flight, even if the flight is fully booked. If the flight's capacity is reached, this command will cancel a regular booking to make space for the VIP booking. The constructor initializes the command with the customer's ID, flight ID, booking ID, and booking date. When executed, the command retrieves the customer, flight, and booking details using their respective IDs. It checks the flight's current passenger count against its capacity. If the flight is full and the specified booking is not a VIP booking, it cancels the regular booking. It then issues the VIP booking and stores the updated flight booking system data using FlightBookingSystemData.store(). The command throws exceptions if any of the entities do not exist or if there is an issue during the booking process. The example usage illustrates how to create and execute this command to allocate a VIP seat.

```java
    public VIPSeatAllocation(int customerId, int flightId, LocalDate bookingDate, int bookingid) {
        this.customerId = customerId;
        this.flightId = flightId;
        this.bookingDate = bookingDate;
        this.bookingid = bookingid;
    }

    /**
     * Executes the VIPSeatAllocation command within the provided FlightBookingSystem instance.
     * Retrieves the customer, flight, and booking based on IDs, checks if the flight has capacity,
     * and if not, cancels the specified regular booking to make room for the VIP booking.
     * If successful, the updated system data is stored using FlightBookingSystemData.
     *
     * @param flightBookingSystem The FlightBookingSystem instance on which the VIP booking is to be added
     * @throws FlightBookingSystemException If the customer, flight, or booking does not exist, or if there is an error during the process
     * @throws IOException If there is an error storing data using FlightBookingSystemData
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException, IOException {
        Customer customer = flightBookingSystem.getCustomerByID(this.customerId);
        Flight flight = flightBookingSystem.getFlightByID(this.flightId);
        Booking bookingToDisplace = flightBookingSystem.getBookingByID(this.bookingid);

        if (flight.getPassengerCount() >= flight.getCapacity()) {
            // Flight is full, cancel the specified regular booking
            if (bookingToDisplace == null || bookingToDisplace.getCustomer().isVIP()) {
                throw new FlightBookingSystemException("Invalid booking ID provided or booking is not eligible for displacement.");
            }

            // Cancel the regular booking
            Command cancelBookingCommand = new CancelBooking(bookingToDisplace.getCustomer().getId(), flightId);
            cancelBookingCommand.execute(flightBookingSystem);
        }

        // Issue the VIP booking
        flightBookingSystem.issueBooking(customer, flight, bookingDate);
        System.out.println("VIP booking added successfully.");

        FlightBookingSystemData.store(flightBookingSystem);
    }
}
```

OUTPUT

```
> allocatevipseat
Enter customerId:
12
Enter flightId:
41
Enter bookingId:
21
VIPSeatAllocation object created:
```

Models package

This package consists of the classes used in the Flight booking system which are also given and explain below:

Booking.java

```java
/**
 * Represents a booking made by a customer for one or more flights.
 * A booking can have an outbound flight and optionally a return flight.
 */
public class Booking {

    private Customer customer; // The customer who made the booking
    private Flight outboundFlight; // The outbound flight of the booking
    private Flight returnFlight; // The return flight of the booking (optional)
    private LocalDate bookingDate; // The date when the booking was made
    private double price; // The price of the booking
    private boolean cancelled; // Indicates whether the booking is cancelled or not

    /**
     * Constructs a new Booking object with only an outbound flight.
     *
     * @param customer       The customer making the booking
     * @param outboundFlight The outbound flight to be booked
     */
    public Booking(Customer customer, Flight outboundFlight) {
        this.customer = customer;
        this.outboundFlight = outboundFlight;
        this.bookingDate = outboundFlight.getDepartureDate();
        this.cancelled = false; // By default, booking is not cancelled
        this.price = outboundFlight.getPrice();
    }

    /**
     * Constructs a new Booking object with both outbound and return flights.
     *
     * @param customer       The customer making the booking
     * @param outboundFlight The outbound flight to be booked
     * @param returnFlight   The return flight to be booked
     */
    public Booking(Customer customer, Flight outboundFlight, Flight returnFlight) {
        this.customer = customer;
        this.outboundFlight = outboundFlight;
        this.returnFlight = returnFlight;
        this.bookingDate = outboundFlight.getDepartureDate();
```

Important code section(constructor) of Booking class

The Booking class models a booking within a flight booking system, encapsulating details such as the customer, flight(s), booking date, price, and cancellation status. The class provides two constructors: one for creating a booking with only an outbound flight, and another for bookings that include both outbound and return flights. This class has getter and setter methods to access and modify booking details such as the customer, flights, booking date, and price. It also includes methods to check the cancellation status and to cancel the booking, applying a cancellation fee if the booking is canceled. The class ensures that the price can be adjusted, including the ability to apply a promotional code that offers a discount, specifically a 20% reduction for the promo code "nabinOpensFlightCompany20".

Additionally, the Booking class includes utility methods such as getId() which generates an ID based on the customer's ID, and toString() for providing a string representation of the booking object. The class also supports modifying the flight number of the outbound flight and maintains the booking's overall details in a structured format. This design allows for flexible and comprehensive management of bookings, accommodating both outbound and return flights and facilitating operations such as booking modification, cancellation, and price adjustments. The

class is designed to integrate smoothly with other components of the flight booking system, ensuring consistency and ease of use in managing booking details.

Customer.java

The Customer class models a customer in the flight booking system, containing personal details such as ID, name, phone number, email, and a list of bookings. It includes flags to indicate if the customer is marked as deleted or is a VIP. The class provides methods to retrieve and set these details, as well as to manage the list of bookings, including adding a new booking and retrieving a copy of the bookings list. It also includes utility methods for obtaining short and detailed string representations of the customer's details and bookings. For instance, showDetails() prints detailed information about the customer's bookings to the console, while getShowDetails() returns a formatted string of all details, making it useful for generating reports or displaying customer information in a user interface.

```java
public class Customer {

    private int id; // The unique identifier of the customer
    private String name; // The name of the customer
    private String phone; // The phone number of the customer
    private String email; // The email address of the customer
    private final List<Booking> bookings = new ArrayList<>(); // The list of bookings made by the customer
    private boolean isDeleted = false; // Indicates whether the customer is deleted from the system
    private boolean isVIP = false;

    /**
     * Constructs a new Customer object with the specified attributes.
     *
     * @param id        The unique identifier of the customer
     * @param name      The name of the customer
     * @param phone     The phone number of the customer
     * @param email     The email address of the customer
     * @param isDeleted Indicates whether the customer is deleted from the system
     */
    public Customer(int id, String name, String phone, String email, boolean isDeleted,boolean isVIP) {
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.isDeleted = isDeleted;
        this.isVIP = isVIP;
    }

    /**
     * Retrieves the unique identifier of the customer.
     *
     * @return The unique identifier of the customer
     */
    public int getId() {
        return id;
    }
```

The Customer class ensures encapsulation and proper management of customer data, offering functionality to mark the customer as deleted and to check if the customer is VIP. This design supports the operational needs of the booking system by allowing easy tracking of customer information, booking details, and status flags, facilitating efficient data handling and user interaction within the system.

Feedback.java

```java
public class Feedback {
    private static int lastFeedbackID = 0;

    private final int id;           // The unique identifier for the feedback
    private final int bookingID;    // The ID of the booking associated with the feedback
    private final int customerID;   // The ID of the customer providing the feedback
    private final String message;   // The feedback message provided by the customer

    /**
     * Constructs a new Feedback object with the specified booking ID, customer ID, and message.
     * The feedback ID is automatically assigned and incremented with each new feedback.
     *
     * @param bookingID The ID of the booking related to the feedback
     * @param customerID The ID of the customer providing the feedback
     * @param message The feedback message provided by the customer
     */
    public Feedback(int bookingID, int customerID, String message) {
        this.id = ++lastFeedbackID;
        this.bookingID = bookingID;
        this.customerID = customerID;
        this.message = message;
    }

    /**
     * Retrieves the unique identifier of the feedback.
     *
     * @return The ID of the feedback
     */
    public int getId() {
        return id;
    }

    /**
     * Retrieves the ID of the booking associated with the feedback.
```

The Feedback class represents customer feedback related to a booking in a flight booking system. Each feedback entry is uniquely identified by an ID, which is automatically incremented with each new instance, ensuring uniqueness. The class includes attributes for the feedback ID, the associated booking ID, the customer ID, and the feedback message. The constructor initializes these attributes, setting the feedback ID to the next available value and assigning the provided booking and customer IDs along with the feedback message. Getter methods are provided to retrieve the values of these attributes, enabling access to the feedback details within the system. This class is essential for capturing and managing customer feedback to improve the service quality.

Flight.java

The Flight class models a flight within a flight booking system, encapsulating attributes such as a unique ID, flight number, origin, destination, departure date, seating capacity, and price per seat. It also manages a set of passengers, ensuring that the flight can accommodate only up to its full capacity and tracking whether it has been marked as deleted. The class provides methods to add and remove passengers, check flight deletion status, and retrieve detailed information about the flight and its passengers. Additionally, it includes methods to calculate the flight price based on the days remaining until departure and the current capacity, and to check if the flight has already departed. The class is designed to handle both short and detailed representations of flight information, facilitating easy access and display of flight data.

```java
public class Flight {

    private int id; // The unique identifier of the flight
    private String flightNumber; // The flight number
    private String origin; // The origin of the flight
    private String destination; // The destination of the flight
    private LocalDate departureDate; // The departure date of the flight
    private int capacity; // The number of seats available on the flight
    private double price; // The price per seat of the flight
    private final Set<Customer> passengers; // The set of passengers booked on the flight
    private boolean isDeleted = false; // Indicates whether the flight is deleted from the system

    /**
     * Constructs a new Flight object with the specified attributes.
     *
     * @param id            The unique identifier of the flight
     * @param flightNumber  The flight number
     * @param origin        The origin of the flight
     * @param destination   The destination of the flight
     * @param departureDate The departure date of the flight
     * @param capacity      The number of seats available on the flight
     * @param price         The price per seat of the flight
     * @param isDeleted     Indicates whether the flight is deleted from the system
     */
    public Flight(int id, String flightNumber, String origin, String destination, LocalDate departureDate, int capacity, double price, boolean isDelet
        this.id = id;
        this.flightNumber = flightNumber;
        this.origin = origin;
        this.destination = destination;
        this.departureDate = departureDate;
        this.capacity = capacity;
        this.price = price;
        this.isDeleted = isDeleted;
        passengers = new HashSet<>();
    }
```

FlightBookingSystem.java

The FlightBookingSystem class is designed to manage the operations of a flight booking system, encompassing a range of functionalities to handle customers, flights, bookings, and feedback. It maintains the current system date and collections for customers, flights, bookings, and feedback. Key methods include adding and retrieving customers and flights, issuing and canceling bookings, and retrieving specific bookings by ID. The system ensures that flights have a maximum capacity and that bookings are handled only if the flight is not full. Additionally, it allows feedback to be added for bookings and retrieves all feedback given by customers.

```java
public class FlightBookingSystem {

    private final LocalDate systemDate = LocalDate.now(); // The current date of the system

    private final Map<Integer, Customer> customers = new TreeMap<>(); // Collection of customers in the system
    private final Map<Integer, Flight> flights = new TreeMap<>(); // Collection of flights in the system
    private final List<Booking> bookings = new ArrayList<>(); // Collection of bookings in the system
    private final List<Feedback> feedbacks = new ArrayList<>();
    /**
     * Retrieves the current date of the system.
     *
     * @return The current date of the system
     */
    public LocalDate getSystemDate() {
        return systemDate;
    }

    /**
     * Adds a booking to the system.
     *
     * @param booking The booking to be added to the system
     */
    public void addBooking(Booking booking) {
        bookings.add(booking);
    }

    /**
     * Retrieves a list of all future flights in the system that have not departed.
     *
     * @param systemDate The current date of the system
     * @return A list of all future flights in the system that have not departed
     */
    public List<Flight> getFutureFlights(LocalDate systemDate) {
        List<Flight> futureFlights = new ArrayList<>();
        for (Flight flight : flights.values()) {
            if (!flight.getDepartureDate().isBefore(systemDate) && !flight.hasDeparted(systemDate)) {
```

The class also includes methods to handle specific customer queries by name, managing scenarios where multiple customers may share the same name by prompting the user to select the correct customer via email. Furthermore, it includes functionality to fetch future flights, both to display available options to users and to manage flight operations effectively. Error handling is integrated throughout, throwing exceptions if attempts are made to add duplicate flights or customers, or if bookings are attempted on non-existent flights or customers. This design ensures robustness and user-friendly management of the flight booking system.

Data

This package consists of data handled in this software. The glimpse of data from each class are given below:

Bookings.txt

```
2::2::47::2029-12-12::
5::5::24::2024-12-12::
10::10::24::2024-12-12::
12::12::24::2024-12-12::
13::13::24::2024-12-12::
13::13::45::2024-12-01::
13::13::46::2025-12-12::
14::14::24::2024-12-12::
18::18::45::2024-12-01::
18::18::45::2024-12-01::
19::19::45::2024-12-01::
19::19::45::2024-12-01::
20::20::45::2024-12-01::
21::21::47::2029-12-12::
23::23::47::2029-12-12::
24::24::47::2029-12-12::
24::24::49::2029-12-12::
```

customers.txt

```
1::Nabin Oli::980000::nabin@gmail.com::false::false::
2::Nabraj Kripa::9800010::n@gmail.com::false::false::
3::Nabu Khadka::980800::na@gmail.com::false::false::
4::Rabu Oli::9809990::ba@gmail.com::false::false::
5::Rina Shah::980002::rinashah@gmail.com::false::false::
6::Rajan Singh::980003::rajan@gmail.com::false::false::
7::Sita Thapa::980004::sita.thapa@gmail.com::false::false::
8::Ram Karki::980005::ramkarki@gmail.com::true::false::
9::Shyam Acharya::980006::shyam.acharya@gmail.com::false::false::
10::Gita Lama::980007::gita.lama@gmail.com::false::false::
11::Bibek Sharma::980008::bibek.sharma@gmail.com::false::false::
12::Anita Maharjan::980009::anita.maharjan@gmail.com::true::false::
13::Ramesh Chhetri::980010::ramesh.chhetri@gmail.com::true::false::
14::Saraswoti Pokhrel::980011::saraswoti.pokhrel@gmail.com::false::false::
15::Bikash Adhikari::980012::bikash.adhikari@gmail.com::false::false::
16::Shova Shrestha::980013::shova.shrestha@gmail.com::false::false::
17::Prakash Maharjan::980014::prakash.maharjan@gmail.com::false::false::
18::Nisha Tamang::980015::nisha.tamang@gmail.com::false::false::
19::Dinesh Shrestha::980016::dinesh.shrestha@gmail.com::false::false::
20::Kamal Pandey::980017::kamal.pandey@gmail.com::false::false::
21::Sunita Bhandari::980018::sunita.bhandari@gmail.com::false::false::
22::Roshan Khatri::980019::roshan.khatri@gmail.com::false::false::
23::Sarita Sharma::980020::sarita.sharma@gmail.com::false::false::
24::Nabin Olivia::9800000::oliv@gmail.com::false::false::
25::Nabin Oli::9848185797::nabinoli2004@gmail.com::false::false::
26::Rameshwor Yadav::9848000000::nab@gmail.com::true::false::
27
```

feedbacks.txt

```
12::12::Poor::
2
```

Flights.txt

```
24::FL024::RUKUM::LUKUM::2024-12-12::300::400.0::false::
41::200::DDL::LDD::2024-12-12::34000::1200.0::false::
42::2004::KMT::RMK::2024-09-09::20000::200.0::false::
44::44::MTV::BKP::2025-06-06::50::49.0::false::
45::2022::LMK::KML::2024-12-01::13::49.0::false::
46::12::KTM::MKT::2025-12-12::1::51.0::false::
47::1222::KTR::RTK::2029-12-12::4::2.0::false::
48::1200::KTTTM::RKKKM::2025-12-12::12000::122.0::false::
49::1400::TMK::MTK::2029-12-12::1220::1200.0::false::
```

Data Manager

There is a package named Data which is all about managing(loading and storing data). It consists of

BookingDataManager.java

The BookingDataManager class is designed to manage the loading and storing of booking data within a flight booking system, implementing the DataManager interface. This class handles the interaction with a text file specified by the RESOURCE constant, where booking records are stored and retrieved. Each booking record is formatted using the SEPARATOR constant ("::") to distinguish between different fields. The loadData method reads from the file line by line, parsing each line to extract booking details such as booking ID, customer ID, and flight ID. It adds the parsed bookings to the FlightBookingSystem instance, ensuring that the appropriate customers and flights are linked to each booking. This method also includes error handling for parsing errors, throwing exceptions if there are issues with data format or file reading.

```java
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(RESOURCE))){
            int line_idx = 1;
            while(sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                try {
                    int id = Integer.parseInt(properties[0]);
                    int customerId = Integer.parseInt(properties[1]);
                    int flightId = Integer.parseInt(properties[2]);

                    Customer customer = fbs.getCustomerByID(customerId);
                    Flight flight = fbs.getFlightByID(flightId);
                    Booking booking = new Booking(customer, flight);
                    fbs.addBooking(booking);
                    flight.addPassenger(customer);
                    customer.addBooking(booking);

                } catch (NumberFormatException | DateTimeParseException ex) {
                    throw new FlightBookingSystemException("Unable to parse booking on line " + line_idx
                            + "\nError: " + ex);
                }
                line_idx++;
            }
        }
    }
```

The storeData method writes the current bookings from the FlightBookingSystem instance back to the text file. It formats each booking's details into a string, separating fields with the defined SEPARATOR, and writes them line by line to the file. This method ensures that all bookings are accurately recorded and updated in the file, ready for future loading. Both methods are wrapped in try-with-resources statements to manage file resources efficiently, ensuring that files are properly closed after operations. The class provides a clear interface for persisting and retrieving booking data, crucial for maintaining the integrity and continuity of the flight booking system's state across sessions.

```java
    }

    /**
     * Stores current booking data from the provided FlightBookingSystem instance into the bookings data file.
     * Each booking's details are formatted and written to the file, separated by the SEPARATOR.
     *
     * @param fbs The FlightBookingSystem instance to store booking data from
     * @throws IOException If there is an error writing data to the file
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(RESOURCE))) {
            for (Booking booking : fbs.getBookings() ) {
                out.print(booking.getId() + SEPARATOR);
                out.print(booking.getCustomer().getId() + SEPARATOR);
                out.print(booking.getOutboundFlight().getId() + SEPARATOR);
                out.print(booking.getBookingDate() + SEPARATOR);
                out.println();
            }
        }
    }
}
```

CustomerDataManager.java

The CustomerDataManager class implements the DataManager interface to facilitate the loading and storing of customer data between a text file and a FlightBookingSystem instance. This class uses the RESOURCE constant to locate the text file, where each customer record is stored with fields separated by the SEPARATOR ("::"). The loadData method reads from the file, parsing each line to extract customer details such as ID, name, phone, email, deletion status, and VIP status, and then adds these customers to the FlightBookingSystem. Error handling is included to manage parsing issues, throwing exceptions if the data format is incorrect. The storeData method retrieves all customers from the FlightBookingSystem, formats their details into a string, and writes them to the file. Both methods ensure data integrity and continuity between the file and the in-memory system state, supporting essential operations like adding and updating customer records.

```java
public class CustomerDataManager implements DataManager {

    /**
     * The path to the customers data file.
     */
    private final String RESOURCE = "./resources/data/customers.txt";

    /**
     * Retrieves the path to the customers data file.
     *
     * @return The path to the customers data file
     */
    protected String getResourcePath() {
        return RESOURCE;
    }

    /**
     * Loads existing customer data from the customers data file into the provided FlightBookingSystem instance.
     * Each line in the file represents a customer record with fields separated by the SEPARATOR.
     *
     * @param fbs The FlightBookingSystem instance to load customer data into
     * @throws IOException If there is an error reading the data file
     * @throws FlightBookingSystemException If there is an error parsing the customer data
     */
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(getResourcePath()))){
            int line_idx = 1;
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                try {
                    int id = Integer.parseInt(properties[0]);
                    String name = properties[1];
                    String phone = properties[2];
                    String email = properties[3];
                    // Parse boolean value from string
                    boolean isDeleted = Boolean.parseBoolean(properties[4]);
                    boolean isVip = Boolean.parseBoolean(properties[5]);
                    Customer customer = new Customer(id, name, phone, email, isDeleted,isVip);
                    fbs.addCustomer(customer);
                } catch (NumberFormatException ex) {
```

```java
            }
        }
    }

    /**
     * Stores current customer data from the provided FlightBookingSystem instance into the customers data file.
     * Each customer's details are formatted and written to the file, separated by the SEPARATOR.
     *
     * @param fbs The FlightBookingSystem instance to store customer data from
     * @throws IOException If there is an error writing data to the file
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(getResourcePath()))) {
            for (Customer customer : fbs.getCustomers()) {
                out.print(customer.getId() + SEPARATOR);
                out.print(customer.getName() + SEPARATOR);
                out.print(customer.getPhone() + SEPARATOR);
                out.print(customer.getEmail() + SEPARATOR);
                out.print(customer.getDeleted() + SEPARATOR);
                out.print(customer.isVIP() + SEPARATOR);
                out.println();
            }
        }
    }
}
```

DataManager.java

The DataManager interface defines a blueprint for managing data operations between a data source and a FlightBookingSystem instance. It specifies two primary methods: loadData and storeData. The SEPARATOR constant, set to "::", is used as a delimiter to separate fields in the data

source, ensuring consistency in data formatting. The loadData method is intended for reading data from a source, parsing it, and populating the FlightBookingSystem with this data. Implementing classes are expected to handle any exceptions related to reading and parsing data, encapsulating logic specific to the data source format. Conversely, the storeData method is designed to retrieve data from the FlightBookingSystem, format it correctly, and write it to the specified data source. Implementers must handle any exceptions that arise during the writing process. This interface provides a clear contract for data management, allowing for flexibility in how different data sources are handled while maintaining a consistent approach to data loading and storage.

```java
 */
public interface DataManager {

    /**
     * The delimiter used to separate fields in the data source.
     */
    public static final String SEPARATOR = "::";

    /**
     * Loads data from a data source into the provided FlightBookingSystem instance.
     * Implementing classes should define how data is read from the data source,
     * parsed, and added to the FlightBookingSystem.
     *
     * @param fbs The FlightBookingSystem instance to load data into
     * @throws IOException If there is an error reading the data source
     * @throws FlightBookingSystemException If there is an error parsing the data or updating the FlightBookingSystem
     */
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException;

    /**
     * Stores current data from the provided FlightBookingSystem instance
     * into a data source. Implementing classes should define how data is
     * retrieved from the FlightBookingSystem and formatted for storage.
     *
     * @param fbs The FlightBookingSystem instance to store data from
     * @throws IOException If there is an error writing data to the data source
     */
    public void storeData(FlightBookingSystem fbs) throws IOException;
}

```

FeedbackDataManager.java

The FeedbackDataManager class implements the DataManager interface to handle the loading and storing of feedback data for a FlightBookingSystem instance. It manages feedback records stored in a text file specified by the RESOURCE constant. The loadData method reads each line from the feedback file, splits the line into fields using the SEPARATOR, and creates Feedback objects which are then added to the FlightBookingSystem. It handles parsing errors by throwing FlightBookingSystemException if the data format is incorrect. The storeData method writes the feedback data from the FlightBookingSystem to the file, formatting each feedback record with the specified separator. This class ensures that feedback data is consistently read from and written to the file, facilitating persistent storage and retrieval of feedback information for the booking system.

```java
public class FeedbackDataManager implements DataManager {

    private final String RESOURCE = "./resources/data/feedbacks.txt";
    private final String SEPARATOR = "::";

    /**
     * Retrieves the path to the feedback data file.
     *
     * @return The path to the feedback data file
     */
    protected String getResourcePath() {
        return RESOURCE;
    }

    /**
     * Loads existing feedback data from the feedback data file into the provided FlightBookingSystem instance.
     * Each line in the file represents a feedback record with fields separated by the SEPARATOR.
     *
     * @param fbs The FlightBookingSystem instance to load feedback data into
     * @throws IOException If there is an error reading the data file
     * @throws FlightBookingSystemException If there is an error parsing the feedback data
     */
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(getResourcePath()))) {
            int lineIdx = 1;
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                try {
                    int bookingId = Integer.parseInt(properties[0]);
                    int customerId = Integer.parseInt(properties[1]);
                    String message = properties[2];
                    Feedback feedback = new Feedback(bookingId, customerId, message);
                    fbs.addFeedback(feedback);
                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse feedback on line " + lineIdx + "\nError: " + ex);
                }
                lineIdx++;
```


```java
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(getResourcePath()))) {
            int lineIdx = 1;
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                try {
                    int bookingId = Integer.parseInt(properties[0]);
                    int customerId = Integer.parseInt(properties[1]);
                    String message = properties[2];
                    Feedback feedback = new Feedback(bookingId, customerId, message);
                    fbs.addFeedback(feedback);
                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse feedback on line " + lineIdx + "\nError: " + ex);
                }
                lineIdx++;
            }
        }
    }
```


```java
    /**
     * Stores current feedback data from the provided FlightBookingSystem instance into the feedback data file.
     * Each feedback's details are formatted and written to the file, separated by the SEPARATOR.
     *
     * @param fbs The FlightBookingSystem instance to store feedback data from
     * @throws IOException If there is an error writing data to the file
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(getResourcePath()))) {
            for (Feedback feedback : fbs.getFeedbacks()) {
                out.print(feedback.getBookingID() + SEPARATOR);
                out.print(feedback.getCustomerID() + SEPARATOR);
                out.print(feedback.getMessage() + SEPARATOR);
                out.println();
            }
        }
    }
}
```

FlightBookingSystemData.java

The FlightBookingSystemData class manages the loading and storing of data for the FlightBookingSystem by coordinating multiple DataManager instances. This class uses a static list to hold references to various data managers such as FlightDataManager, CustomerDataManager, BookingDataManager, and FeedbackDataManager. Upon loading, the class initializes these data managers, and the load method creates a new FlightBookingSystem instance, populating it by invoking the loadData method on each registered data manager. Conversely, the store method saves the current state of the FlightBookingSystem by calling the storeData method on all registered data managers. This design allows for modular and organized handling of different aspects of the flight booking system's data, although the loadData and storeData methods for CustomerDataManager and BookingDataManager need to be implemented.

```java
public class FlightBookingSystemData {

    private static final List<DataManager> dataManagers = new ArrayList<>();

    // runs only once when the object gets loaded to memory
    static {
        dataManagers.add(new FlightDataManager());

        /* Uncomment the two lines below when the implementation of their
        loadData() and storeData() methods is complete */
         dataManagers.add(new CustomerDataManager());
         dataManagers.add(new BookingDataManager());
         dataManagers.add(new FeedbackDataManager());
    }

    /**
     * Loads the flight booking system data.
     *
     * <p>This method creates a new instance of {@link FlightBookingSystem} and
     * populates it with data from all registered {@link DataManager} instances.
     *
     * @return a fully populated {@link FlightBookingSystem} instance
     * @throws FlightBookingSystemException if there is an issue with loading the data
     * @throws IOException if there is an I/O error during the loading process
     */
    public static FlightBookingSystem load() throws FlightBookingSystemException, IOException {

        FlightBookingSystem fbs = new FlightBookingSystem();
        for (DataManager dm : dataManagers) {
            dm.loadData(fbs);
        }
        return fbs;
    }

    /**
     * Stores the flight booking system data.
     *
     * <p>This method takes a {@link FlightBookingSystem} instance and saves its data
     * using all registered {@link DataManager} instances.
     *
```

```java
    public static FlightBookingSystem load() throws FlightBookingSystemException, IOException {

        FlightBookingSystem fbs = new FlightBookingSystem();
        for (DataManager dm : dataManagers) {
            dm.loadData(fbs);
        }
        return fbs;
    }

    /**
     * Stores the flight booking system data.
     *
     * <p>This method takes a {@link FlightBookingSystem} instance and saves its data
     * using all registered {@link DataManager} instances.
     *
     * @param fbs the {@link FlightBookingSystem} instance to store
     * @throws IOException if there is an I/O error during the storing process
     */
    public static void store(FlightBookingSystem fbs) throws IOException {
        for (DataManager dm : dataManagers) {
            dm.storeData(fbs);
        }
    }

}
```

FlightDataManager.java

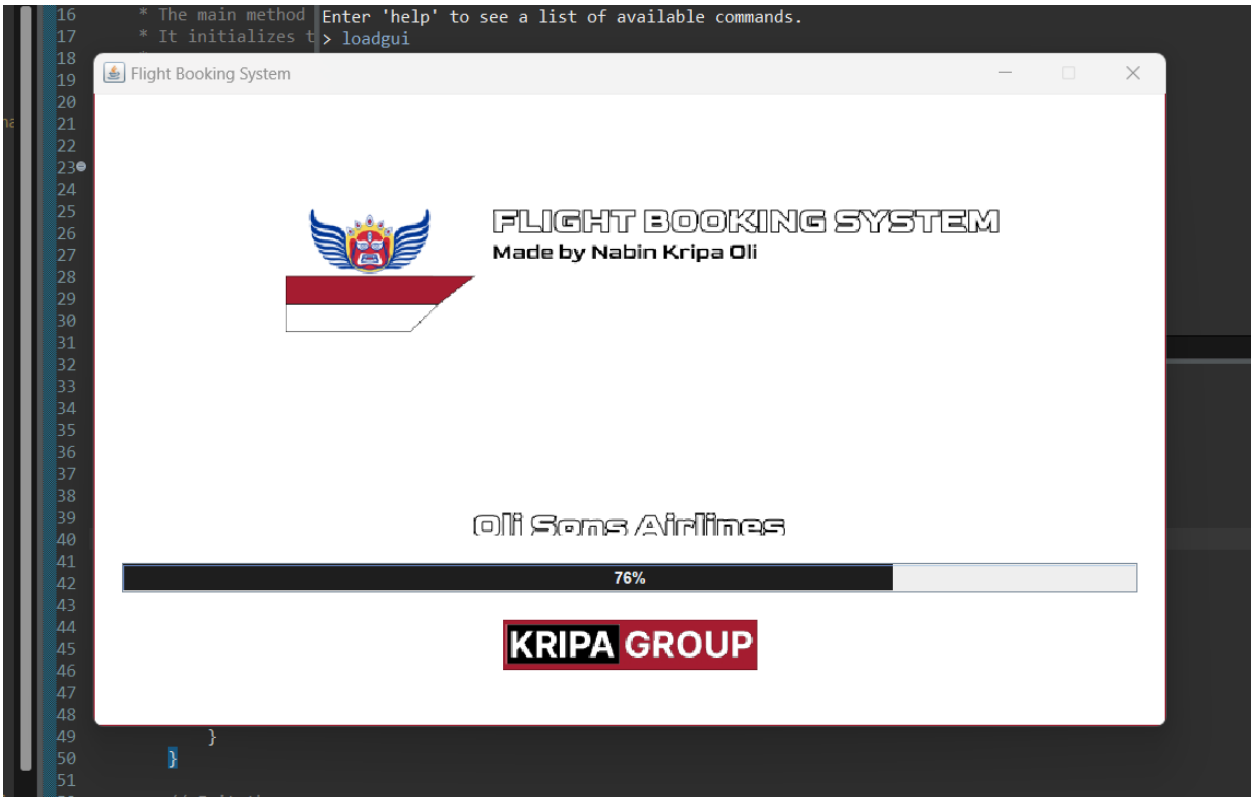
The FlightDataManager class is designed to handle the loading and storing of flight data for the flight booking system. It implements the DataManager interface and operates on a text file located at ./resources/data/flights.txt. The loadData method reads each line from the file, splits the line into properties using a separator, and parses these properties into a Flight object, which is then added to the FlightBookingSystem instance. It handles errors in parsing by throwing a FlightBookingSystemException if any data format issues arise. The storeData method iterates over all flights in the FlightBookingSystem, formats their details into a string, and writes each flight's information to the file, separated by a defined separator. This class ensures that flight data is accurately persisted and loaded from the file system, maintaining the integrity and state of the flight booking system.

```java
public class FlightDataManager implements DataManager {

    /** The path to the resource file containing flight data. */
    final static String RESOURCE = "./resources/data/flights.txt";

    /**
     * Loads the flight data from the resource file and populates the
     * {@link FlightBookingSystem} with flight information.
     *
     * @param fbs the {@link FlightBookingSystem} to be populated with flight data
     * @throws IOException if there is an I/O error during the loading process
     * @throws FlightBookingSystemException if there is an error in the flight data format
     */
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(RESOURCE))) {
            int line_idx = 1;
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                try {
                    int id = Integer.parseInt(properties[0]);
                    String flightNumber = properties[1];
                    String origin = properties[2];
                    String destination = properties[3];
                    LocalDate departureDate = LocalDate.parse(properties[4]);
                    int capacity = Integer.parseInt(properties[5]);
                    double price = Double.parseDouble(properties[6]);
                    boolean isDeleted = Boolean.parseBoolean(properties[7]);
                    Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, capacity, price, isDeleted);
                    fbs.addFlight(flight);
                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse flight id " + properties[0] + " on line " + line_idx
                        + "\nError: " + ex);
                }
                line_idx++;
            }
        }
    }

    /**
     * Stores the flight data from the {@link FlightBookingSystem} to the resource file.
     *
     * @param fbs the {@link FlightBookingSystem} containing flight data to be stored
     * @throws IOException if there is an I/O error during the storing process
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(RESOURCE))) {
            for (Flight flight : fbs.getFlights()) {
                out.print(flight.getId() + SEPARATOR);
                out.print(flight.getFlightNumber() + SEPARATOR);
                out.print(flight.getOrigin() + SEPARATOR);
                out.print(flight.getDestination() + SEPARATOR);
                out.print(flight.getDepartureDate() + SEPARATOR);
                out.print(flight.getCapacity() + SEPARATOR);
                out.print(flight.getPrice() + SEPARATOR);
                out.print(flight.getDeleteStatusFlight() + SEPARATOR);
                out.println();
            }
        }
    }
}
```

Graphical User Interface(GUI)

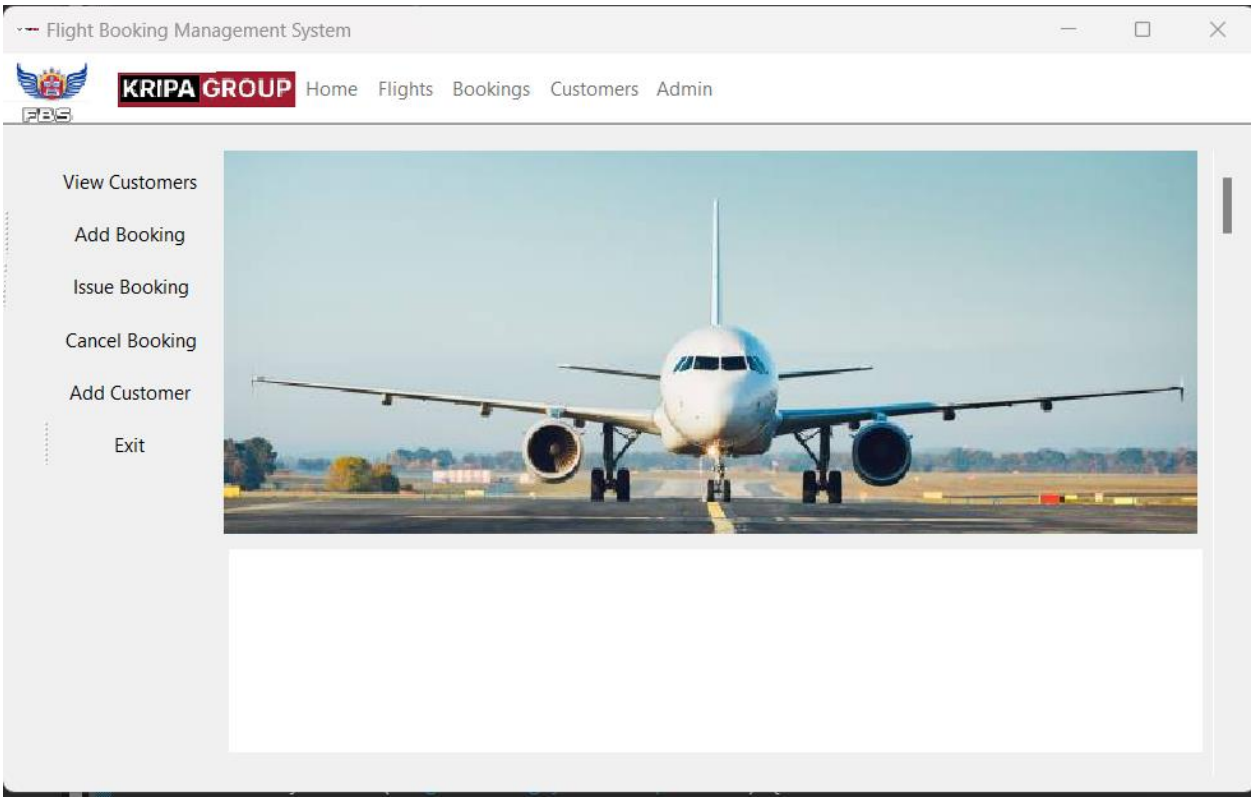
Loading Screen

I have designed and implemented a loading screen that prominently displays the company's brand, including its logo, allowing customers to see and engage with the company's branding while they wait. This feature enhances the user experience by reinforcing brand identity and providing a visually appealing waiting period.

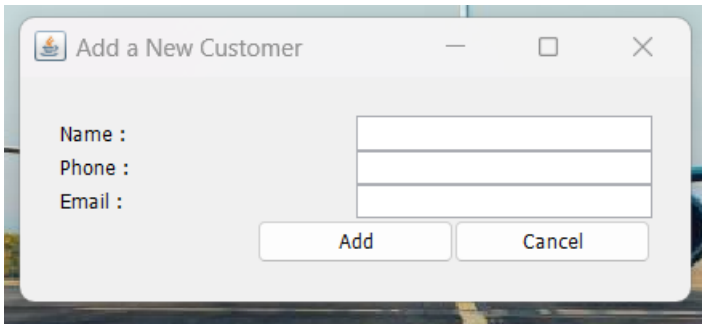


Landing Frame

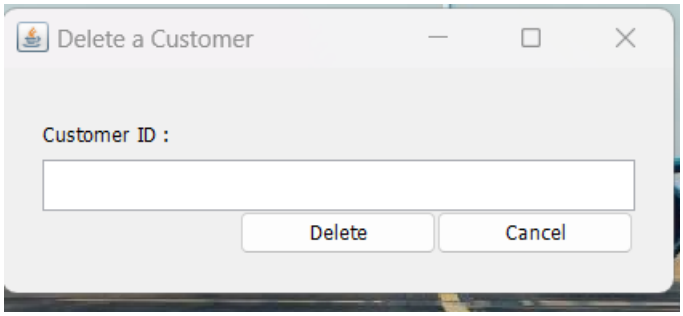
After loading is completed, Now is the time for displaying a landing frame, where you'll get to explore almost all the features of flight booking system. It has cool looking picture of aero plane which attracts the eyeballs of user, it also has logo of Brand on top of window, with logo of the airline company too. It also has menu bar which has commands for using Flight Booking System.



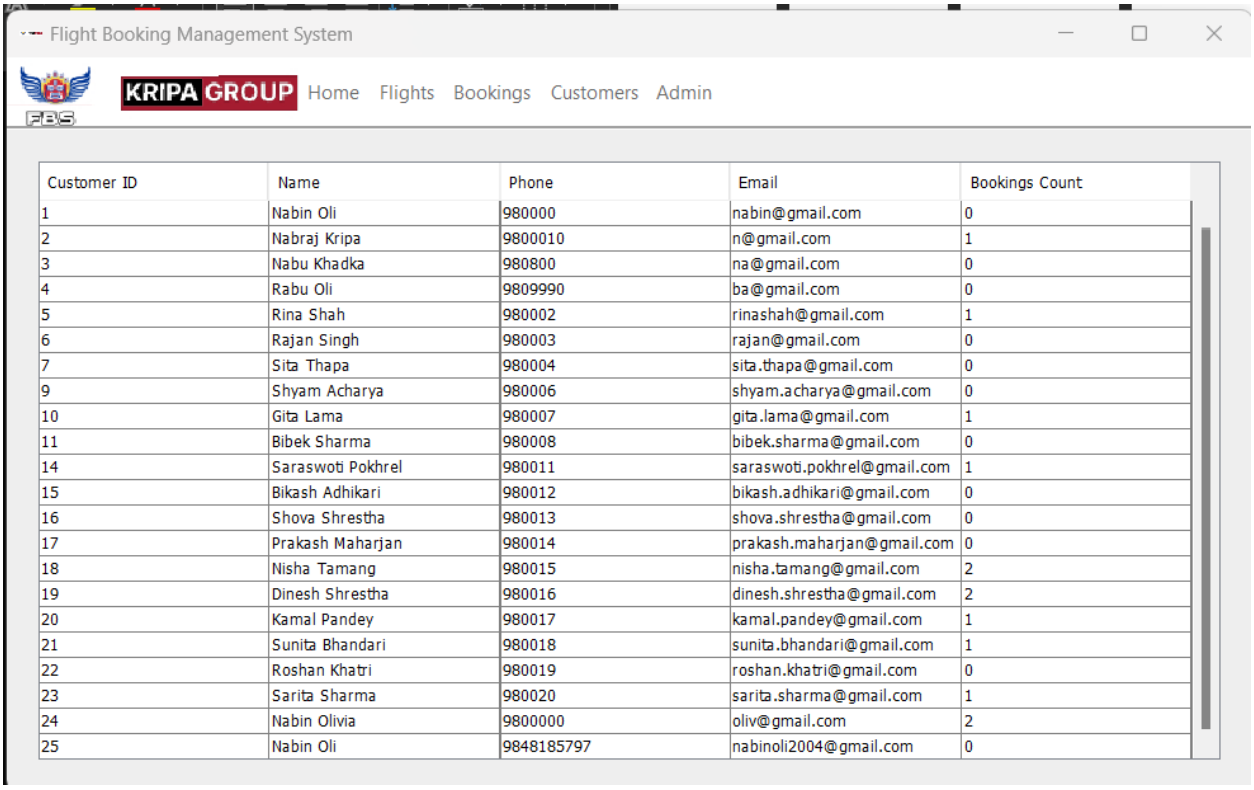
AddCustomerWindow



DeleteCustomerWindow

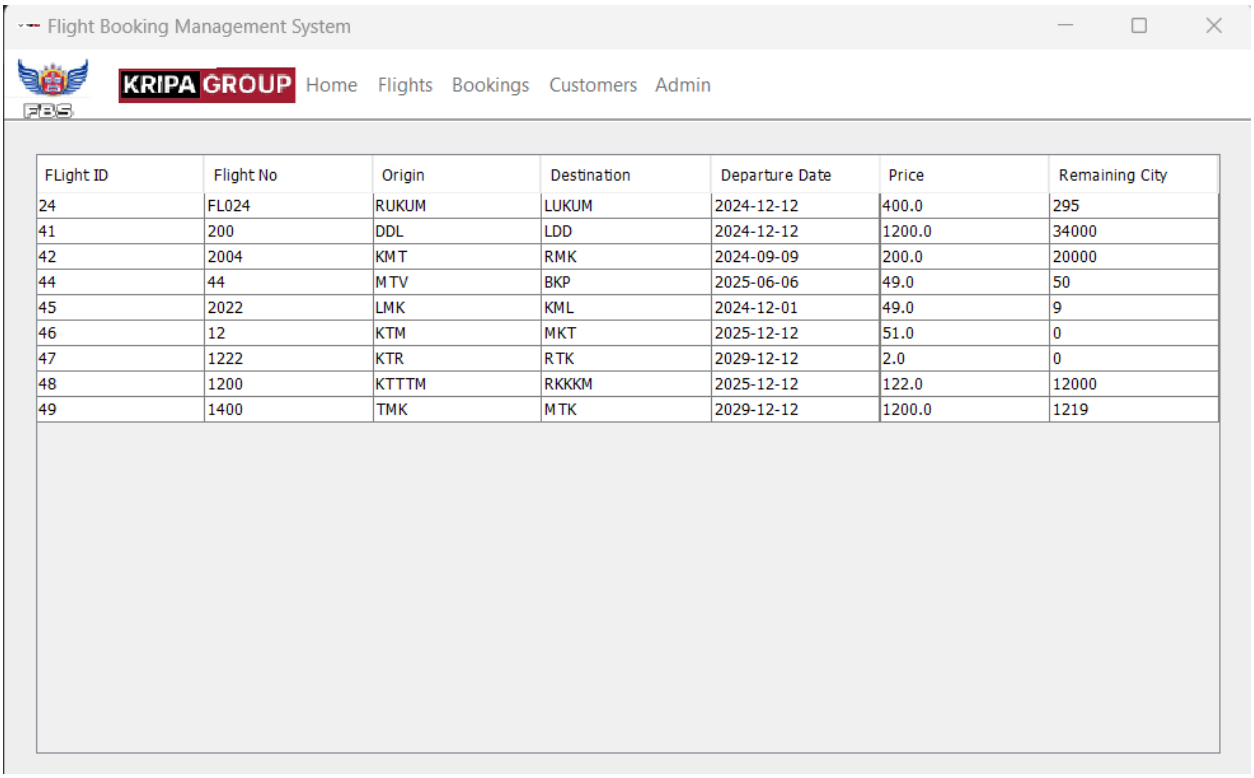


View CustomerWindow



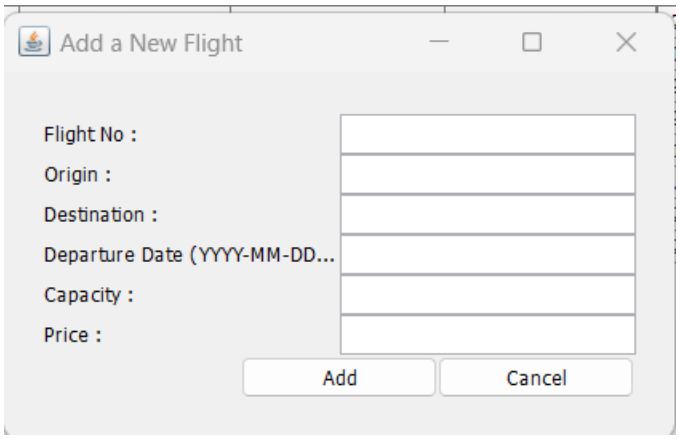
Customer ID	Name	Phone	Email	Bookings Count
1	Nabin Oli	980000	nabin@gmail.com	0
2	Nabraj Kripa	9800010	n@gmail.com	1
3	Nabu Khadka	980800	na@gmail.com	0
4	Rabu Oli	9809990	ba@gmail.com	0
5	Rina Shah	980002	rinashah@gmail.com	1
6	Rajan Singh	980003	rajan@gmail.com	0
7	Sita Thapa	980004	sita.thapa@gmail.com	0
9	Shyam Acharya	980006	shyam.acharya@gmail.com	0
10	Gita Lama	980007	gita.lama@gmail.com	1
11	Bibek Sharma	980008	bibek.sharma@gmail.com	0
14	Saraswoti Pokhrel	980011	saraswoti.pokhrel@gmail.com	1
15	Bikash Adhikari	980012	bikash.adhikari@gmail.com	0
16	Shova Shrestha	980013	shova.shrestha@gmail.com	0
17	Prakash Maharjan	980014	prakash.maharjan@gmail.com	0
18	Nisha Tamang	980015	nisha.tamang@gmail.com	2
19	Dinesh Shrestha	980016	dinesh.shrestha@gmail.com	2
20	Kamal Pandey	980017	kamal.pandey@gmail.com	1
21	Sunita Bhandari	980018	sunita.bhandari@gmail.com	1
22	Roshan Khatri	980019	roshan.khatri@gmail.com	0
23	Sarita Sharma	980020	sarita.sharma@gmail.com	1
24	Nabin Olivia	9800000	oliv@gmail.com	2
25	Nabin Oli	9848185797	nabinoli2004@gmail.com	0

View Flights

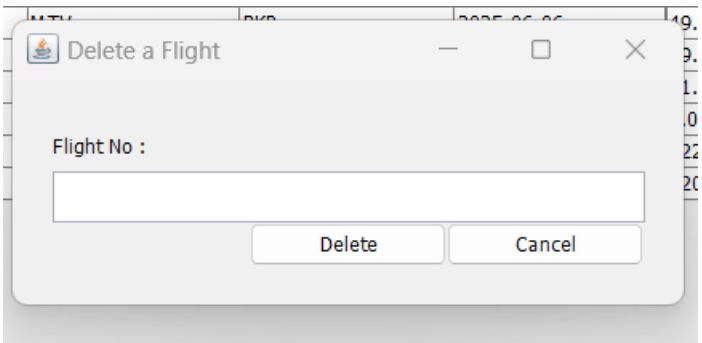


FLight ID	Flight No	Origin	Destination	Departure Date	Price	Remaining City
24	FL024	RUKUM	LUKUM	2024-12-12	400.0	295
41	200	DDL	LDD	2024-12-12	1200.0	34000
42	2004	KMT	RMK	2024-09-09	200.0	20000
44	44	MTV	BKP	2025-06-06	49.0	50
45	2022	LMK	KML	2024-12-01	49.0	9
46	12	KTM	MKT	2025-12-12	51.0	0
47	1222	KTR	RTK	2029-12-12	2.0	0
48	1200	KTTTM	RKKKM	2025-12-12	122.0	12000
49	1400	TMK	MTK	2029-12-12	1200.0	1219

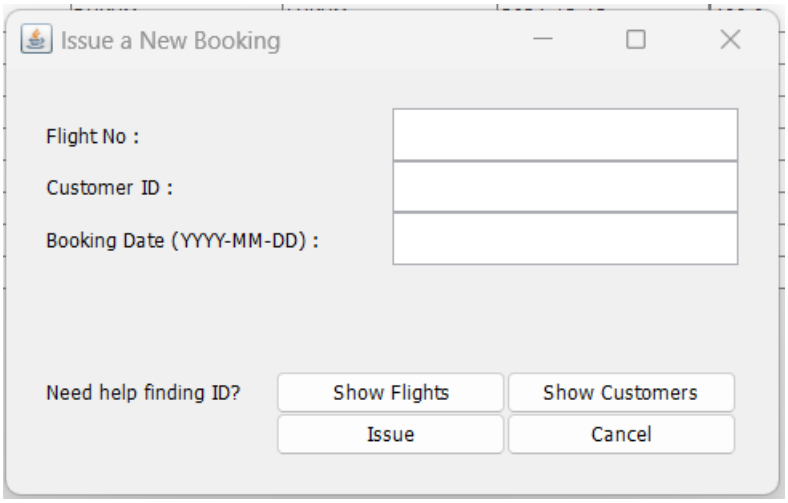
AddFlightWindow



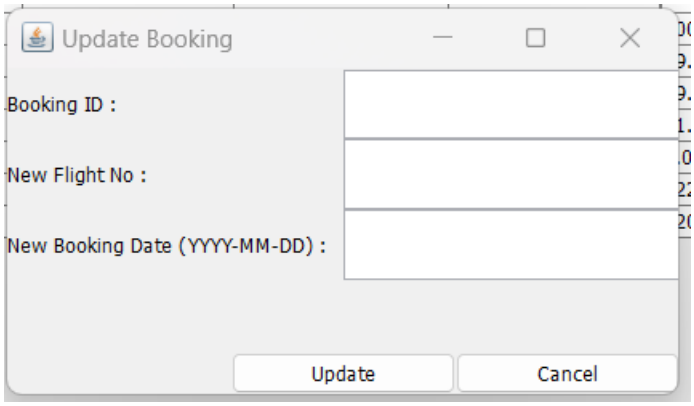
Deleteflight



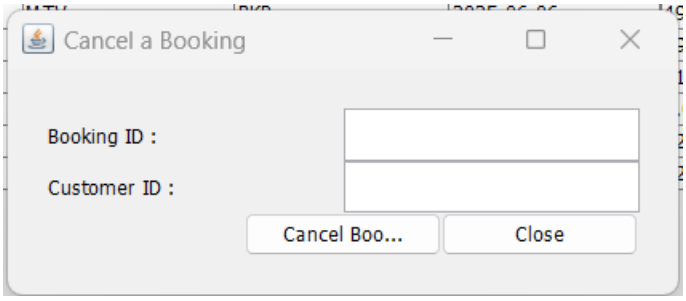
IssueBookingWindow



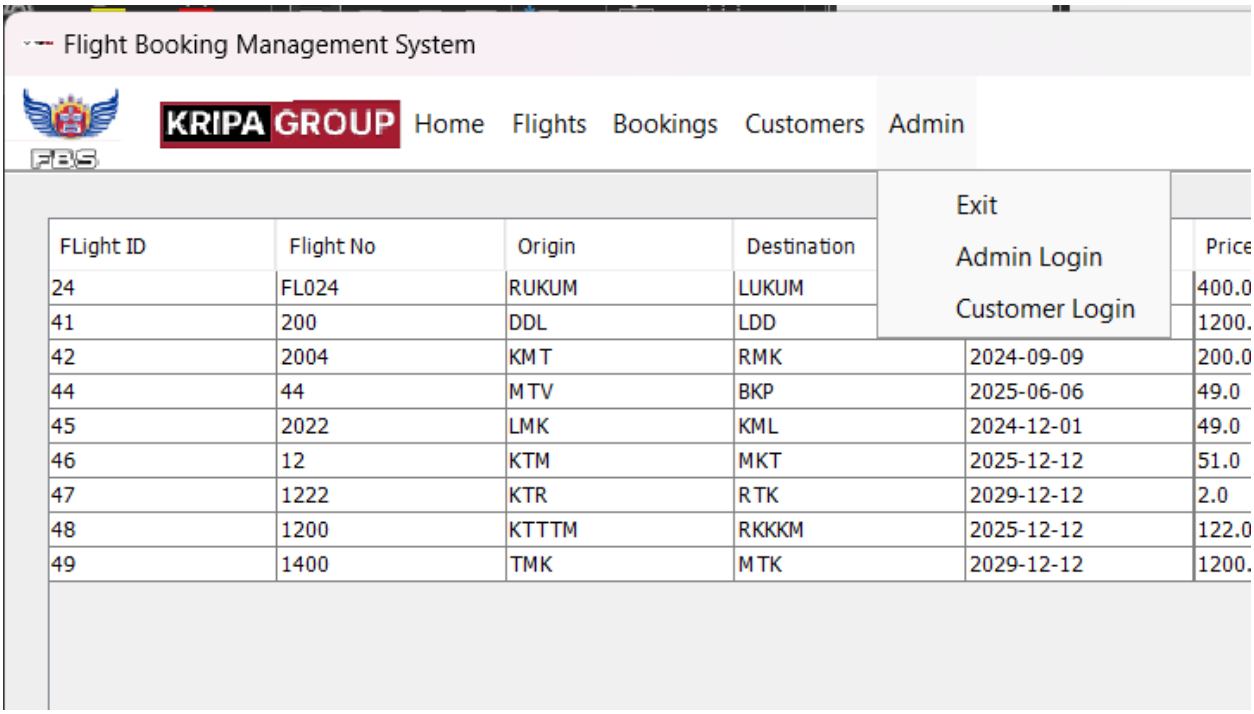
UpdateBookingWindow



Cancel Booking

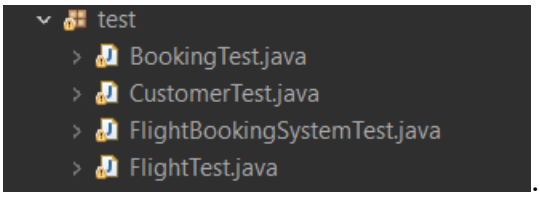


AdminLoginExitWindow



Testing

I have through testing for the code written in this flight booking system. These are files structure



Bookingtest.java

CustomerTest.java

FlightBookingSystemTest.java

```java
public class FlightTest {

    private Flight flight;
    private Customer passenger1;
    private Customer passenger2;

    @BeforeEach
    public void setUp() {
        flight = new Flight(1, "ABC123", "OriginCity", "DestinationCity",
                LocalDate.of(2024, 7, 1), 100, 200.0, false);

        passenger1 = new Customer(7, "Matthew Martinez", "1111111111", "matthew@example.com", false,false);
        passenger2 = new Customer(8, "Olivia Robinson", "9999999999", "olivia@example.com", true,false);

    }

    @Test
    public void testAddPassenger() throws FlightBookingSystemException {
        flight.addPassenger(passenger1);
        assertEquals(1, flight.getPassengerCount());
        assertTrue(flight.getPassengers().contains(passenger1));
    }

    @Test
    public void testAddPassengerAtFullCapacity() throws FlightBookingSystemException {
        for (int i = 0; i < 100; i++) {
```

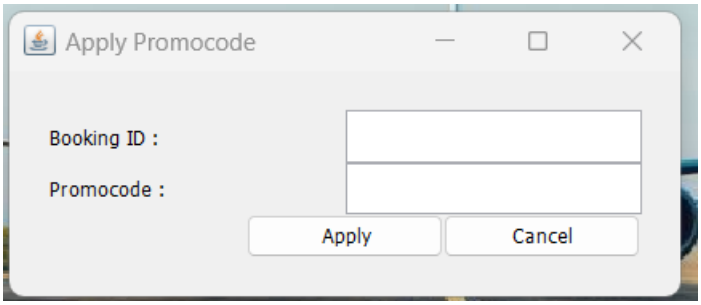
FlightTest.java

These are the test files that I created for testing this system. In addition to it, I have also created a testing that are mentioned in assessment brief.

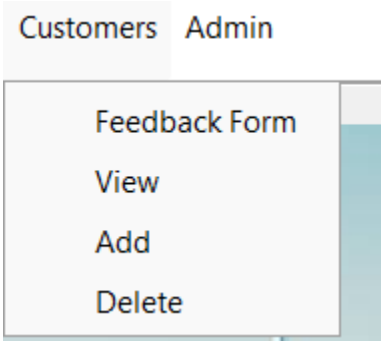
Additional Features and Enhancement

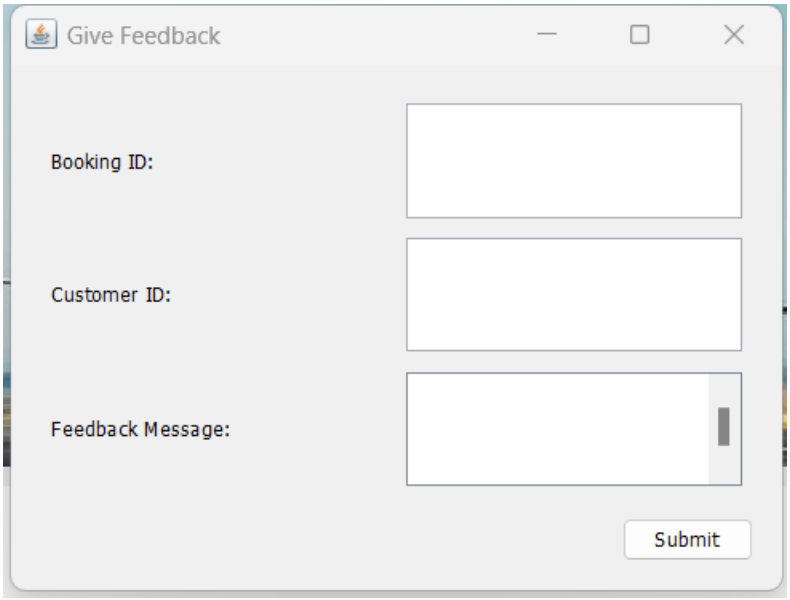
VIP Seat Allocation

Apply Promocode



Feedback form





Conclusion

In summary, the flight booking system offers a user-friendly interface with an aesthetic GUI, making interaction with the booking system both intuitive and visually appealing, thereby enhancing the overall user experience and simplifying the booking process. This can be used for handling

operations in flight booking management company. By the use of OOP principes, this flight booking will play crucial role in digitalizing the transactions and flight schedule and other operations