CMP5332 OBJECT ORIENTED PROGRAMMING IN JAVA

FLIGHT BOOKING SYSTEM

Submitted By: Prashanta Acharya, Srijana Subedi

Student ID: 24128456, 24128457

BSc (Hons) Computer Science with Artificial Intelligence

(Faculty of Computing, Engineering and the Built Environment)

Birmingham City University (BCU)

Sunway college Kathmandu, Nepal

Contents

Introduction 4
Entities/Classes Involved 4
Main.java 5
Commands Package 6
AddBooking.java 7
AddCustomer.java 8
AddFlight.java 9
CancelBooking.java 9
Help.java 10
ListCustomers.java 11
ListFlights.java 11
ShowCustomer.java 12
ShowFlight.java 13
EditBooking.java 13
Feedback.java 14
Models Package 15
Booking.java 15
Customer.java 16
Flight.java 17
FlightBookingSystem.java 18
Feedback.java 19
Data Storage 20
Bookings.txt 20
Customers.txt 20
Flights.txt 20
Feedback.txt 21
Data Management System 21
BookingDataManager.java 22
CustomerDataManager.java 23
FlightDataManager.java 24
DataManager.java 25
FlightBookingSystemData.java 26

FeedbackDataManager.java 27
Graphical User Interface(GUI) 28
Landing Frame 28
AddCustomerWindow 28
DeleteCustomerWindow 29
View CustomerWindow 29
AddFlightWindow 30
View Flights 30
Deleteflight 31
AddBooking 31
UpdateBookingWindow 32
Cancel Booking 32
AddFeedback 33
Testing 33
FlightAndCustomerTest.java 33
Additional Features and Enhancement 34
Flight Class 34
Feedback 34
Food Preference 35
Conclusion 36

Flight Booking System

Introduction

The Flight Booking System is a Java application that was created with the help of object-oriented programming techniques. With a user-friendly interface, it offers features for managing customers, flights, and flight reservations. Numerous tasks, such as flight scheduling, customer service, and booking administration, are effectively managed by the system. The project makes use of Java's OOP characteristics, including inheritance, abstraction, and encapsulation, to simplify these processes.

Entities/Classes Involved

To ensure the smooth operation of the flight booking system, five major classes play a crucial role:

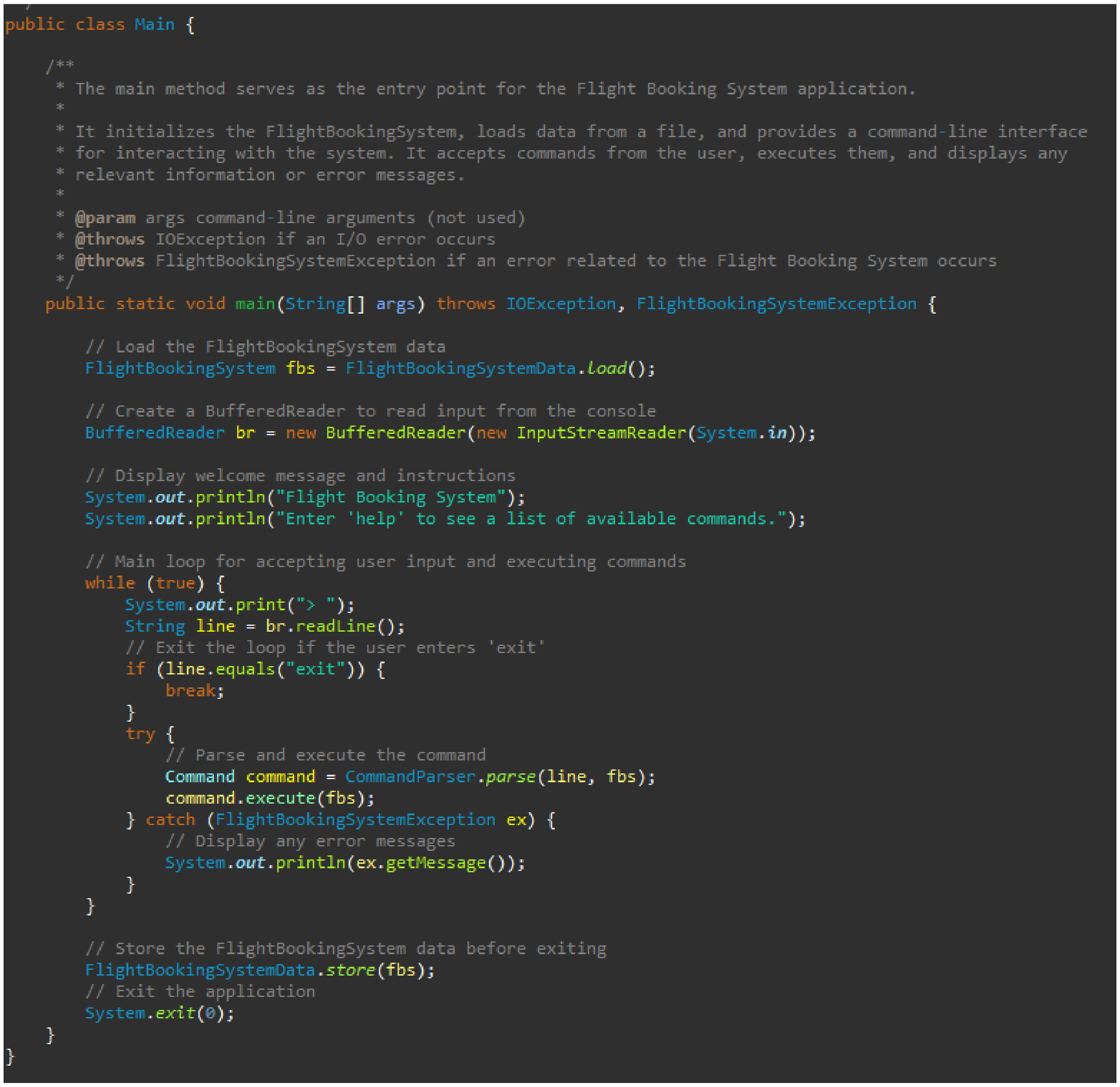
1. Customer - Represents individual passengers using the system.
2. Flight - Holds data related to available flights, such as departure, arrival, and seating capacity.
3. Booking - Manages reservations for customers, linking them to flights.
4. Feedback - Stores user reviews and comments regarding their travel experience.
5. FlightBookingSystem - Acts as the control center, managing all operations between customers, flights, bookings, and feedback.

When a new customer is added, the system processes the request through a dedicated command, stores the customer details, and makes them accessible within FlightBookingSystem. This class maintains the entire ecosystem, ensuring proper handling of user actions and system operations.

Main.java

As the primary execution file, Main.java serves as the backbone of the system, performing these key roles:

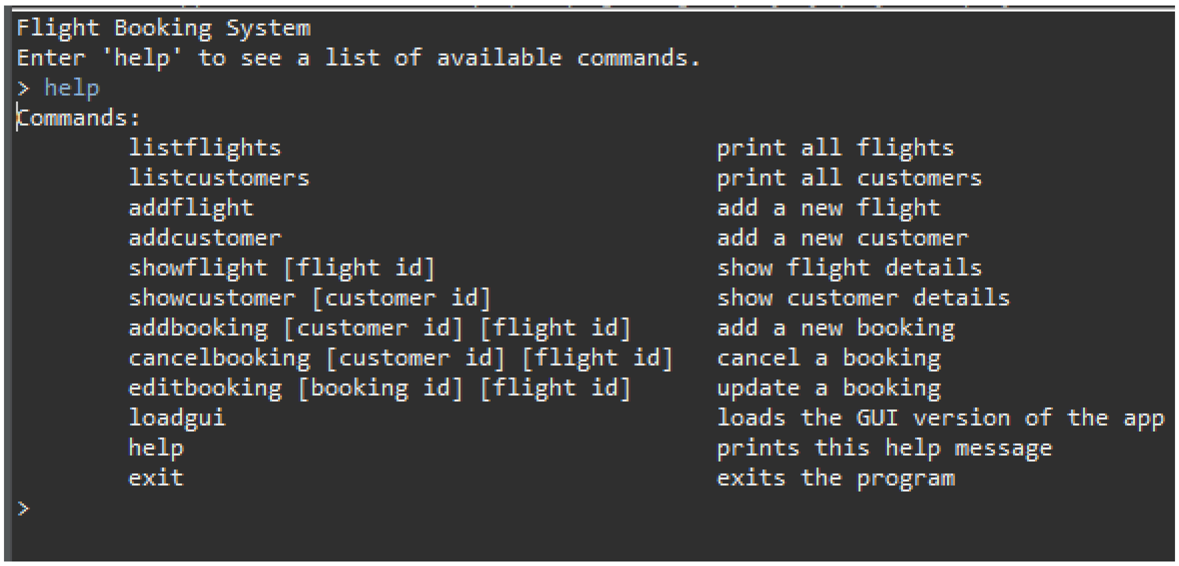
1. Retrieving stored data to populate the system with existing flights, customers, and reservations.
2. Accepting user inputs to determine which operations to execute.
3. Executing CommandParser, which interprets and processes user commands.



```java
public class Main {

    /**
     * The main method serves as the entry point for the Flight Booking System application.
     *
     * It initializes the FlightBookingSystem, loads data from a file, and provides a command-line interface
     * for interacting with the system. It accepts commands from the user, executes them, and displays any
     * relevant information or error messages.
     *
     * @param args command-line arguments (not used)
     * @throws IOException if an I/O error occurs
     * @throws FlightBookingSystemException if an error related to the Flight Booking System occurs
     */
    public static void main(String[] args) throws IOException, FlightBookingSystemException {

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

        // Create a BufferedReader to read input from the console
        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 accepting user input and executing commands
        while (true) {
            System.out.print("> ");
            String line = br.readLine();
            // Exit the loop if the user enters 'exit'
            if (line.equals("exit")) {
                break;
            }
            try {
                // Parse and execute the command
                Command command = CommandParser.parse(line, fbs);
                command.execute(fbs);
            } catch (FlightBookingSystemException ex) {
                // Display any error messages
                System.out.println(ex.getMessage());
            }
        }

        // Store the FlightBookingSystem data before exiting
        FlightBookingSystemData.store(fbs);
        // Exit the application
        System.exit(0);
    }
}
```

Fig: code snippet of main.java

When users type help, the system responds with a list of available commands and their descriptions, guiding them on how to navigate the interface.With Main.java setting up the system, the next section details the Commands Package, which executes specific actions based on user input.



```
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
        loadgui                                     loads the GUI version of the app
        help                                        prints this help message
        exit                                        exits the program
>
```

Fig: command snippet of Flight Booking System

Commands Package

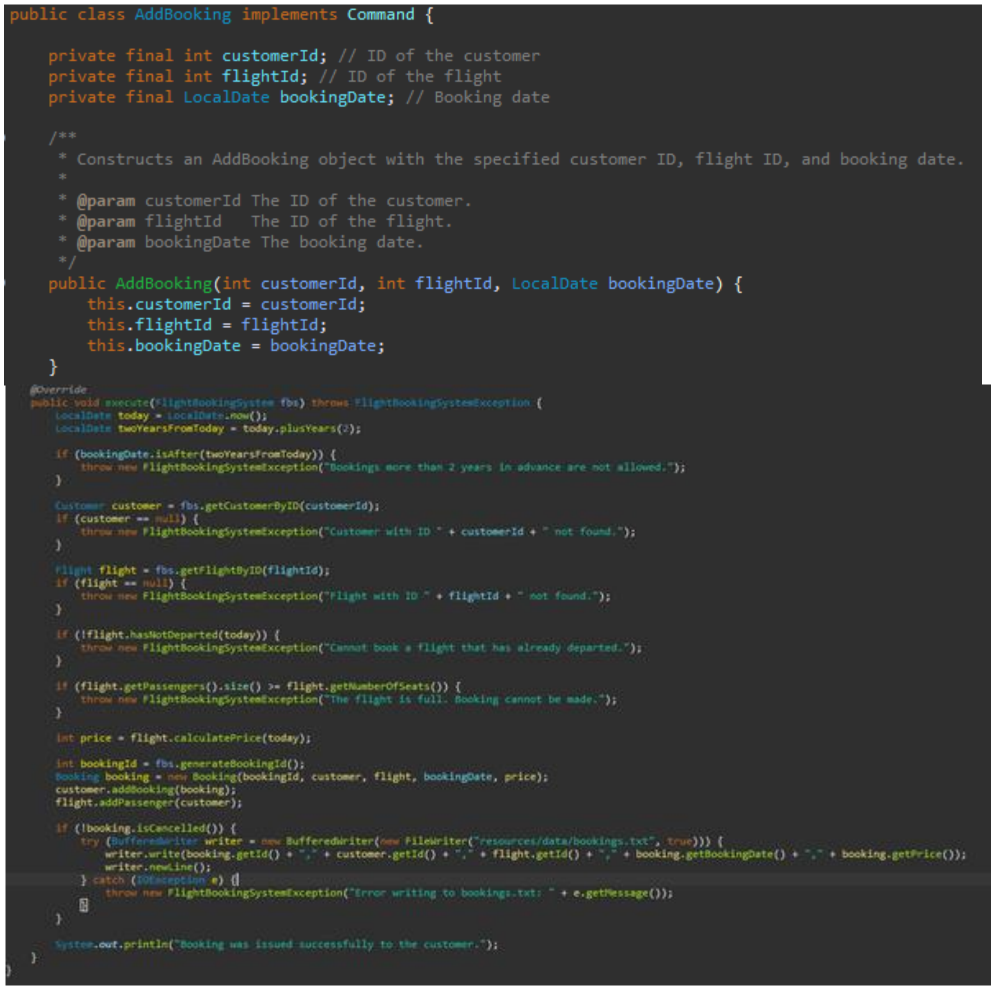
The commands package contains various classes responsible for handling user actions. Each command executes a specific function, allowing seamless interaction with the system. Below are the core classes and their functionalities:

- AddBooking.java
- AddCustomer.java
- AddFlight.java
- CancelBooking.java
- DeleteCustomer.java
- DeleteFlight.java
- Help.java
- ListCustomers.java
- ListFlights.java
- ShowCustomer.java
- ShowFlight.java
- ShowFlights.java

- UpdateBooking.java
- AddFeedback.java

AddBooking.java

The AddBooking class oversees adding a new reservation for a client on a certain aircraft. First, it checks to see if the flight and client supplied are in the system. If one of them is absent, the reservation procedure is stopped. After confirming their existence, the system determines whether there are seats available on the aircraft. The consumer gets linked to the flight once the booking is successfully added, if there are seats available. However, an exception is made, and the booking cannot be finalized if the flight is already filled. By doing this, overbooking is prevented, and the flight's seating capacity is preserved.

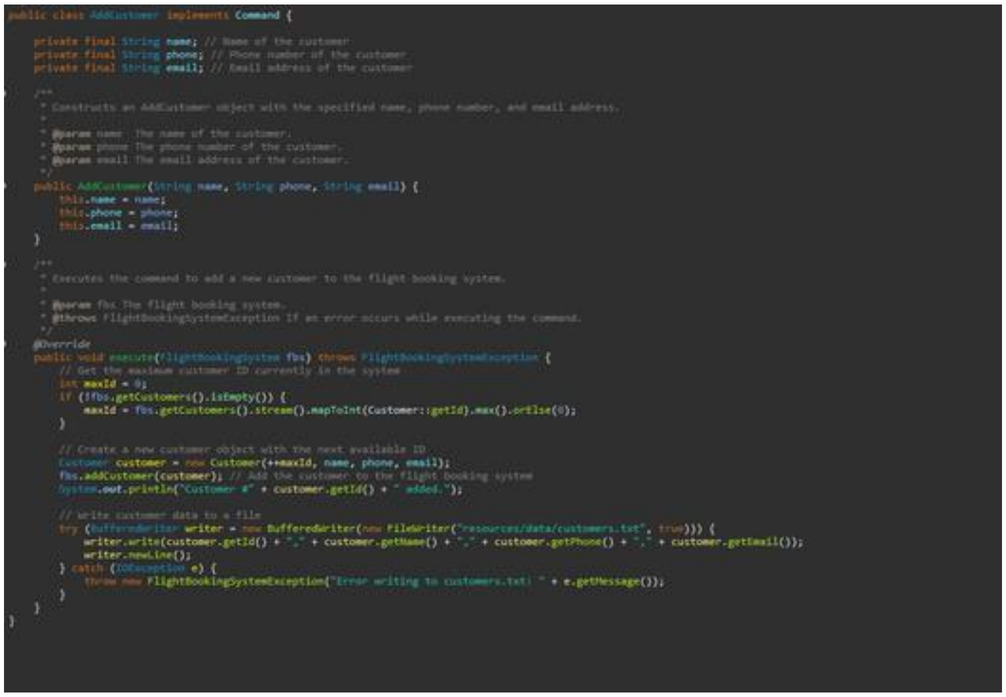


```java
public class AddBooking implements Command {

    private final int customerId; // ID of the customer
    private final int flightId; // ID of the flight
    private final LocalDate bookingDate; // Booking date

    /**
     * Constructs an AddBooking object with the specified customer ID, flight ID, and booking date.
     *
     * @param customerId The ID of the customer.
     * @param flightId   The ID of the flight.
     * @param bookingDate The booking date.
     */
    public AddBooking(int customerId, int flightId, LocalDate bookingDate) {
        this.customerId = customerId;
        this.flightId = flightId;
        this.bookingDate = bookingDate;
    }
  @Override
  public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
      LocalDate today = LocalDate.now();
      LocalDate twoYearsFromToday = today.plusYears(2);

      if (bookingDate.isAfter(twoYearsFromToday)) {
          throw new FlightBookingSystemException("Bookings more than 2 years in advance are not allowed.");
      }

      Customer customer = fbs.getCustomerByID(customerId);
      if (customer == null) {
          throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
      }

      Flight flight = fbs.getFlightByID(flightId);
      if (flight == null) {
          throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
      }

      if (!flight.hasNotDeparted(today)) {
          throw new FlightBookingSystemException("Cannot book a flight that has already departed.");
      }

      if (flight.getPassengers().size() >= flight.getNumberOfSeats()) {
          throw new FlightBookingSystemException("The flight is full. Booking cannot be made.");
      }

      int price = flight.calculatePrice(today);

      int bookingId = fbs.generateBookingId();
      Booking booking = new Booking(bookingId, customer, flight, bookingDate, price);
      customer.addBooking(booking);
      flight.addPassenger(customer);

      if (!booking.isCancelled()) {
          try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/bookings.txt", true))) {
              writer.write(booking.getId() + "," + customer.getId() + "," + flight.getId() + "," + booking.getBookingDate() + "," + booking.getPrice());
              writer.newLine();
          } catch (IOException e) {
              throw new FlightBookingSystemException("Error writing to bookings.txt: " + e.getMessage());
          }
      }

      System.out.println("Booking was issued successfully to the customer.");
  }
}
```

Fig: code snippet of AddBooking

AddCustomer.java

The AddCustomer class is designed to add a new customer to the flight booking system. It takes essential customer details, including their name, phone number, and a unique ID, ensuring that each customer is identifiable within the system. Once the necessary details are provided, the customer is added to the system's database. This functionality allows the system to manage customer records efficiently, enabling future operations such as booking flights, retrieving customer details, and managing reservations.

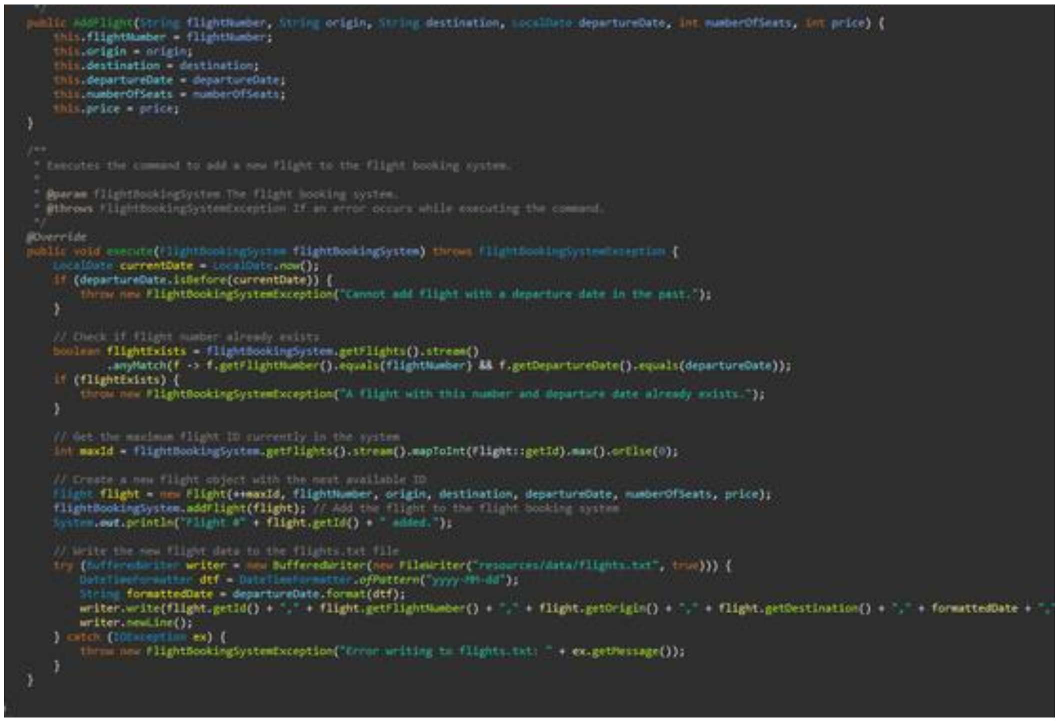


```java
public class AddCustomer implements Command {

    private final String name; // Name of the customer
    private final String phone; // Phone number of the customer
    private final String email; // Email address of the customer

    /**
     * Constructs an AddCustomer object 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 command to add a new customer to the flight booking system.
     *
     * @param fbs The flight booking system.
     * @throws FlightBookingSystemException If an error occurs while executing the command.
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        // Get the maximum customer ID currently in the system
        int maxId = 0;
        if (!fbs.getCustomers().isEmpty()) {
            maxId = fbs.getCustomers().stream().mapToInt(Customer::getId).max().orElse(0);
        }

        // Create a new customer object with the next available ID
        Customer customer = new Customer(++maxId, name, phone, email);
        fbs.addCustomer(customer); // Add the customer to the flight booking system
        System.out.println("Customer #" + customer.getId() + " added.");

        // Write customer data to a file
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/customers.txt", true))) {
            writer.write(customer.getId() + "," + customer.getName() + "," + customer.getPhone() + "," + customer.getEmail());
            writer.newLine();
        } catch (IOException e) {
            throw new FlightBookingSystemException("Error writing to customers.txt: " + e.getMessage());
        }
    }
}
```

Fig: code snippet of AddCustomer

AddFlight.java

The AddFlight class is responsible for adding a new flight to the system. It collects essential flight details, including a unique flight ID, the origin and destination locations, the seating capacity, and the scheduled departure time. Once these details are provided, the flight is added to the system, making it available for customers to book. This functionality ensures that the system maintains an updated list of flights, allowing for efficient flight management and scheduling.

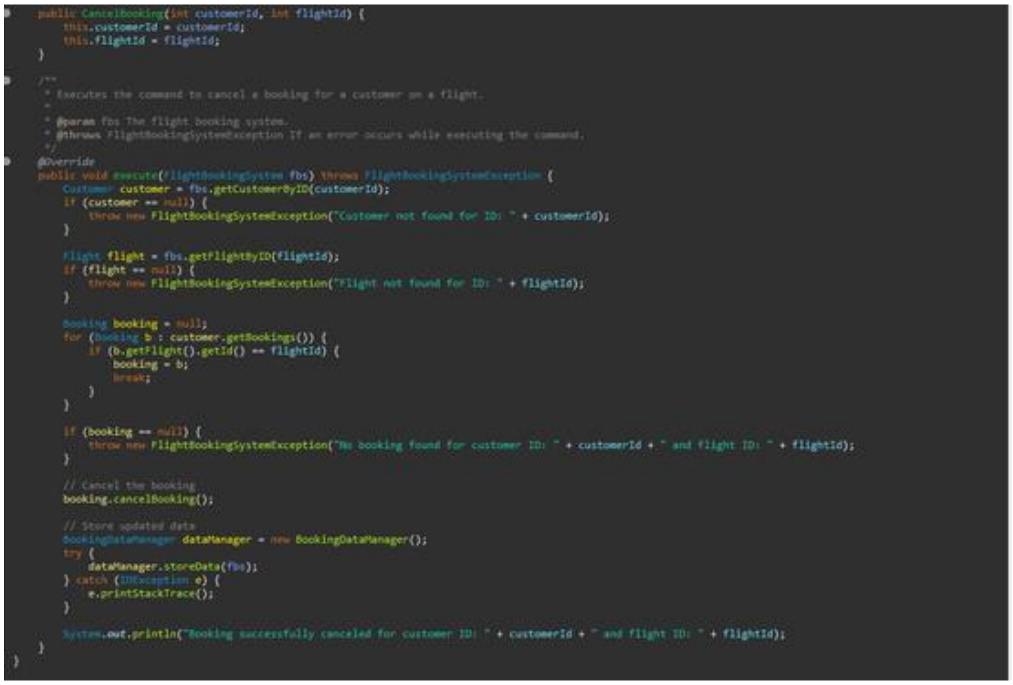


```java
public AddFlight(String flightNumber, String origin, String destination, LocalDate departureDate, int numberOfSeats, int price) {
    this.flightNumber = flightNumber;
    this.origin = origin;
    this.destination = destination;
    this.departureDate = departureDate;
    this.numberOfSeats = numberOfSeats;
    this.price = price;
}

/**
 * Executes the command to add a new flight to the flight booking system.
 *
 * @param flightBookingSystem The flight booking system.
 * @throws FlightBookingSystemException If an error occurs while executing the command.
 */
@Override
public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
    LocalDate currentDate = LocalDate.now();
    if (departureDate.isBefore(currentDate)) {
        throw new FlightBookingSystemException("Cannot add flight with a departure date in the past.");
    }

    // Check if flight number already exists
    boolean flightExists = flightBookingSystem.getFlights().stream()
            .anyMatch(f -> f.getFlightNumber().equals(flightNumber) && f.getDepartureDate().equals(departureDate));
    if (flightExists) {
        throw new FlightBookingSystemException("A flight with this number and departure date already exists.");
    }

    // Get the maximum flight ID currently in the system
    int maxId = flightBookingSystem.getFlights().stream().mapToInt(Flight::getId).max().orElse(0);

    // Create a new flight object with the next available ID
    Flight flight = new Flight(++maxId, flightNumber, origin, destination, departureDate, numberOfSeats, price);
    flightBookingSystem.addFlight(flight); // Add the flight to the flight booking system
    System.out.println("Flight #" + flight.getId() + " added.");

    // Write the new flight data to the flights.txt file
    try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/flights.txt", true))) {
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        String formattedDate = departureDate.format(dtf);
        writer.write(flight.getId() + "," + flight.getFlightNumber() + "," + flight.getOrigin() + "," + flight.getDestination() + "," + formattedDate + ","
        writer.newLine();
    } catch (IOException ex) {
        throw new FlightBookingSystemException("Error writing to flights.txt: " + ex.getMessage());
    }
}
```

Fig: code snippet of AddFlight

CancelBooking.java

The CancelBooking class is designed to handle the cancellation of an existing booking in the system. It works by searching for a booking using the customer's ID and the flight ID to ensure that the reservation exists. If a matching booking is found, it is removed from the system, freeing up a seat on the flight. This functionality allows customers to modify their travel plans efficiently while ensuring that flight availability is updated in real time.

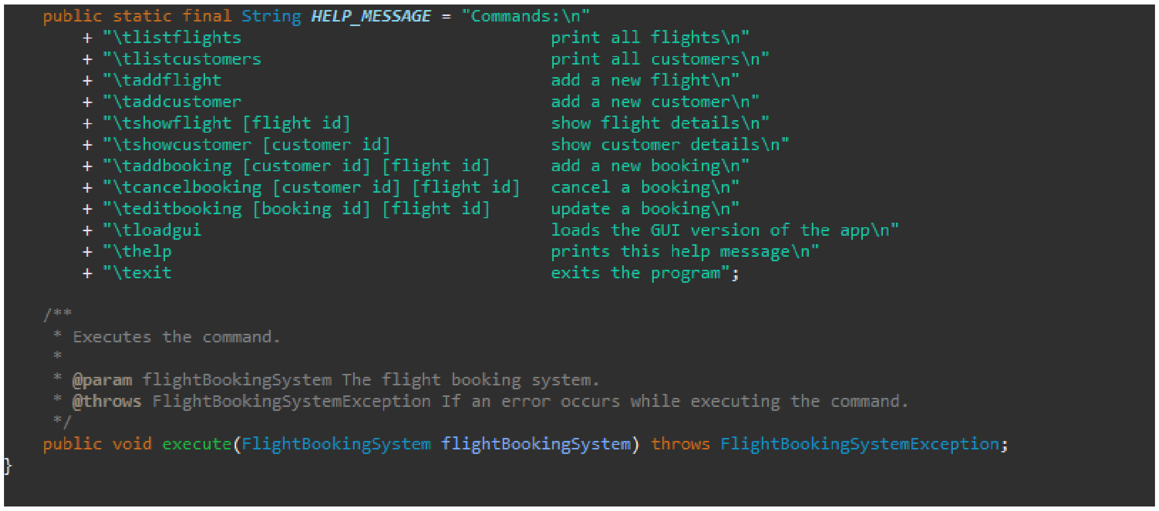


```java
    public CancelBooking(int customerId, int flightId) {
        this.customerId = customerId;
        this.flightId = flightId;
    }

    /**
     * Executes the command to cancel a booking for a customer on a flight.
     *
     * @param fbs The flight booking system.
     * @throws FlightBookingSystemException If an error occurs while executing the command.
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer not found for ID: " + customerId);
        }

        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight not found for ID: " + flightId);
        }

        Booking booking = null;
        for (Booking b : customer.getBookings()) {
            if (b.getFlight().getId() == flightId) {
                booking = b;
                break;
            }
        }

        if (booking == null) {
            throw new FlightBookingSystemException("No booking found for customer ID: " + customerId + " and flight ID: " + flightId);
        }

        // Cancel the booking
        booking.cancelBooking();

        // Store updated data
        BookingDataManager dataManager = new BookingDataManager();
        try {
            dataManager.storeData(fbs);
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Booking successfully canceled for customer ID: " + customerId + " and flight ID: " + flightId);
    }
}
```

Fig: code snippet of CancelBooking

Help.java

The Help class is responsible for displaying a list of available commands within the flight booking system. It provides users with an overview of the different actions they can perform, such as adding customers, booking flights, listing available flights, and retrieving customer details. By offering clear guidance on system functionality, this feature enhances user experience and ensures smooth navigation within the system.

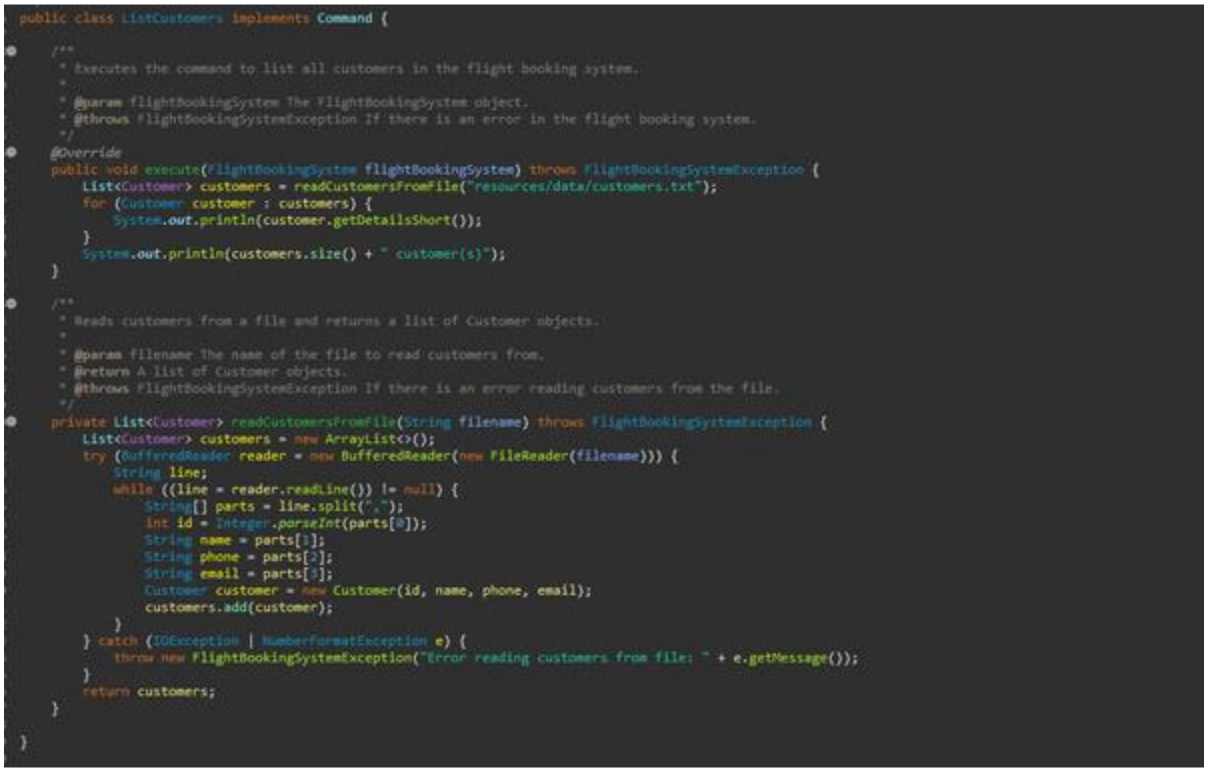


```java
    public static final String HELP_MESSAGE = "Commands:\n"
        + "\tlistflights                              print all flights\n"
        + "\tlistcustomers                            print all customers\n"
        + "\taddflight                                add a new flight\n"
        + "\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"
        + "\tloadgui                                  loads the GUI version of the app\n"
        + "\thelp                                     prints this help message\n"
        + "\texit                                     exits the program";

    /**
     * Executes the command.
     *
     * @param flightBookingSystem The flight booking system.
     * @throws FlightBookingSystemException If an error occurs while executing the command.
     */
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException;
}
```

Fig: code snippet of help commands

ListCustomers.java

The ListCustomers class is responsible for displaying a list of all registered customers in the system. It retrieves and presents customer details, allowing administrators or users to view essential information about each customer. This functionality ensures efficient customer management by providing a quick and organized way to access customer records, making it easier to handle bookings, updates, and other related operations.

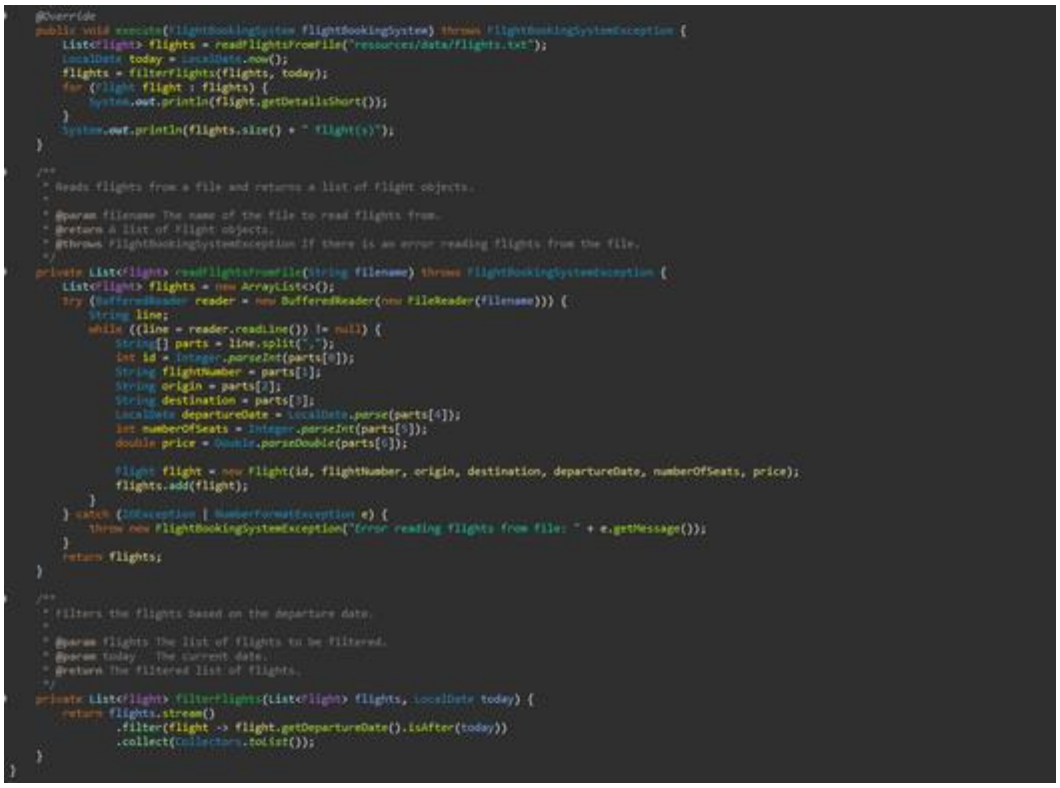


```java
public class ListCustomers implements Command {

    /**
     * Executes the command to list all customers in the flight booking system.
     *
     * @param flightBookingSystem The FlightBookingSystem object.
     * @throws FlightBookingSystemException If there is an error in the flight booking system.
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Customer> customers = readCustomersFromFile("resources/data/customers.txt");
        for (Customer customer : customers) {
            System.out.println(customer.getDetailsShort());
        }
        System.out.println(customers.size() + " customer(s)");
    }

    /**
     * Reads customers from a file and returns a list of Customer objects.
     *
     * @param filename The name of the file to read customers from.
     * @return A list of Customer objects.
     * @throws FlightBookingSystemException If there is an error reading customers from the file.
     */
    private List<Customer> readCustomersFromFile(String filename) throws FlightBookingSystemException {
        List<Customer> customers = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String name = parts[1];
                String phone = parts[2];
                String email = parts[3];
                Customer customer = new Customer(id, name, phone, email);
                customers.add(customer);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading customers from file: " + e.getMessage());
        }
        return customers;
    }

}
```

Fig: code snippet of ListCustomers

ListFlights.java

The ListFlights class displays all flights available in the system. It retrieves and presents flight details, allowing users to view essential information such as flight ID, origin, destination, capacity, and departure time. This feature ensures easy access to flight schedules and helps in managing bookings efficiently.

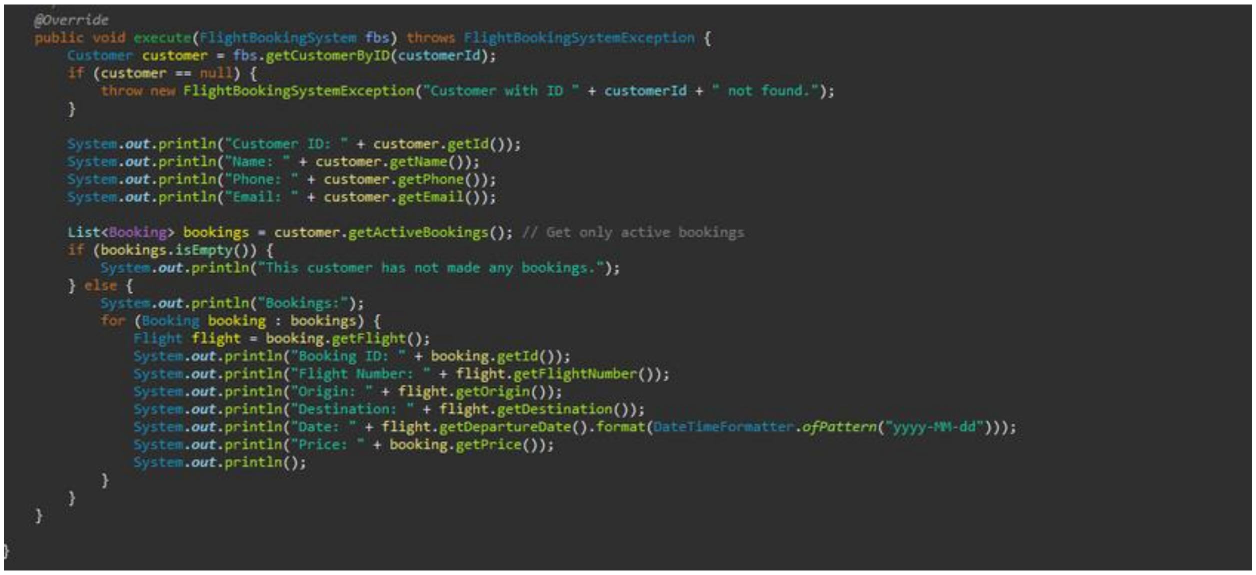


```java
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Flight> flights = readFlightsFromFile("resources/data/flights.txt");
        LocalDate today = LocalDate.now();
        flights = filterFlights(flights, today);
        for (Flight flight : flights) {
            System.out.println(flight.getDetailsShort());
        }
        System.out.println(flights.size() + " flight(s)");
    }

    /**
     * Reads flights from a file and returns a list of Flight objects.
     *
     * @param filename The name of the file to read flights from.
     * @return A list of Flight objects.
     * @throws FlightBookingSystemException If there is an error reading flights from the file.
     */
    private List<Flight> readFlightsFromFile(String filename) throws FlightBookingSystemException {
        List<Flight> flights = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String flightNumber = parts[1];
                String origin = parts[2];
                String destination = parts[3];
                LocalDate departureDate = LocalDate.parse(parts[4]);
                int numberOfSeats = Integer.parseInt(parts[5]);
                double price = Double.parseDouble(parts[6]);

                Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
                flights.add(flight);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading flights from file: " + e.getMessage());
        }
        return flights;
    }

    /**
     * Filters the flights based on the departure date.
     *
     * @param flights The list of flights to be filtered.
     * @param today   The current date.
     * @return The filtered list of flights.
     */
    private List<Flight> filterFlights(List<Flight> flights, LocalDate today) {
        return flights.stream()
                .filter(flight -> flight.getDepartureDate().isAfter(today))
                .collect(Collectors.toList());
    }
}
```

Fig: code snippet of ListFlights

ShowCustomer.java

The ShowCustomer class retrieves and displays details of a specific customer. By using the customer's unique ID, the system fetches relevant information, such as their name and contact details. This feature allows for quick access to customer records, making it easier to manage bookings and provide support when needed.

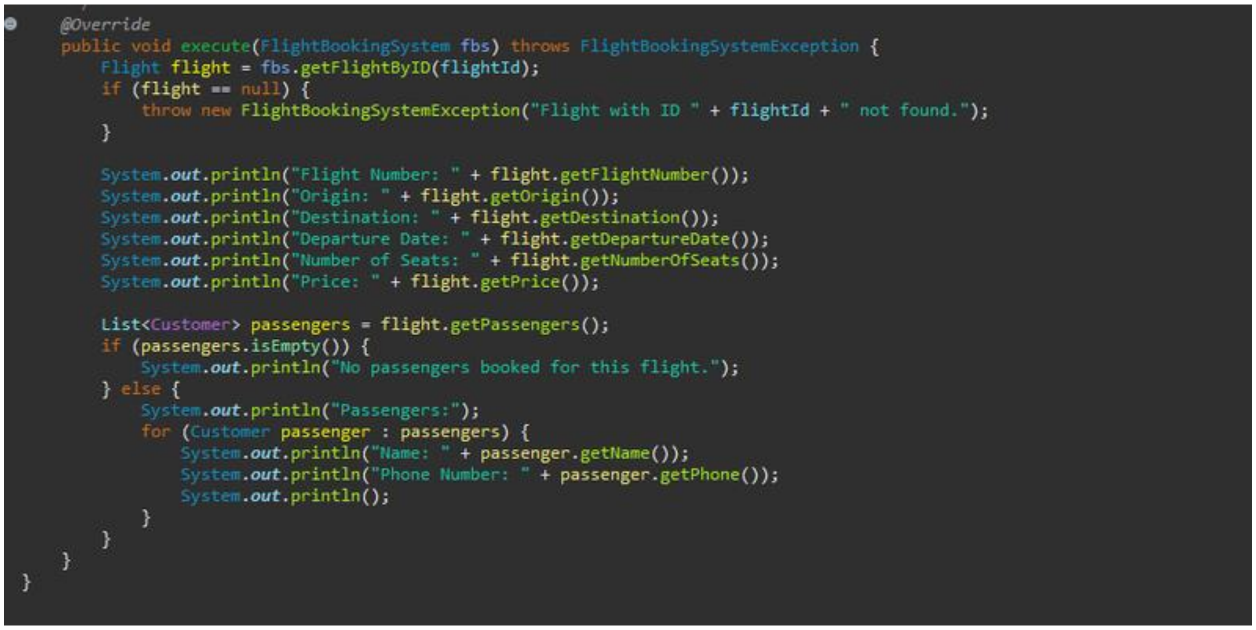


```java
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
        }

        System.out.println("Customer ID: " + customer.getId());
        System.out.println("Name: " + customer.getName());
        System.out.println("Phone: " + customer.getPhone());
        System.out.println("Email: " + customer.getEmail());

        List<Booking> bookings = customer.getActiveBookings(); // Get only active bookings
        if (bookings.isEmpty()) {
            System.out.println("This customer has not made any bookings.");
        } else {
            System.out.println("Bookings:");
            for (Booking booking : bookings) {
                Flight flight = booking.getFlight();
                System.out.println("Booking ID: " + booking.getId());
                System.out.println("Flight Number: " + flight.getFlightNumber());
                System.out.println("Origin: " + flight.getOrigin());
                System.out.println("Destination: " + flight.getDestination());
                System.out.println("Date: " + flight.getDepartureDate().format(DateTimeFormatter.ofPattern("yyyy-MM-dd")));
                System.out.println("Price: " + booking.getPrice());
                System.out.println();
            }
        }
    }
}
```

Fig: code snippet of ShowCustomer

ShowFlight.java

The ShowFlight class is responsible for retrieving and displaying details of a specific flight. Using the flight's unique ID, it provides essential information such as origin, destination, capacity, and departure time. This feature allows users to quickly access flight details for booking, updates, or management purposes.

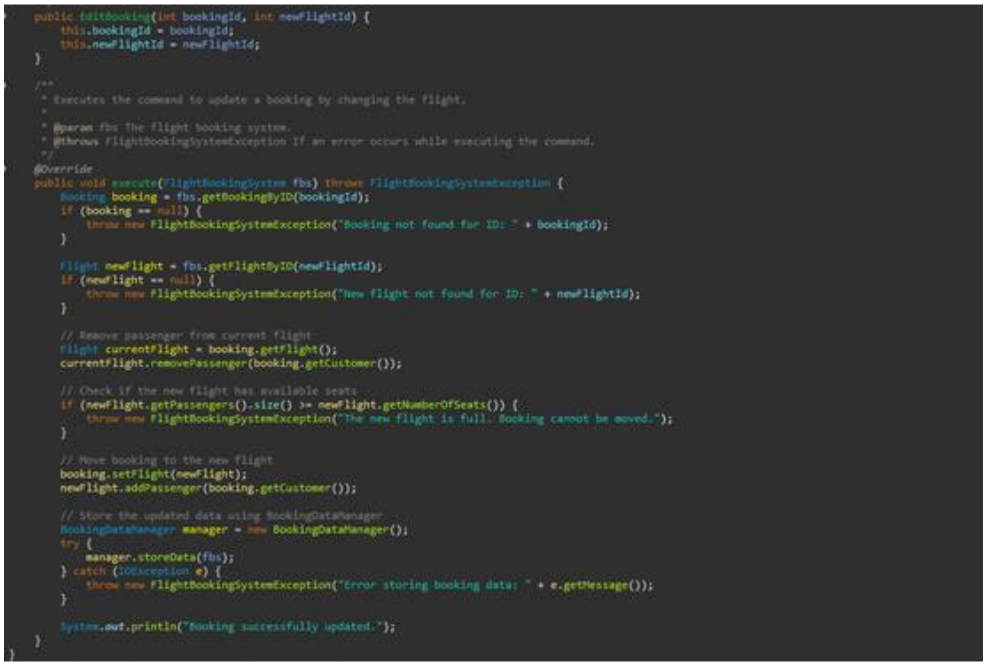


```java
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
        }

        System.out.println("Flight Number: " + flight.getFlightNumber());
        System.out.println("Origin: " + flight.getOrigin());
        System.out.println("Destination: " + flight.getDestination());
        System.out.println("Departure Date: " + flight.getDepartureDate());
        System.out.println("Number of Seats: " + flight.getNumberOfSeats());
        System.out.println("Price: " + flight.getPrice());

        List<Customer> passengers = flight.getPassengers();
        if (passengers.isEmpty()) {
            System.out.println("No passengers booked for this flight.");
        } else {
            System.out.println("Passengers:");
            for (Customer passenger : passengers) {
                System.out.println("Name: " + passenger.getName());
                System.out.println("Phone Number: " + passenger.getPhone());
                System.out.println();
            }
        }
    }
}
```

Fig: code snippet of ShowFlights

EditBooking.java

The purpose of this function is to update an existing booking. It allows modifications to be made to the booking details, ensuring that any changes or corrections, such as date, time, or customer information, can be applied to an existing reservation. This update ensures that the booking information is accurate and up to date for both the customer and the service provider.

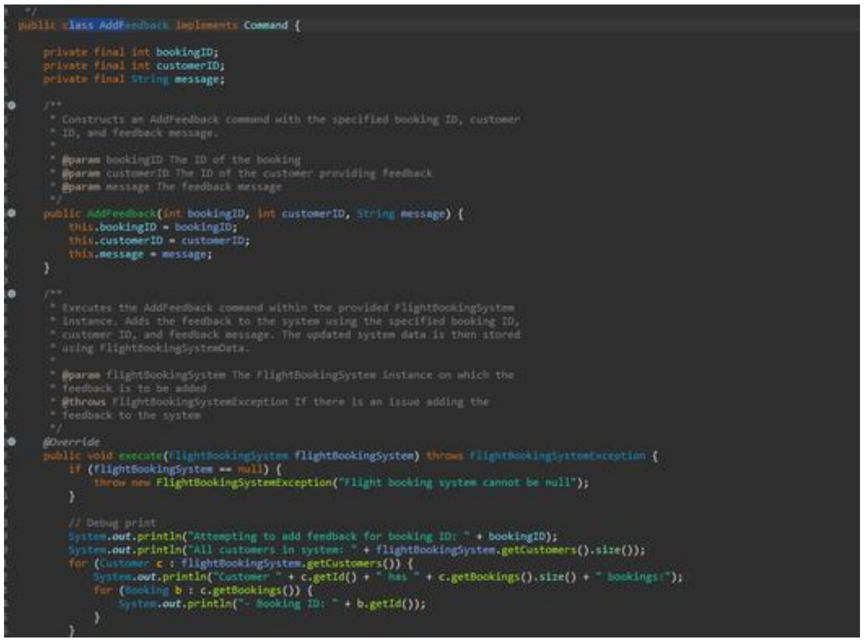


```java
    public EditBooking(int bookingId, int newFlightId) {
        this.bookingId = bookingId;
        this.newFlightId = newFlightId;
    }

    /**
     * Executes the command to update a booking by changing the flight.
     *
     * @param fbs The flight booking system.
     * @throws FlightBookingSystemException If an error occurs while executing the command.
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Booking booking = fbs.getBookingByID(bookingId);
        if (booking == null) {
            throw new FlightBookingSystemException("Booking not found for ID: " + bookingId);
        }

        Flight newFlight = fbs.getFlightByID(newFlightId);
        if (newFlight == null) {
            throw new FlightBookingSystemException("New flight not found for ID: " + newFlightId);
        }

        // Remove passenger from current flight
        Flight currentFlight = booking.getFlight();
        currentFlight.removePassenger(booking.getCustomer());

        // Check if the new flight has available seats
        if (newFlight.getPassengers().size() >= newFlight.getNumberOfSeats()) {
            throw new FlightBookingSystemException("The new flight is full. Booking cannot be moved.");
        }

        // Move booking to the new flight
        booking.setFlight(newFlight);
        newFlight.addPassenger(booking.getCustomer());

        // Store the updated data using BookingDataManager
        BookingDataManager manager = new BookingDataManager();
        try {
            manager.storeData(fbs);
        } catch (IOException e) {
            throw new FlightBookingSystemException("Error storing booking data: " + e.getMessage());
        }

        System.out.println("Booking successfully updated.");
    }
}
```

Fig: code snippet of EditBooking

Feedback.java

The Feedback.java class allows customers to submit comments and ratings about their travel experience, which are stored in a text file for persistence. Administrators can access and review this feedback to improve service quality and system functionality.



```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
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        if (flightBookingSystem == null) {
            throw new FlightBookingSystemException("Flight booking system cannot be null");
        }

        // Debug print
        System.out.println("Attempting to add feedback for booking ID: " + bookingID);
        System.out.println("All customers in system: " + flightBookingSystem.getCustomers().size());
        for (Customer c : flightBookingSystem.getCustomers()) {
            System.out.println("Customer " + c.getId() + " has " + c.getBookings().size() + " bookings:");
            for (Booking b : c.getBookings()) {
                System.out.println("- Booking ID: " + b.getId());
            }
        }
```

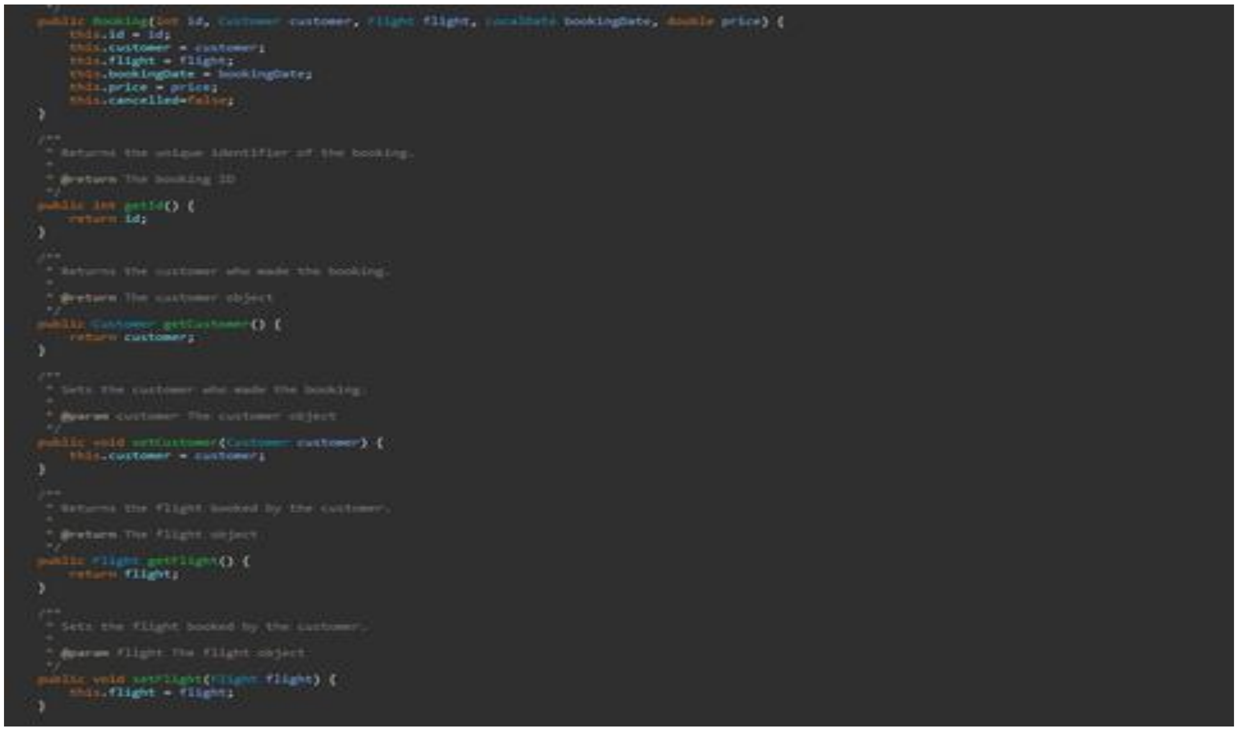
Fig: code snippet of feedback

Models Package

The Models Package in the Flight Booking System contains core classes that define the essential entities: Booking, Customer, Flight, Feedback and FlightBookingSystem. These classes work together to manage flight reservations, customer details, and flight information. The package ensures a structured approach to handling bookings, cancellations, and system management. Below is a detailed explanation of each class.

Booking.java

The Booking class represents a reservation made by a customer for a flight. Each booking has a unique id, a reference to a Customer object, and a reference to a Flight object. The class also includes an isCancelled flag to indicate whether the booking is active. The primary functions of this class include creating a booking, checking its status, and allowing cancellations. This class ensures that a booking can be canceled while maintaining a record of its status.

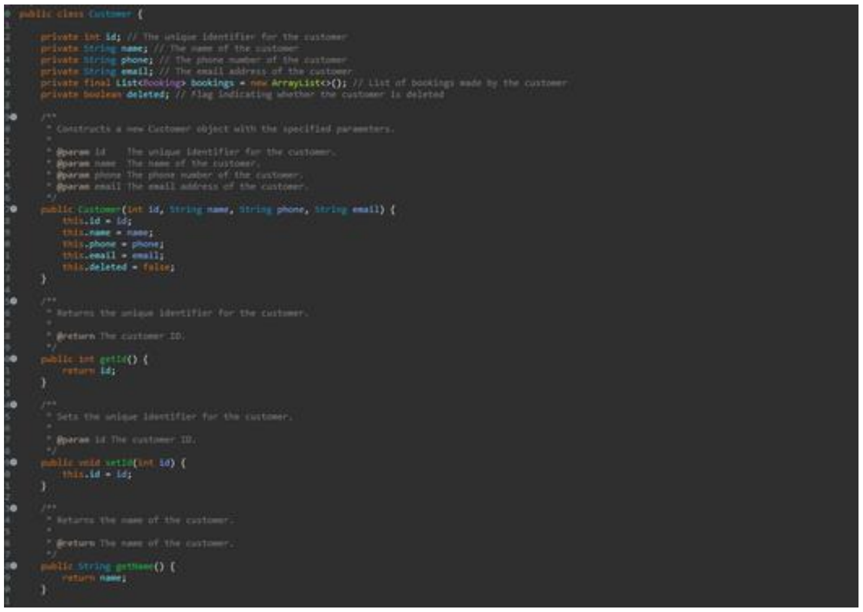


```java
public Booking(int id, Customer customer, Flight flight, LocalDate bookingDate, double price) {
    this.id = id;
    this.customer = customer;
    this.flight = flight;
    this.bookingDate = bookingDate;
    this.price = price;
    this.cancelled=false;
}

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

/**
 * Returns the customer who made the booking.
 *
 * @return The customer object
 */
public Customer getCustomer() {
    return customer;
}

/**
 * Sets the customer who made the booking.
 *
 * @param customer The customer object
 */
public void setCustomer(Customer customer) {
    this.customer = customer;
}
/**
 * Returns the flight booked by the customer.
 *
 * @return The flight object
 */
public Flight getFlight() {
    return flight;
}

/**
 * Sets the flight booked by the customer.
 *
 * @param flight The flight object
 */
public void setFlight(Flight flight) {
    this.flight = flight;
}
```

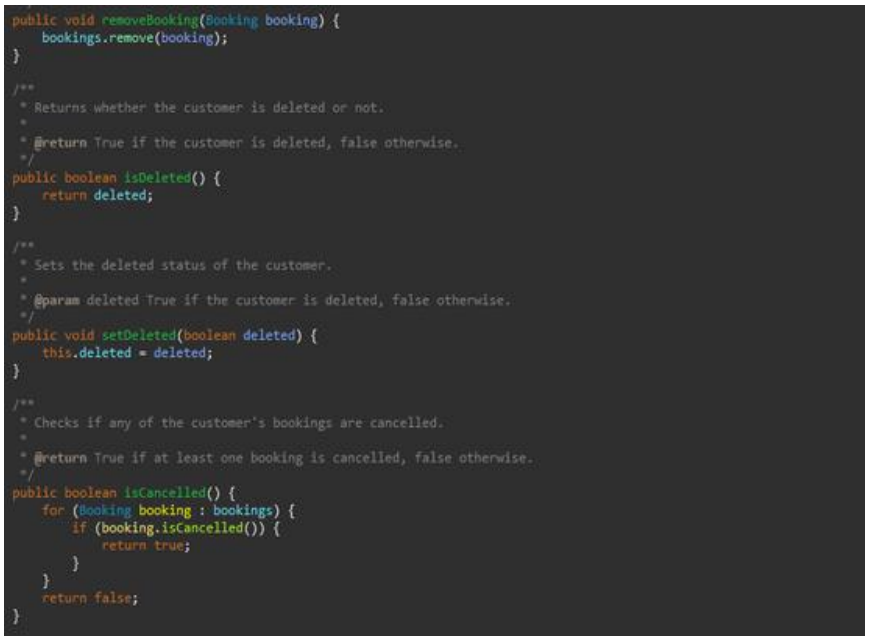
Fig: code snippet of Bookin.java

Customer.java

The Customer class defines a passenger in the booking system. It includes attributes such as id, name, and phone, along with a list of Booking objects associated with the customer. The class allows customers to make and cancel bookings while maintaining their personal details. This class is essential for managing customer information and ensuring that each customer can book and manage their flights.



```java
public class Customer {

    private int id; // The unique identifier for 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<>(); // List of bookings made by the customer
    private boolean deleted; // Flag indicating whether the customer is deleted

    /**
     * Constructs a new Customer object with the specified parameters.
     *
     * @param id    The unique identifier for 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.
     */
    public Customer(int id, String name, String phone, String email) {
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.deleted = false;
    }

    /**
     * Returns the unique identifier for the customer.
     *
     * @return The customer ID.
     */
    public int getId() {
        return id;
    }

    /**
     * Sets the unique identifier for the customer.
     *
     * @param id The customer ID.
     */
    public void setId(int id) {
        this.id = id;
    }

    /**
     * Returns the name of the customer.
     *
     * @return The name of the customer.
     */
    public String getName() {
        return name;
    }
```

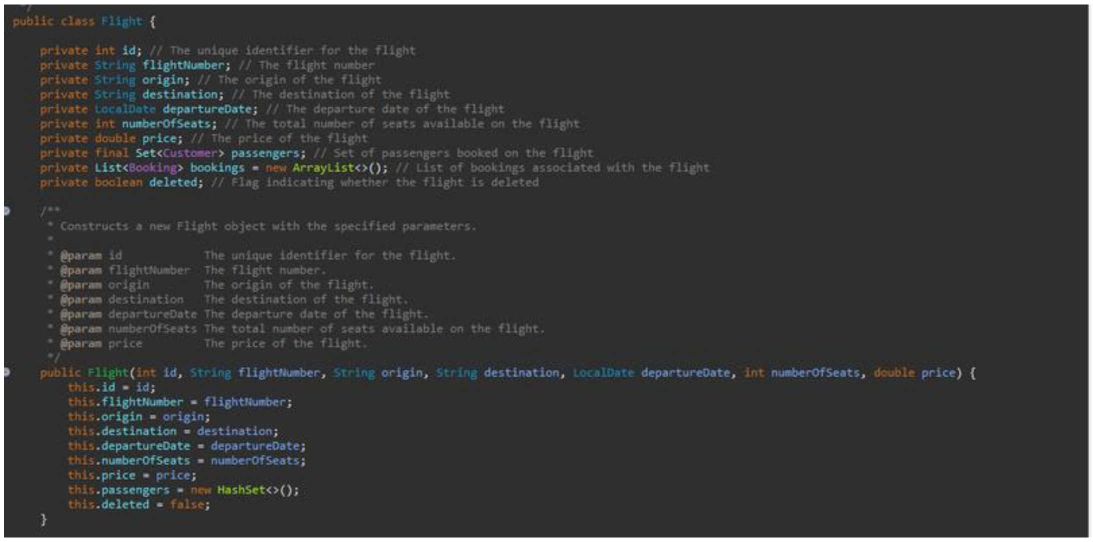


```java
public void removeBooking(Booking booking) {
    bookings.remove(booking);
}

/**
 * Returns whether the customer is deleted or not.
 *
 * @return True if the customer is deleted, false otherwise.
 */
public boolean isDeleted() {
    return deleted;
}

/**
 * Sets the deleted status of the customer.
 *
 * @param deleted True if the customer is deleted, false otherwise.
 */
public void setDeleted(boolean deleted) {
    this.deleted = deleted;
}

/**
 * Checks if any of the customer's bookings are cancelled.
 *
 * @return True if at least one booking is cancelled, false otherwise.
 */
public boolean isCancelled() {
    for (Booking booking : bookings) {
        if (booking.isCancelled()) {
            return true;
        }
    }
    return false;
}
```

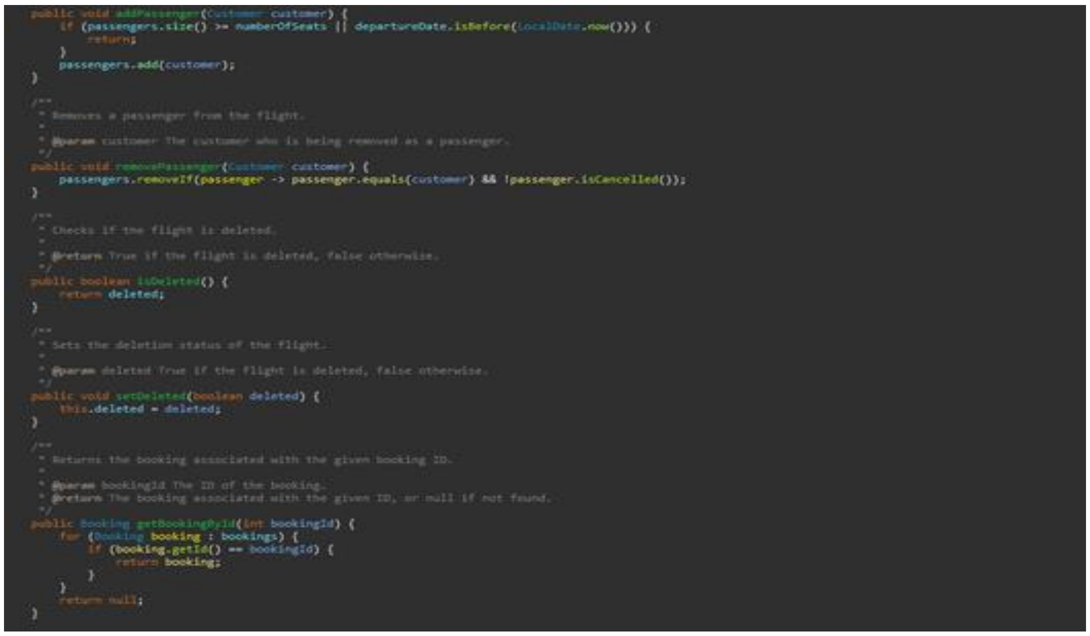
Fig: code snippet of Customer

Flight.java

The Flight class represents a flight in the system. It contains attributes such as id, destination, departureTime, and capacity, which define the flight details. The class also manages a list of Booking objects to track reservations and ensures that no more passengers are booked than the flight's capacity allows. This class ensures efficient flight management by keeping track of passenger reservations and limiting bookings to available capacity.



```java
public class Flight {

    private int id; // The unique identifier for 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 numberOfSeats; // The total number of seats available on the flight
    private double price; // The price of the flight
    private final Set<Customer> passengers; // Set of passengers booked on the flight
    private List<Booking> bookings = new ArrayList<>(); // List of bookings associated with the flight
    private boolean deleted; // Flag indicating whether the flight is deleted

    /**
     * Constructs a new Flight object with the specified parameters.
     *
     * @param id            The unique identifier for 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 numberOfSeats The total number of seats available on the flight.
     * @param price         The price of the flight.
     */
    public Flight(int id, String flightNumber, String origin, String destination, LocalDate departureDate, int numberOfSeats, double price) {
        this.id = id;
        this.flightNumber = flightNumber;
        this.origin = origin;
        this.destination = destination;
        this.departureDate = departureDate;
        this.numberOfSeats = numberOfSeats;
        this.price = price;
        this.passengers = new HashSet<>();
        this.deleted = false;
    }
```

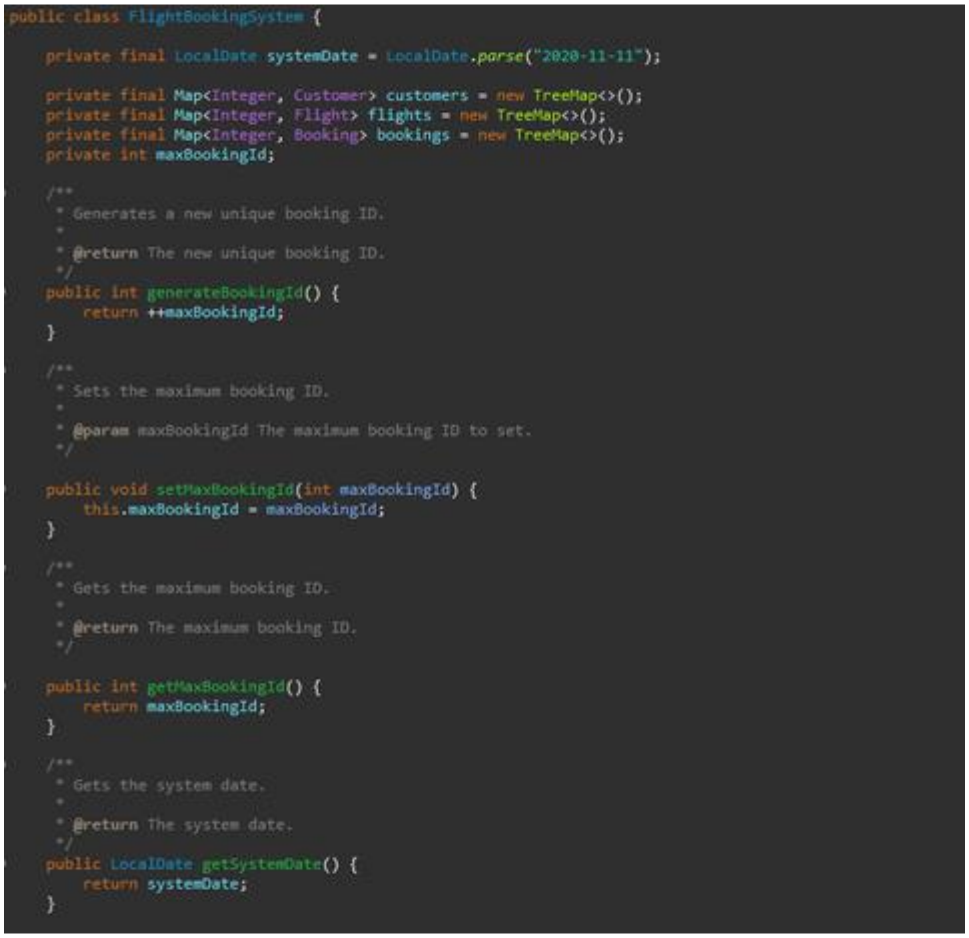


```java
    public void addPassenger(Customer customer) {
        if (passengers.size() >= numberOfSeats || departureDate.isBefore(LocalDate.now())) {
            return;
        }
        passengers.add(customer);
    }

    /**
     * Removes a passenger from the flight.
     *
     * @param customer The customer who is being removed as a passenger.
     */
    public void removePassenger(Customer customer) {
        passengers.removeIf(passenger -> passenger.equals(customer) && !passenger.isCancelled());
    }

    /**
     * Checks if the flight is deleted.
     *
     * @return True if the flight is deleted, false otherwise.
     */
    public boolean isDeleted() {
        return deleted;
    }

    /**
     * Sets the deletion status of the flight.
     *
     * @param deleted True if the flight is deleted, false otherwise.
     */
    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }

    /**
     * Returns the booking associated with the given booking ID.
     *
     * @param bookingId The ID of the booking.
     * @return The booking associated with the given ID, or null if not found.
     */
    public Booking getBookingById(int bookingId) {
        for (Booking booking : bookings) {
            if (booking.getId() == bookingId) {
                return booking;
            }
        }
        return null;
    }
```

Fig: code snippet of Flight

FlightBookingSystem.java

The FlightBookingSystem class serves as the main controller for the entire booking system. It manages a list of customers and flights, allowing new flights and customers to be added. The system also handles the booking process, ensuring that flights are not overbooked. This class acts as the heart of the booking system, ensuring smooth customer and flight management while preventing overbooking.

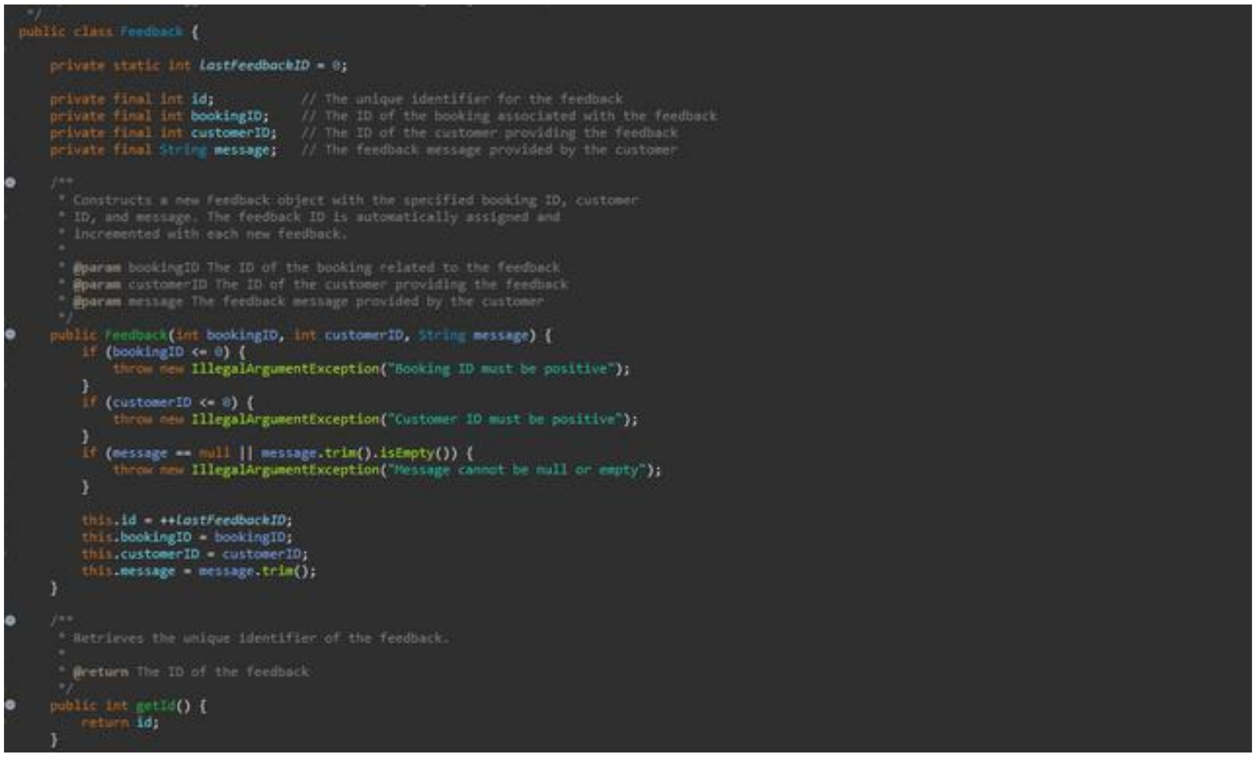


```java
public class FlightBookingSystem {

    private final LocalDate systemDate = LocalDate.parse("2020-11-11");

    private final Map<Integer, Customer> customers = new TreeMap<>();
    private final Map<Integer, Flight> flights = new TreeMap<>();
    private final Map<Integer, Booking> bookings = new TreeMap<>();
    private int maxBookingId;

    /**
     * Generates a new unique booking ID.
     *
     * @return The new unique booking ID.
     */
    public int generateBookingId() {
        return ++maxBookingId;
    }

    /**
     * Sets the maximum booking ID.
     *
     * @param maxBookingId The maximum booking ID to set.
     */

    public void setMaxBookingId(int maxBookingId) {
        this.maxBookingId = maxBookingId;
    }

    /**
     * Gets the maximum booking ID.
     *
     * @return The maximum booking ID.
     */

    public int getMaxBookingId() {
        return maxBookingId;
    }

    /**
     * Gets the system date.
     *
     * @return The system date.
     */
    public LocalDate getSystemDate() {
        return systemDate;
    }
```

Fig: code snippet of FlightBookingSystem

Feedback.java

The Feedback.java class manages customer reviews and ratings, storing structured feedback for analysis and service improvements. It encapsulates customer ID, feedback text, and timestamp while applying key object-oriented principles such as encapsulation for data protection, inheritance for reusability, abstraction to hide implementation details, and polymorphism to ensure flexible handling of feedback operations.



```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) {
        if (bookingID <= 0) {
            throw new IllegalArgumentException("Booking ID must be positive");
        }
        if (customerID <= 0) {
            throw new IllegalArgumentException("Customer ID must be positive");
        }
        if (message == null || message.trim().isEmpty()) {
            throw new IllegalArgumentException("Message cannot be null or empty");
        }

        this.id = ++lastFeedbackID;
        this.bookingID = bookingID;
        this.customerID = customerID;
        this.message = message.trim();
    }

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

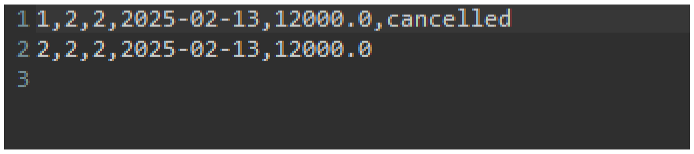
Fig: code snippet of Feedback.java

Data Storage

The Data Storage system in the Flight Booking System consists of text files that store information about bookings, customers, flights, and feedback. These files allow persistent storage and retrieval of data for system functionality.

Bookings.txt

The Bookings.txt file records all flight reservations made by customers. Each entry contains a unique booking ID, a reference to the customer ID and flight ID, and the booking status (active or canceled). This allows the system to track which customers have booked which flights and whether their reservations are still valid.

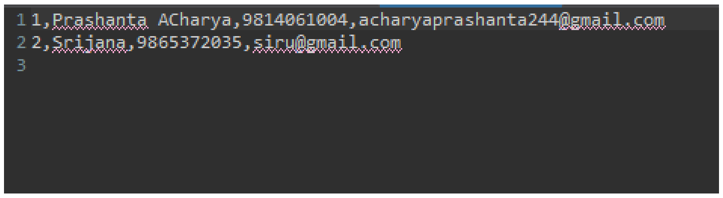


```
1,2,2,2025-02-13,12000.0,cancelled
2,2,2,2025-02-13,12000.0
```

Fig: snippet of bookings.txt

Customers.txt

The Customers.txt file stores passenger details, including their unique customer ID, name, and phone number. This file helps identify customers and associate them with their respective bookings. It ensures that the system can retrieve customer information efficiently and facilitate smooth booking management.

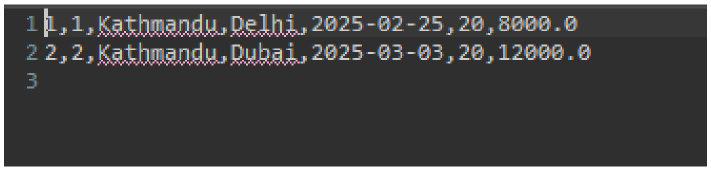


```
1,Prashanta ACharya,9814061004,acharyaprashanta244@gmail.com
2,Srijana,9865372035,siru@gmail.com
```

Fig: snippet of customers.txt

Flights.txt

The Flights.txt file contains details of all available flights in the system. Each record includes a flight ID, destination, departure time, and capacity. This ensures that the system can manage flight schedules, check seat availability, and prevent overbooking.

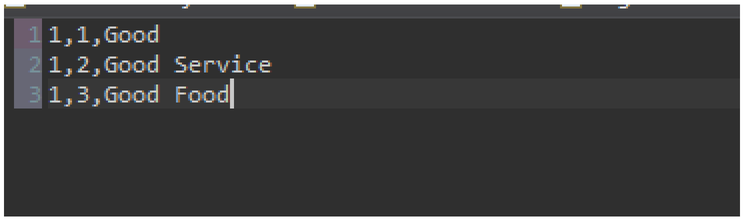


```
1,1,Kathmandu,Delhi,2025-02-25,20,8000.0
2,2,Kathmandu,Dubai,2025-03-03,20,12000.0
```

Fig: snippet of flights.txt

Feedback.txt

The Feedbacks.txt file is used to store customer reviews and feedback about their flight experience. This file helps track customer satisfaction and improve services based on passenger input. Each entry in the file typically includes the Customer ID, Flight ID, and the Feedback message provided by the customer. This ensures that feedback is linked to both the customer and the specific flight they reviewed.



```
1,1,Good
1,2,Good Service
1,3,Good Food
```

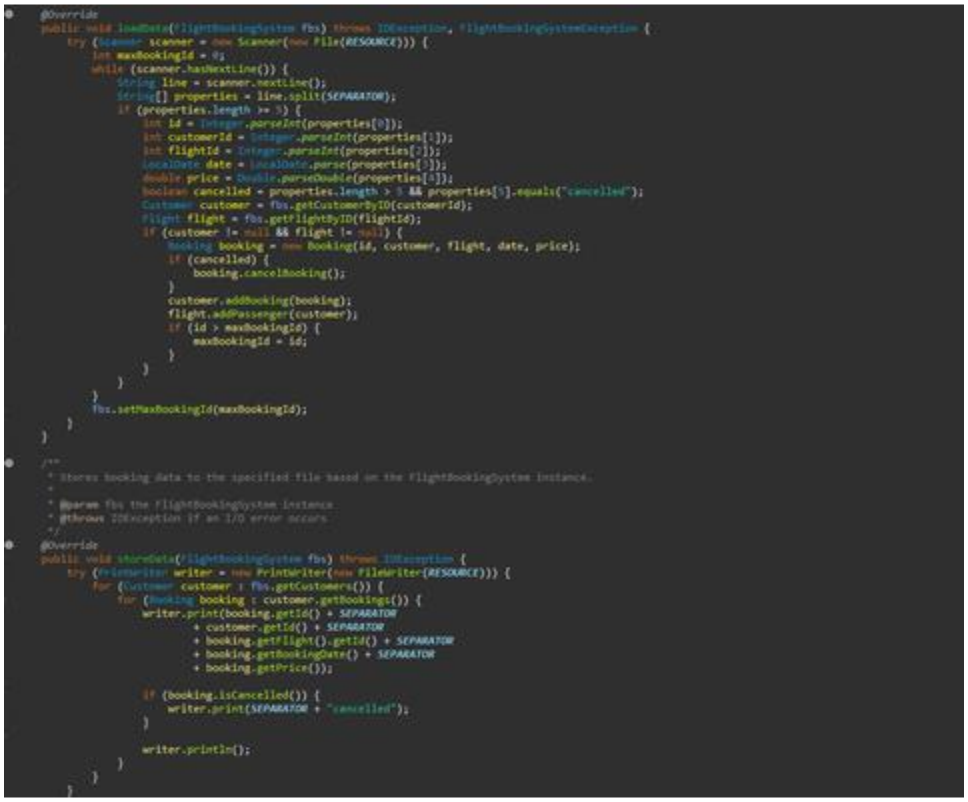
Fig: snippet of feedback.txt

Data Management System

The Data Management System in the Flight Booking System is responsible for handling data storage and retrieval from text files. It consists of multiple manager classes: BookingDataManager.java, CustomerDataManager.java, FlightDataManager.java, DataManager.java, FlightBookingSystemData.java and FeedbackDataManager.java. These classes ensure that bookings, customers, flights, and feedback data are correctly read from and written to their respective files.

BookingDataManager.java

The BookingDataManager.java class manages all operations related to bookings. It reads booking data from Bookings.txt and loads it into the system. It also provides functionality to add new bookings and update their status. This class ensures that bookings are properly linked with customers and flights.

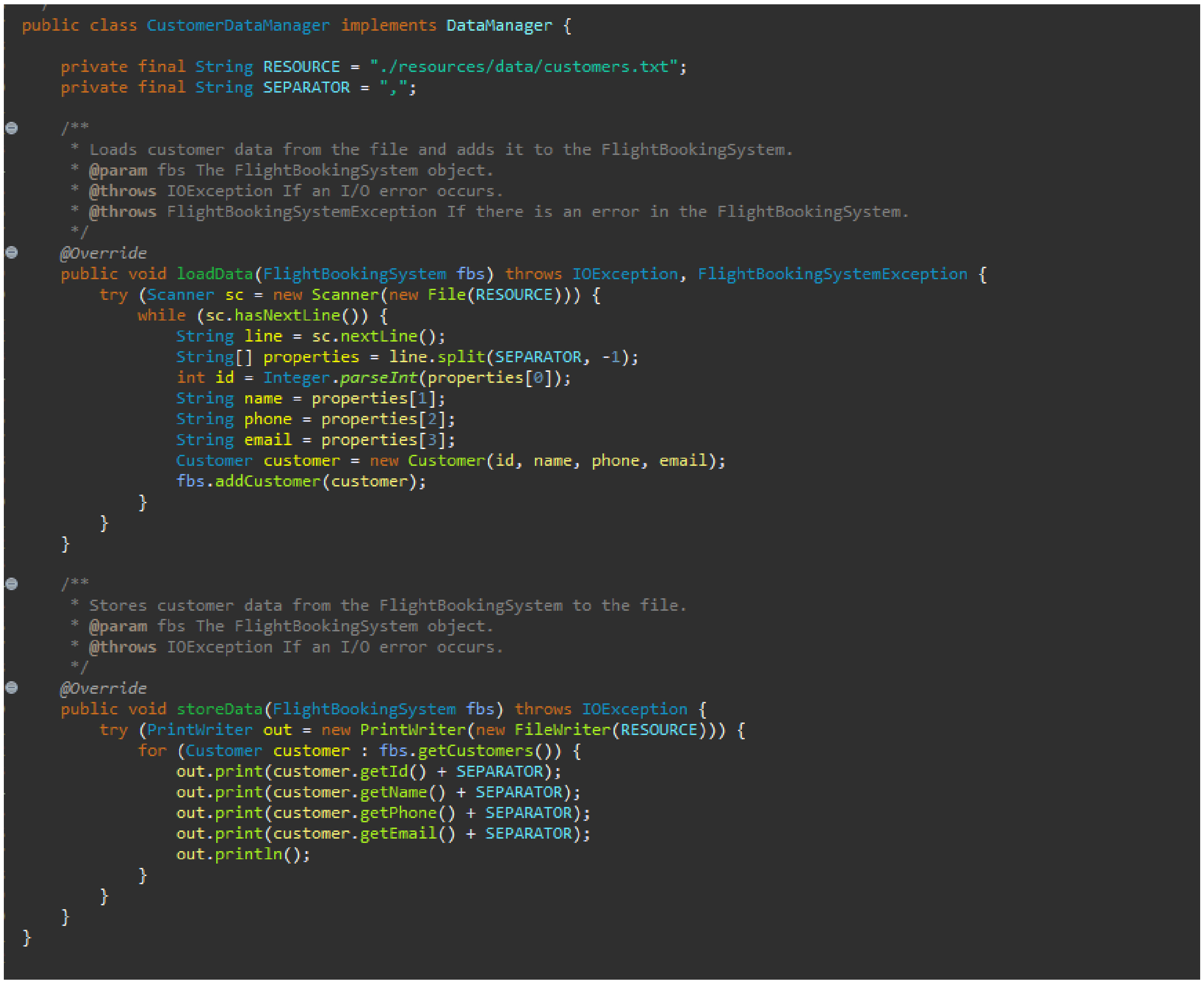


```java
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner scanner = new Scanner(new File(RESOURCE))) {
            int maxBookingId = 0;
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] properties = line.split(SEPARATOR);
                if (properties.length >= 5) {
                    int id = Integer.parseInt(properties[0]);
                    int customerId = Integer.parseInt(properties[1]);
                    int flightId = Integer.parseInt(properties[2]);
                    LocalDate date = LocalDate.parse(properties[3]);
                    double price = Double.parseDouble(properties[4]);
                    boolean cancelled = properties.length > 5 && properties[5].equals("cancelled");
                    Customer customer = fbs.getCustomerByID(customerId);
                    Flight flight = fbs.getFlightByID(flightId);
                    if (customer != null && flight != null) {
                        Booking booking = new Booking(id, customer, flight, date, price);
                        if (cancelled) {
                            booking.cancelBooking();
                        }
                        customer.addBooking(booking);
                        flight.addPassenger(customer);
                        if (id > maxBookingId) {
                            maxBookingId = id;
                        }
                    }
                }
            }
            fbs.setMaxBookingId(maxBookingId);
        }
    }

    /**
     * Stores booking data to the specified file based on the FlightBookingSystem instance.
     *
     * @param fbs the FlightBookingSystem instance
     * @throws IOException if an I/O error occurs
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter writer = new PrintWriter(new FileWriter(RESOURCE))) {
            for (Customer customer : fbs.getCustomers()) {
                for (Booking booking : customer.getBookings()) {
                    writer.print(booking.getId() + SEPARATOR
                            + customer.getId() + SEPARATOR
                            + booking.getFlight().getId() + SEPARATOR
                            + booking.getBookingDate() + SEPARATOR
                            + booking.getPrice());

                    if (booking.isCancelled()) {
                        writer.print(SEPARATOR + "cancelled");
                    }

                    writer.println();
                }
            }
        }
```

Fig: code snippet of BookingDataManager

CustomerDataManager.java

The CustomerDataManager.java class is responsible for loading and saving customer information to Customers.txt. It helps retrieve customer details and create new customer entries in the system. This class ensures that each customer has a unique ID and maintains their booking history.

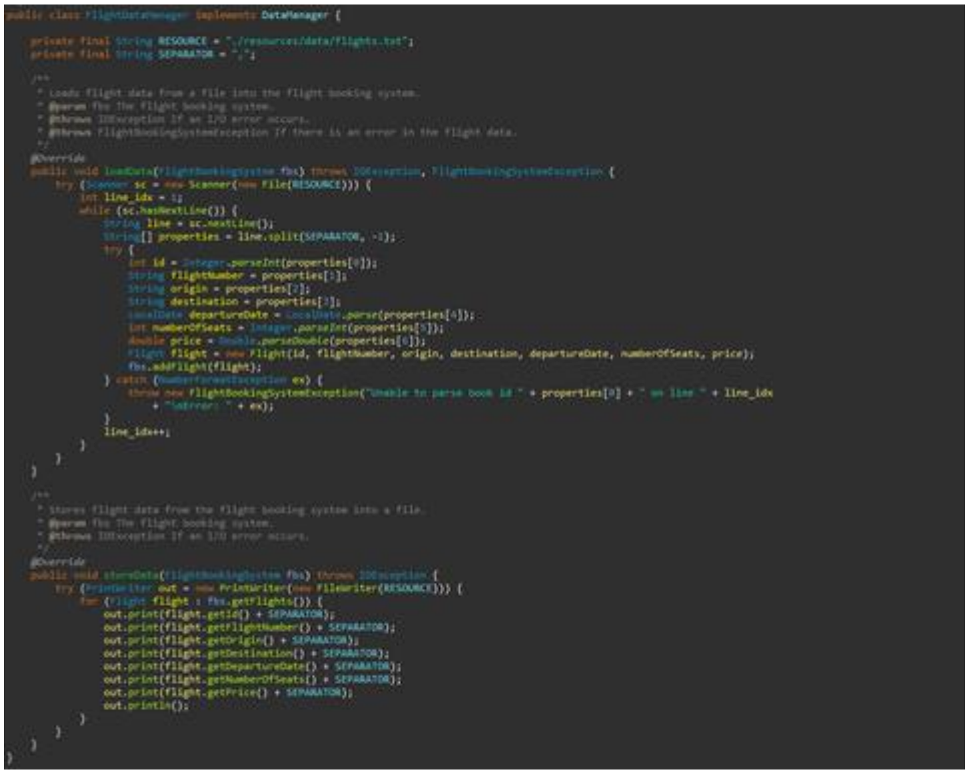


```java
public class CustomerDataManager implements DataManager {

    private final String RESOURCE = "./resources/data/customers.txt";
    private final String SEPARATOR = ",";

    /**
     * Loads customer data from the file and adds it to the FlightBookingSystem.
     * @param fbs The FlightBookingSystem object.
     * @throws IOException If an I/O error occurs.
     * @throws FlightBookingSystemException If there is an error in the FlightBookingSystem.
     */
    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        try (Scanner sc = new Scanner(new File(RESOURCE))) {
            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                String[] properties = line.split(SEPARATOR, -1);
                int id = Integer.parseInt(properties[0]);
                String name = properties[1];
                String phone = properties[2];
                String email = properties[3];
                Customer customer = new Customer(id, name, phone, email);
                fbs.addCustomer(customer);
            }
        }
    }

    /**
     * Stores customer data from the FlightBookingSystem to the file.
     * @param fbs The FlightBookingSystem object.
     * @throws IOException If an I/O error occurs.
     */
    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(RESOURCE))) {
            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.println();
            }
        }
    }
}
```

Fig: code snippet of CustomerDataManager

FlightDataManager.java

The FlightDataManager.java class handles flight data stored in Flights.txt. It loads flight schedules into the system and ensures that flight details such as destination, departure time, and seat capacity are properly managed. It also checks for flight availability before allowing new bookings.

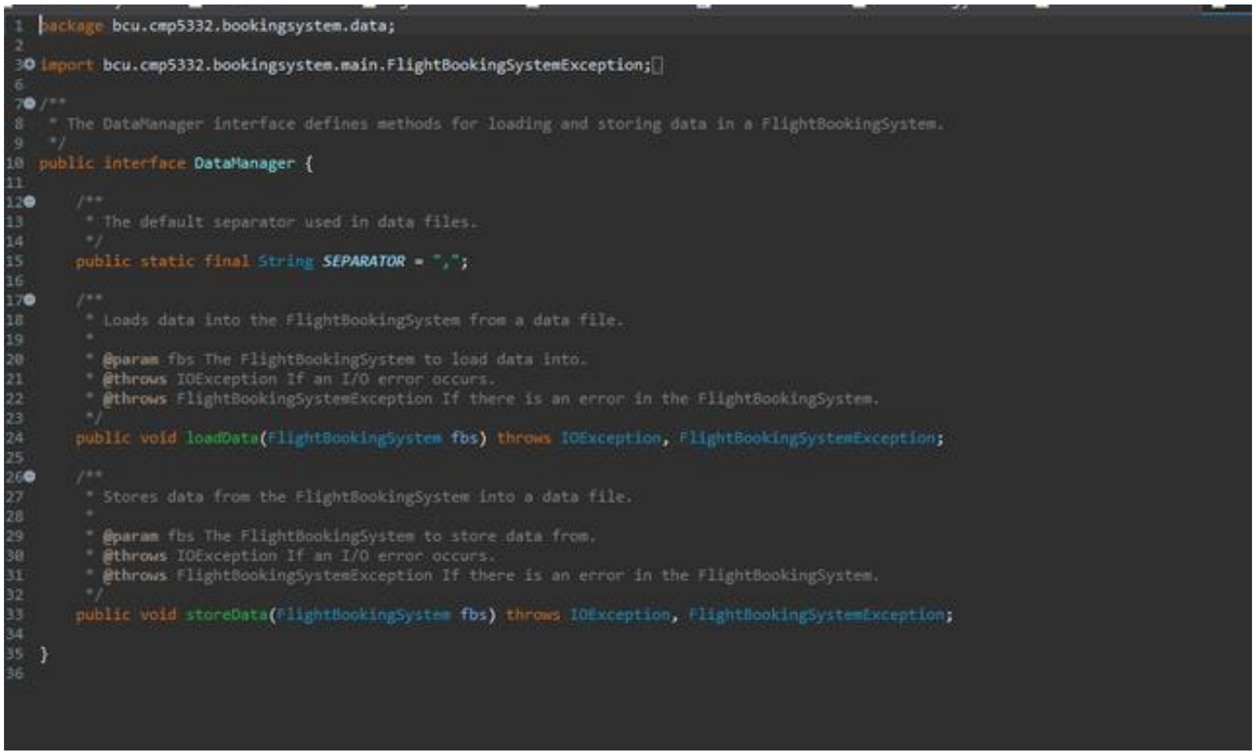


```java
public class FlightDataManager implements DataManager {

    private final String RESOURCE = "./resources/data/flights.txt";
    private final String SEPARATOR = ",";

    /**
     * Loads flight data from a file into the flight booking system.
     * @param fbs The flight booking system.
     * @throws IOException If an I/O error occurs.
     * @throws FlightBookingSystemException If there is an error in the flight data.
     */
    @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 numberOfSeats = Integer.parseInt(properties[5]);
                    double price = Double.parseDouble(properties[6]);
                    Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
                    fbs.addFlight(flight);
                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse book id " + properties[0] + " on line " + line_idx
                        + "\nError: " + ex);
                }
                line_idx++;
            }
        }
    }

    /**
     * Stores flight data from the flight booking system into a file.
     * @param fbs The flight booking system.
     * @throws IOException If an I/O error occurs.
     */
    @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.getNumberOfSeats() + SEPARATOR);
                out.print(flight.getPrice() + SEPARATOR);
                out.println();
            }
        }
    }
}
```

Fig: code snippet of FlightDataManager

DataManager.java

The DataManager interface defines the core methods required for handling data in the system, providing a contract for loading and storing data in the FlightBookingSystem. The loadData(FlightBookingSystem fbs) method is responsible for loading flight booking data from a text file, while the storeData(FlightBookingSystem fbs) method saves flight booking data to a text file, ensuring structured data management and seamless integration within the system.

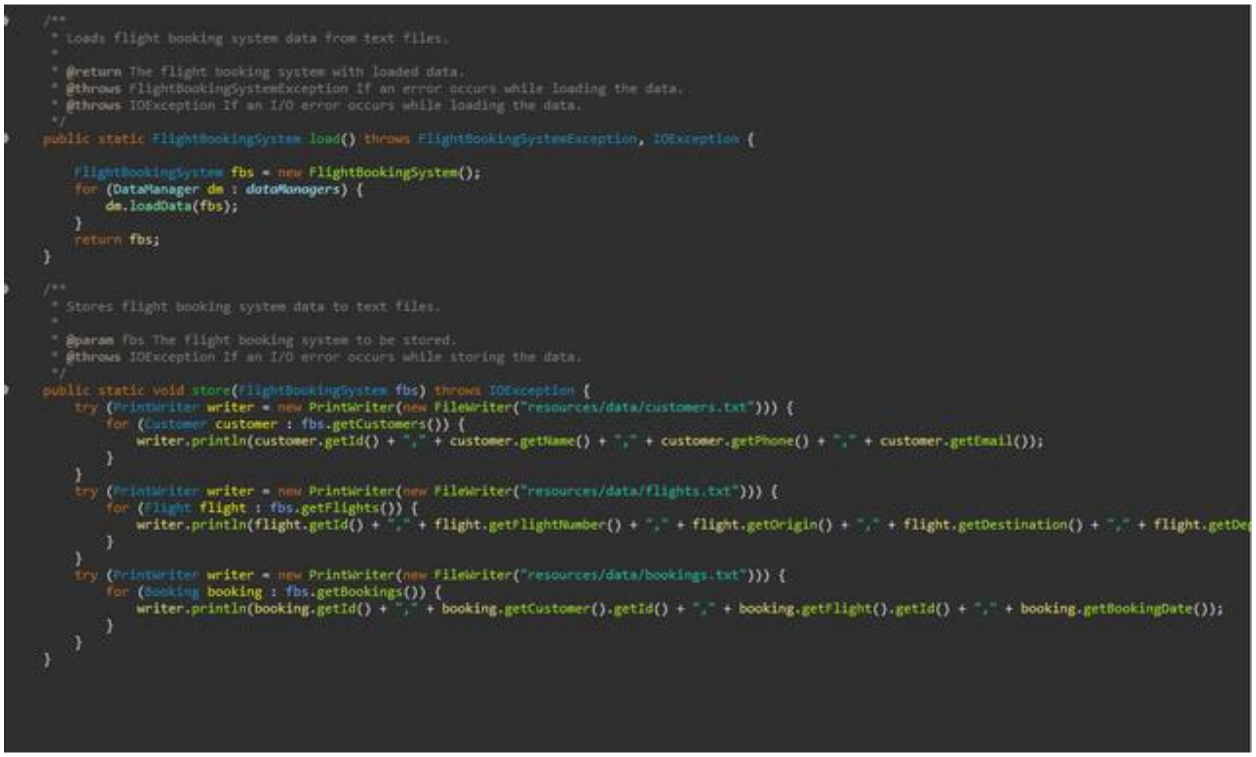


```java
package bcu.cmp5332.bookingsystem.data;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * The DataManager interface defines methods for loading and storing data in a FlightBookingSystem.
 */
public interface DataManager {

    /**
     * The default separator used in data files.
     */
    public static final String SEPARATOR = ",";

    /**
     * Loads data into the FlightBookingSystem from a data file.
     *
     * @param fbs The FlightBookingSystem to load data into.
     * @throws IOException If an I/O error occurs.
     * @throws FlightBookingSystemException If there is an error in the FlightBookingSystem.
     */
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException;

    /**
     * Stores data from the FlightBookingSystem into a data file.
     *
     * @param fbs The FlightBookingSystem to store data from.
     * @throws IOException If an I/O error occurs.
     * @throws FlightBookingSystemException If there is an error in the FlightBookingSystem.
     */
    public void storeData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException;

}
```

Fig: code snippet of DataManager

FlightBookingSystemData.java

The FlightBookingSystemData class manages the loading and storing of flight, customer, and booking data using the DataManager interface. It utilizes a list of DataManager instances to handle data operations for flights, customers, and bookings. A static block initializes this list by adding FlightDataManager, CustomerDataManager, and BookingDataManager. Additionally, the class is responsible for reading and writing customers, flight, and booking records to text files, ensuring efficient data management within the system.

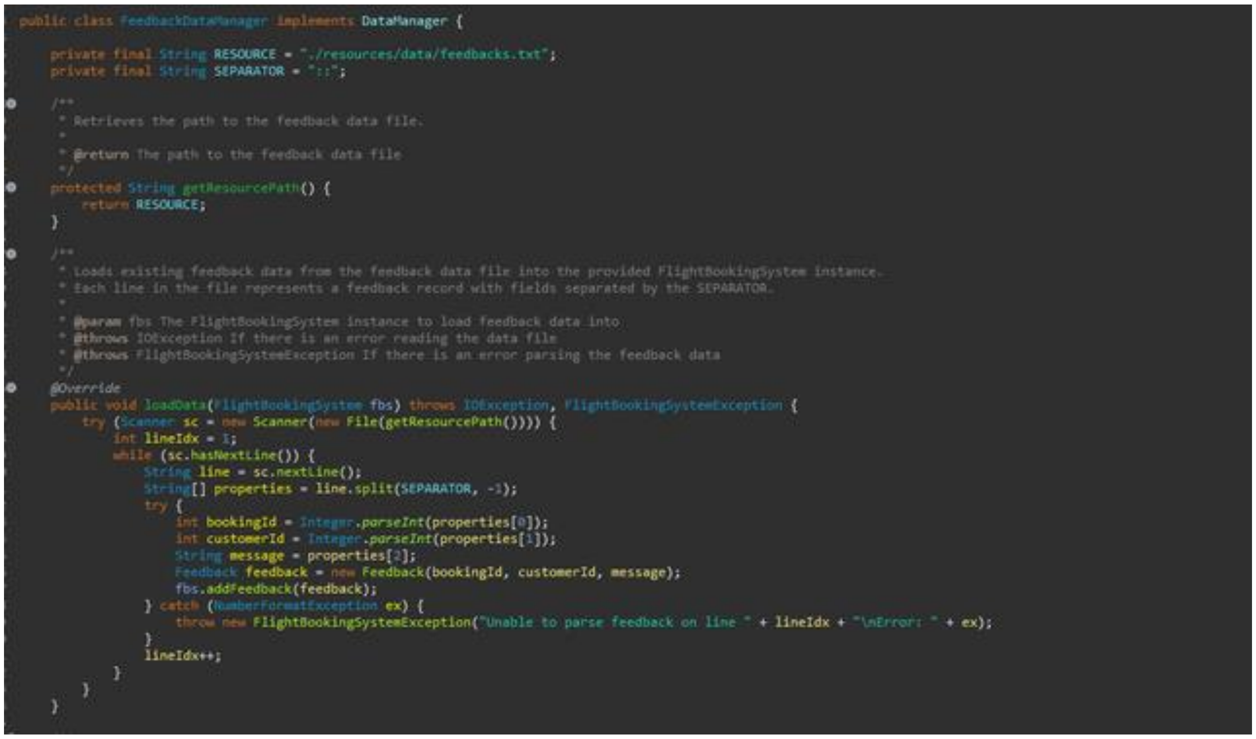


```java
/**
 * Loads flight booking system data from text files.
 *
 * @return The flight booking system with loaded data.
 * @throws FlightBookingSystemException If an error occurs while loading the data.
 * @throws IOException If an I/O error occurs while loading the data.
 */
public static FlightBookingSystem load() throws FlightBookingSystemException, IOException {

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

/**
 * Stores flight booking system data to text files.
 *
 * @param fbs The flight booking system to be stored.
 * @throws IOException If an I/O error occurs while storing the data.
 */
public static void store(FlightBookingSystem fbs) throws IOException {
    try (PrintWriter writer = new PrintWriter(new FileWriter("resources/data/customers.txt"))) {
        for (Customer customer : fbs.getCustomers()) {
            writer.println(customer.getId() + "," + customer.getName() + "," + customer.getPhone() + "," + customer.getEmail());
        }
    }
    try (PrintWriter writer = new PrintWriter(new FileWriter("resources/data/flights.txt"))) {
        for (Flight flight : fbs.getFlights()) {
            writer.println(flight.getId() + "," + flight.getFlightNumber() + "," + flight.getOrigin() + "," + flight.getDestination() + "," + flight.getDe
        }
    }
    try (PrintWriter writer = new PrintWriter(new FileWriter("resources/data/bookings.txt"))) {
        for (Booking booking : fbs.getBookings()) {
            writer.println(booking.getId() + "," + booking.getCustomer().getId() + "," + booking.getFlight().getId() + "," + booking.getBookingDate());
        }
    }
}
```

Fig:code snippet of FlightBookingSystemData

FeedbackDataManager.java

The FeedbackDataManager class extends DataManager to handle customer feedback storage and retrieval by reading and writing feedback data from and to a text file. It uses a defined separator (::) to format feedback data, ensuring proper structuring. Additionally, the class maintains data integrity by correctly linking feedback to the corresponding bookings and customers within the system.



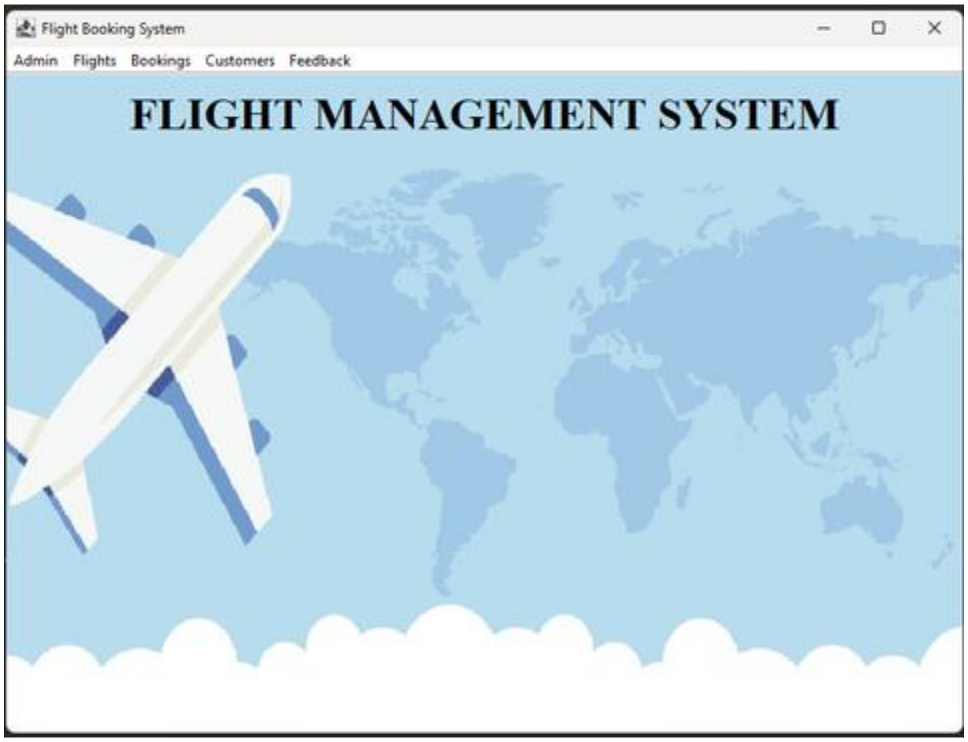
```java
public class FeedbackDataManager implements DataManager {

    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++;
            }
        }
    }
```

Fig: code snippet of FeedbackDataManager

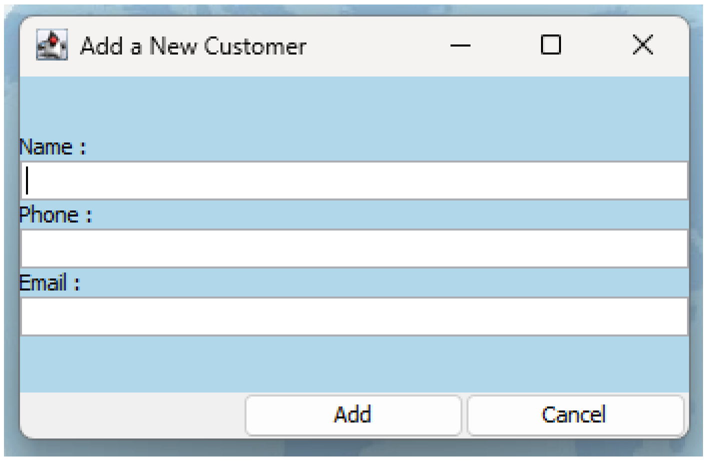
Graphical User Interface(GUI)

Landing Frame

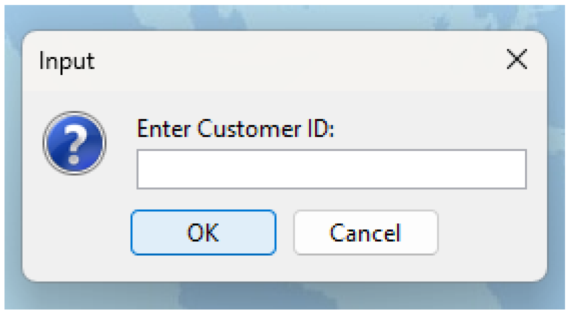
The LandingFrame in the GUI serves as the main entry point for the Flight Management System, providing users with access to various system functionalities.



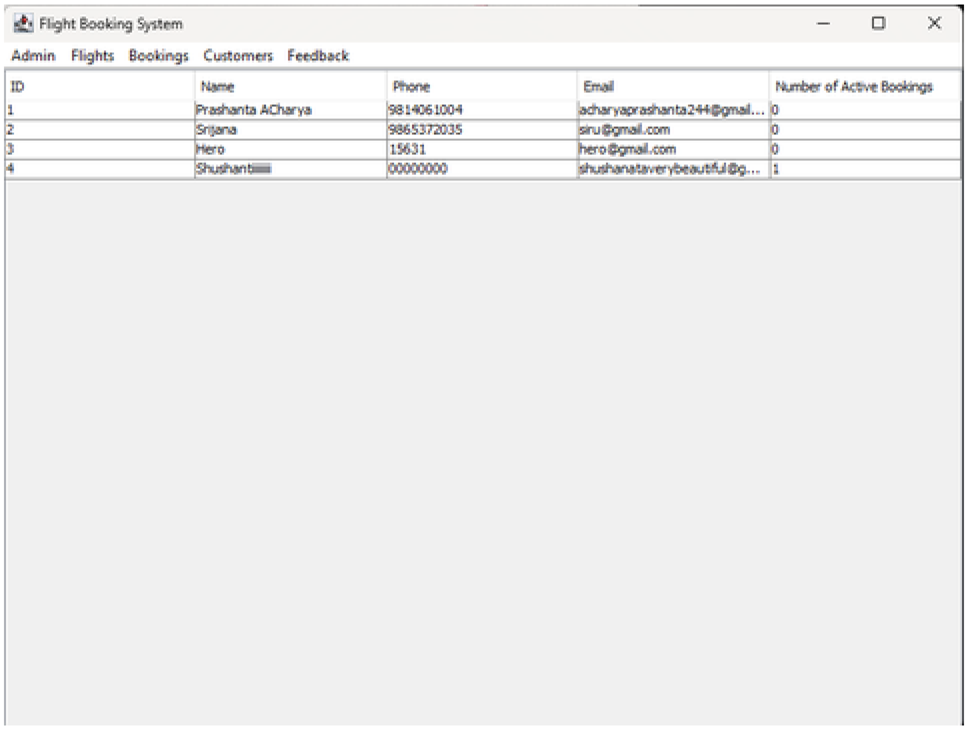
AddCustomerWindow



DeleteCustomerWindow

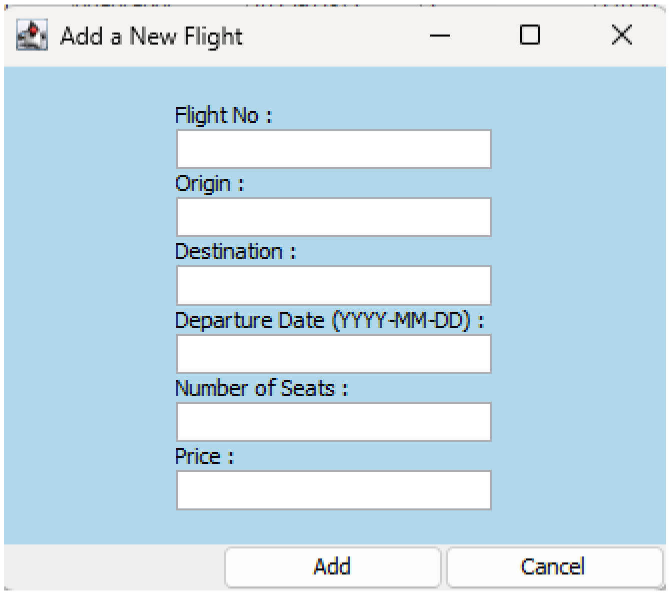


View CustomerWindow



ID	Name	Phone	Email	Number of Active Bookings
1	Prashanta ACharya	9814061004	acharyaprashanta244@gmail...	0
2	Srijana	9865372035	siru@gmail.com	0
3	Hero	15631	hero@gmail.com	0
4	Shushantiiiii	00000000	shushanataverybeautiful@g...	1

AddFlightWindow



Flight No :

Origin :

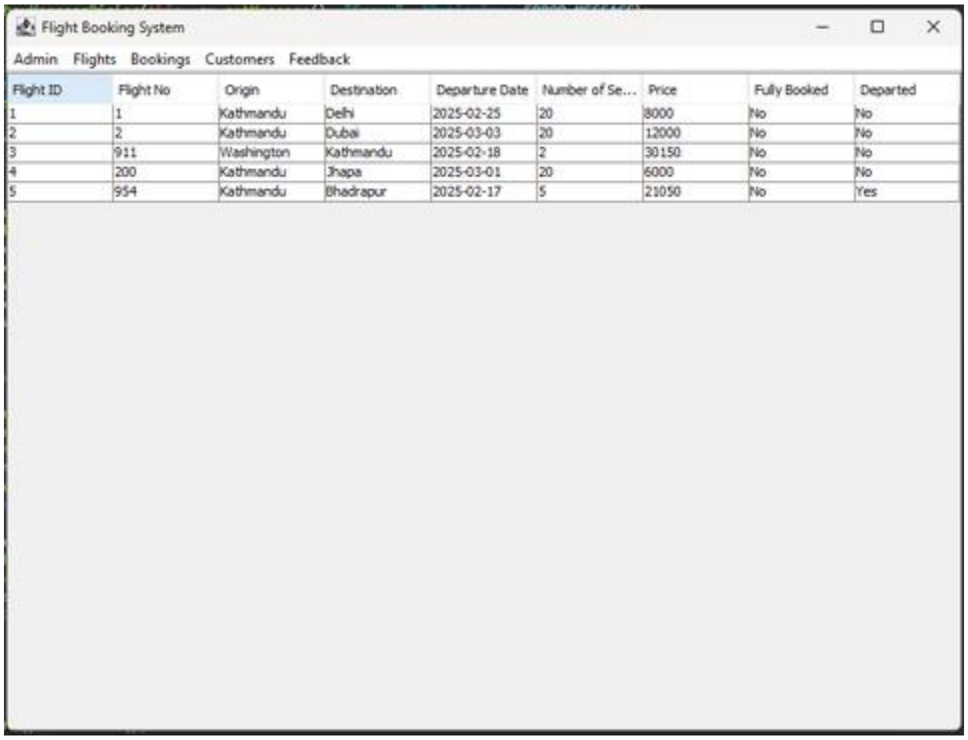
Destination :

Departure Date (YYYY-MM-DD) :

Number of Seats :

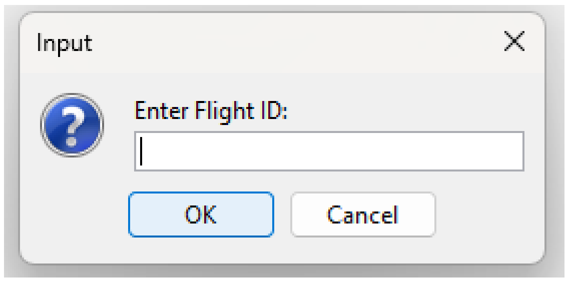
Price :

View Flights

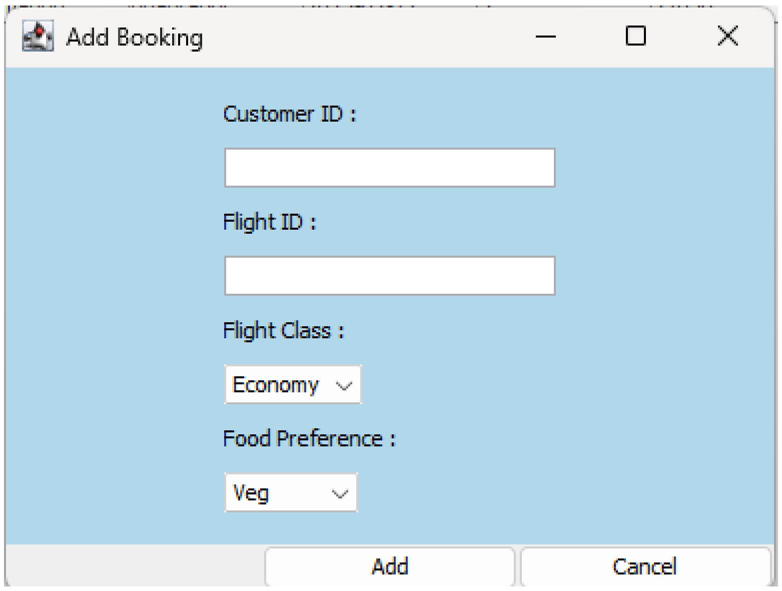


Flight ID	Flight No	Origin	Destination	Departure Date	Number of Se...	Price	Fully Booked	Departed
1	1	Kathmandu	Delhi	2025-02-25	20	8000	No	No
2	2	Kathmandu	Dubai	2025-03-03	20	12000	No	No
3	911	Washington	Kathmandu	2025-02-18	2	30150	No	No
4	200	Kathmandu	Jhapa	2025-03-01	20	6000	No	No
5	954	Kathmandu	Bhadrapur	2025-02-17	5	21050	No	Yes

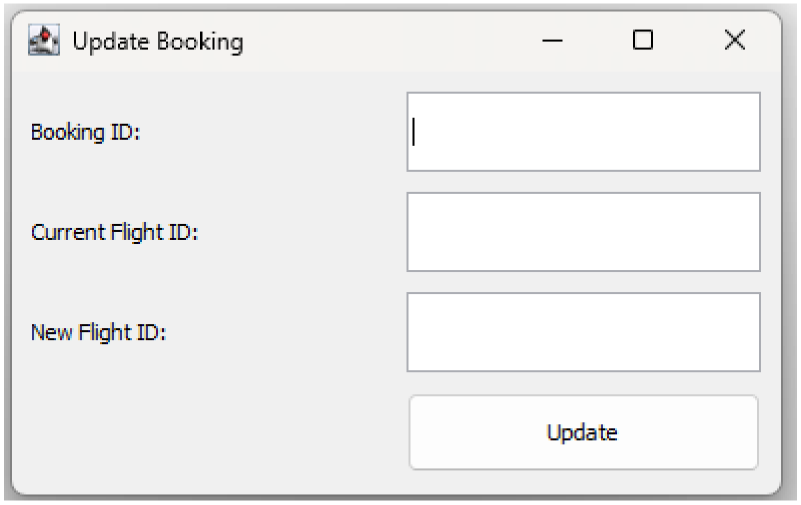
Deleteflight



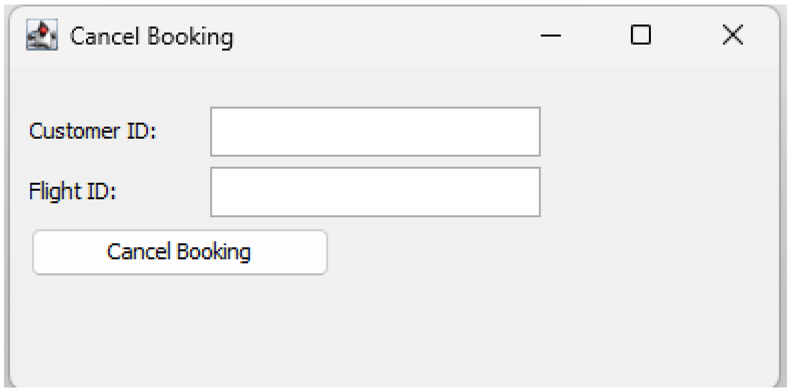
AddBooking



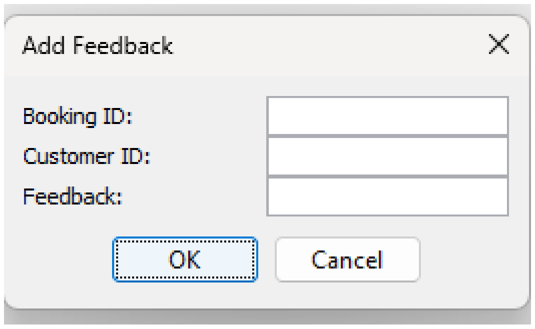
UpdateBookingWindow



Cancel Booking



AddFeedback

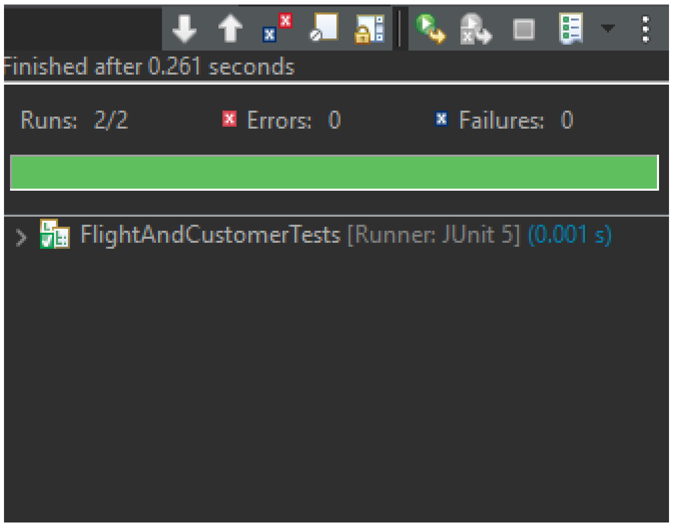

Add Feedback

Booking ID:	
Customer ID:	
Feedback:	

Testing

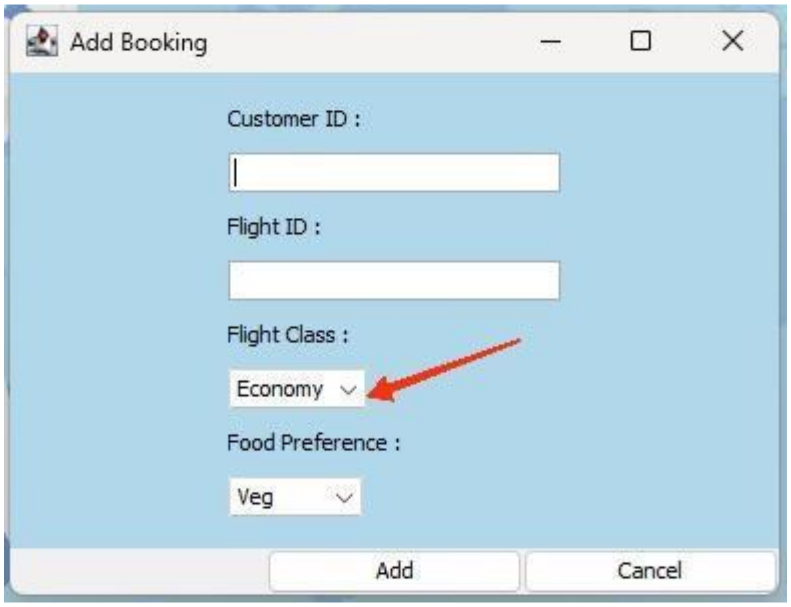
Testing in the Flight Management System includes unit testing for data handling, integration testing for component interaction, and functional testing for user operations like booking and feedback. Using JUnit for automation and manual UI validation ensures reliability, data integrity, and smooth system functionality, providing an error-free user experience.

FlightAndCustomerTest.java


```
Finished after 0.261 seconds

Runs:  2/2      Errors:  0      Failures:  0

> FlightAndCustomerTests [Runner: JUnit 5] (0.001 s)
```

Additional Features and Enhancement

Flight Class



Add Booking

Customer ID :

Flight ID :

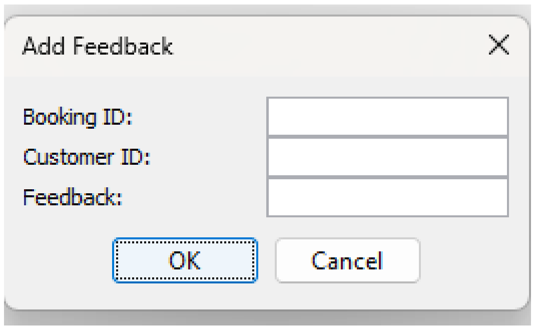
Flight Class :

Economy

Food Preference :

Veg

Feedback



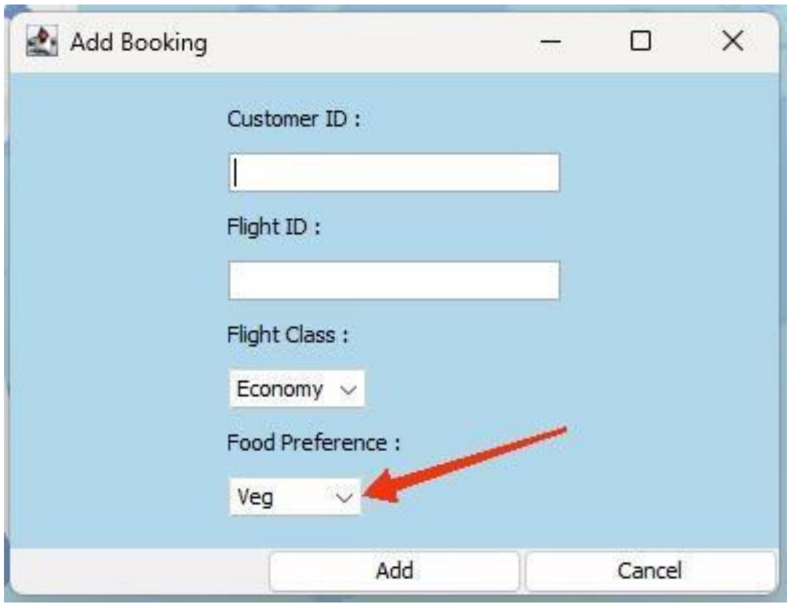
Add Feedback

Booking ID:

Customer ID:

Feedback:

Food Preference



Add Booking

Customer ID :

Flight ID :

Flight Class : Economy

Food Preference : Veg

Conclusion

The Flight Management System is a well-structured application that efficiently handles flight bookings, customer data, and feedback storage. It follows a modular design, using the DataManager interface for standardized data management. Classes like FlightBookingSystemData and FeedbackDataManager ensure seamless data loading and storage via text files. The GUI components, including LandingFrame, provide a user-friendly interface for smooth navigation. The system undergoes unit, integration, and functional testing to ensure reliability and efficiency. Overall, it offers a scalable and organized solution for managing flights, bookings, and customer interactions while maintaining data integrity and ease of use.