



BIRMINGHAM CITY
University

Author: Sujal Manandhar

Course: Object-Oriented Programming

Code: CMP5332

Technical Report: Flight Booking System

Date: 2024/06/23

Student ID: 23189654

Page Count: 36

Word Count: 4553

Table of Contents

1. [Introduction](#)
 - [1.1 Flight Booking System](#)
 - [1.2 Project Scope](#)
 - [1.3 Technologies Used](#)
 - [1.4 Project Structure, Package Organization and JUnit Testing](#)
 - [1.5 Objectives of Flight Booking System](#)
2. [Command Package](#)
 - [2.1 Key Components](#)
 - [2.2 Classes in Command Package](#)
3. [Data Manager Package](#)
 - [3.1 BookingDataManager](#)
 - [3.2 CustomerDataManager](#)
 - [3.3 DataManager Interface](#)
 - [3.4 FlightBookingSystemData](#)
 - [3.5 FlightDataManager](#)
4. [GUI Package](#)
 - [4.1 AddBookingWindow](#)
 - [4.2 AddCustomerWindow](#)
 - [4.3 AddFlightWindow](#)
 - [4.4 AddWindow](#)
 - [4.5 DeleteBookingWindow](#)
 - [4.6 DeleteFlightWindow](#)
 - [4.7 EditCustomerWindow](#)
 - [4.8 EditFlightWindow](#)
 - [4.9 LoginDialog](#)
 - [4.10 MainWindow](#)
 - [4.11 UserLoginDialog](#)
 - [4.12 UserWindow](#)
5. [Model Package](#)
 - [5.1 Booking Class](#)
 - [5.2 Customer Class](#)
 - [5.3 Flight Class](#)
 - [5.4 FlightBookingSystem Class](#)
6. [Main Package](#)

6.1 Main Class

6.2 CommandParser Class

6.3 FlightBookingSystemException Class

7. Creation Test

7.1 testFlightCreation()

7.2 testCustomerCreation()

8. Outputs of commands and GUI

Table of Figures

[Figure 1: Command Interface](#)

[Figure 2: AddFlight Class](#)

[Figure 3: AddBooking Class](#)

[Figure 4: AddCustomer Class](#)

[Figure 5: Help class](#)

[Figure 6: LoadGUI class](#)

[Figure 7: BookingDataManager Code](#)

[Figure 8:CustomerDataManager Code](#)

[Figure 9: DataManager Code](#)

[Figure 10: FlightBookingSystemData Code](#)

[Figure 11: FlightDataMananger Code](#)

[Figure 12: Class Declaration & Javadoc for Booking Class](#)

[Figure 13: Instance Variables for Booking Class](#)

[Figure 14: Constructor for Booking Class](#)

[Figure 15: Class Declaration & Javadoc for Customer Class](#)

[Figure 16: Class Declaration & Javadoc for Customer Class](#)

[Figure 17: Constructor for Customer Class](#)

[Figure 18: Constructor for Flight Class](#)

[Figure 19:FlightBookingSystemException](#)

[Figure 20: CreationTest class](#)

[Figure 21:JUnit Testing for CreationTest Class](#)

Acknowledgement

I would like to express my deep gratitude to Sunway College and BCU for giving me the invaluable opportunity to present my Flight Booking System as part of the "Object-Oriented Programming using Java" module. Throughout this module, I faced various challenges related to the Object-Oriented Project Lifecycle, each of which was effectively addressed with the unwavering support of my instructors. Their guidance was crucial in navigating these complexities, and I am extremely thankful for their assistance.

The focus on practical applications enabled me to seamlessly connect theoretical concepts with real-world scenarios, greatly enhancing my problem-solving skills. The encouragement to venture beyond the classroom and participate in extracurricular activities further expanded my understanding of the rapidly evolving landscape of object-oriented programming in Java. Reflecting on this educational journey, I am confident that the skills and knowledge I have acquired will provide a strong foundation for my future endeavours in the dynamic field of Java-based programming and flight systems.

I sincerely thank Sunway College and BCU for creating an environment that promotes learning, growth, and the development of essential skills needed for success in the world of object-oriented programming and the complexities of flight booking systems.

1. Introduction

1.1 Flight Booking System

The Flight Booking System is a versatile application designed to simplify the flight booking process for travellers while enhancing management capabilities for airlines. In today's fast-paced world, an efficient and intuitive booking system is vital for both passengers and airline companies. This software solution aims to streamline and automate the management of flights, reservations, and customer interactions within a fictional airline, providing a seamless experience for users and robust tools for administrators. By delivering an efficient and user-friendly platform, the system effectively handles various aspects of flight operations, ensuring a smooth and optimised booking process.

1.2 Project Scope

The scope of the Flight Booking System involves developing software that efficiently manages flights, customer reservations, and related transactions. The system enables customers to search for and book flights while providing real-time updates on their reservations. Simultaneously, it equips administrators with tools for overseeing flight operations, adjusting prices dynamically, and handling customer service needs.

1.3 Technologies Used

This project is developed using Java, chosen for its robustness, portability, and extensive library support. The system leverages Java's object-oriented capabilities to ensure modularity and reusability of code. Additionally, various Java libraries and frameworks, such as JavaFX for the user interface, JDBC for database connectivity, and JUnit for testing, are employed to enhance functionality, reliability, and user experience.

1.4 Project Structure, Package Organization and JUnit Testing

The Flight Booking System project is structured into several key packages, each serving a distinct purpose.

- a. **Command Package:** The **'Command'** Package contains classes that execute user actions, such as booking flights or cancelling reservations. It includes the **'CommandParser'** for parsing user input and directing commands like **'BookFlightCommand'** for booking flights, **'CancelBookingCommand'** for cancelling reservations, **'AddCustomerCommand'** for adding customers, and **'DeleteFlightCommand'** for removing flights from the system.
- b. **Data Package:** Responsible for managing bookings, customers, and flight data. It includes **'BookingDataManager'** for bookings and **'CustomerDataManager'** for customer data, with **'DataManager'** as their base. **'FlightBookingSystemData'** centrally manages system data, while **'FlightDataManager'** handles flights.

- c. **GUI Package:** Contains graphical interface components: **'AddBookingWindow'** for bookings, **'AddCustomerWindow'** for customers, and **'AddFlightWindow'** for flights. **'MainWindow'** integrates these for user interaction.
- d. **Main Package:** Centralizes application entry and execution. **'CommandParser'** parses commands, **'Main.java'** launches the app, and **'FlightBookingSystemException'** manages errors.
- e. **Model Package:** Defines core data models: **'Booking'**, **'Customer'**, and **'Flight'**. **'FlightBookingSystem.java'** integrates models and business logic.
- f. **Test Package:** Ensures system reliability with **'CreationTest'** for basic functionality and specific tests like **'BookingDataManagerTest'** and **'CustomerDataManagerTest'** for data management validation. **'JUnit testing'** is employed extensively across these test classes to automate and streamline the testing process, ensuring robustness and reliability in the flight booking system.

1.5 Objectives of Flight Booking System

The Flight Booking System aims to deliver a user-centric experience by enabling efficient booking processes and seamless reservation management through intuitive command-based interactions. It prioritises robust data management capabilities to organise bookings, customer details, and flight information effectively. The system design focuses on creating a user-friendly graphical interface that enhances usability for both customers and administrators. Ensuring reliability under various operational conditions, the system incorporates rigorous error handling mechanisms and informative feedback systems. Clear data models and essential business logic implementations facilitate accurate representation and manipulation of booking data. Thorough testing using the JUnit framework guarantees system reliability and functionality across all operational aspects, affirming its readiness for deployment in real-world scenarios.

2. Command Package

In the Command Package of the Flight Booking System, the primary focus is on encapsulating and implementing various commands that users can execute to interact with the system. These commands are essential for performing actions such as booking flights, managing customer information, and handling reservations efficiently. The Command Package typically includes classes and interfaces that facilitate command execution, parsing user input, and delegating tasks to appropriate components within the system.

2.1 Key Components

a. Command Interface or Base Class:

- Defines the structure and behaviour of all commands within the system.
- Provides methods for executing commands and handling parameters.

b. CommandParser Class:

- Responsible for parsing user input and identifying the corresponding command to execute.
- Decodes user commands into actionable tasks for the system.

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

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public interface Command {

    public static final String HELP_MESSAGE = "Commands:\n"
        + "\tlistflights          print all flights\n"
        + "\tlistbookings             print all bookings\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"
        + "\tcustomerbookinglist [customer id] show customer's booking list\n"
        + "\tdeleteflight [flight id] delete flight details\n"
        + "\tdeletecustomer [customer id] delete customer details\n"
        + "\tremainingseats [flight id] show flight's remaining seats details\n"
        + "\teditcustomer [customer id] edit customer details\n"
        + "\taddbooking [customer id] [flight id] add a new booking\n"
        + "\tcancelbooking [customer id] [flight id] cancel a booking\n"
        + "\teditbooking [customer 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";

    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException;
}
```

Figure 1: Command Interface

2.2 Classes in Command Package

2.2.1 Flight Commands

a. EditFlightCommand:

- Purpose: Modifies details of an existing flight.
- Functionality: Allows changes to flight schedules, destinations, or seat availability.

b. ListFlightsCommand:

- Purpose: Retrieves and displays a list of all available flights.
- Functionality: Shows flight details such as departure times, destinations, and

remaining seat availability.

c. GetRemainingSeatsCommand:

- Purpose: Retrieves and displays the number of remaining seats available on a specific flight.
- Functionality: Provides real-time information on seat availability for booking.

d. ShowFlightsCommand:

- Purpose: Displays detailed information about specific flights.
- Functionality: Shows flight details including passenger bookings and remaining seats.

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

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class AddFlight implements Command {

    private final String flightNumber;
    private final String origin;
    private final String destination;
    private final LocalDate departureDate;
    private int numberOfSeats;
    private int price;

    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;
    }

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        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, numberOfSeats, price);
        flightBookingSystem.addFlight(flight);
        System.out.println("Flight #" + flight.getId() + " added.");
    }
}
```

Figure 2: AddFlight Class

2.2.2 Booking Commands

a. BookFlightCommand:

- Purpose: Handles the process of booking a flight for a customer.
- Functionality: Manages seat reservations, updates availability, and records booking details.

b. CancelBookingCommand:

- Purpose: Executes the cancellation of a flight reservation.
- Functionality: Updates seat availability, removes booking records, and manages customer refunds if applicable.

c. EditBookingCommand:

- Purpose: Modifies details of an existing booking.
- Functionality: Allows changes to flight selection, passenger information, or booking specifics.

d. ListBookingsCommand:

- Purpose: Retrieves and displays a list of all bookings in the system.
- Functionality: Shows details such as booking IDs, passenger names, and flight information.

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

import java.time.LocalDate;

public class AddBooking implements Command {

    private final int cusId;
    private final int flightId;
    private final LocalDate bookingDate;

    public AddBooking(int cusId, int flightId, LocalDate bookingDate) {
        this.cusId = cusId;
        this.flightId = flightId;
        this.bookingDate = bookingDate;
    }

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        flightBookingSystem.addBookingByIds(cusId, flightId, bookingDate);
    }
}
```

Figure 3: AddBooking Class

2.2.3 Customer Commands

a. EditCustomerCommand:

- Purpose: Updates customer information.
- Functionality: Allows modifications to customer details such as contact information or frequent flyer status.

b. ListCustomersCommand:

- Purpose: Displays a list of all customers registered in the system.
- Functionality: Provides customer details including names and contact information.

c. ShowCustomersCommand:

- Purpose: Displays detailed information about specific customers.
- Functionality: Shows customer details including booking history and contact information.

```

package bcu.cmp5332.bookingsystem.commands;

import java.util.ArrayList;

public class AddCustomer implements Command {

    private final String name;
    private final String phone;
    private final String email;
    private final List<Booking> bookings;

    public AddCustomer(String name, String phone, String email, List<Booking> bookings) {

        this.name = name;
        this.phone = phone;
        this.bookings = new ArrayList<>();
        this.email = email;
    }

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        int maxId = 0;
        if (flightBookingSystem.getCustomers().size() > 0) {
            int lastIndex = flightBookingSystem.getCustomers().size() - 1;
            maxId = flightBookingSystem.getCustomers().get(lastIndex).getId();
        }
        // TODO: implementation here
        Customer customer = new Customer(++maxId, name, phone, email, bookings);
        flightBookingSystem.addCustomer(customer);
    }
}

```

Figure 4: AddCustomer Class

2.2.4 General Commands

a. HelpCommand:

- Purpose: Provides assistance and displays information on available commands.
- Functionality: Offers guidance on how to use the system and its functionalities.

```

package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.model.FlightBookingSystem;

public class Help implements Command {

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) {
        System.out.println(Command.HELP_MESSAGE);
    }
}

```

Figure 5: Help class

b. LoadGUICommand:

- Purpose: Loads the graphical user interface (GUI) for the Flight Booking System.
- Functionality: Initialises the GUI components for user interaction and system navigation.

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

import bcu.cmp5332.bookingsystem.model.FlightBookingSystem;

public class LoadGUI implements Command {

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        new MainWindow(flightBookingSystem);
    }

};
```

Figure 6: LoadGUI class

3. Data Manager Package

The Data Manager package in the flight booking system is crucial for handling data persistence and retrieval. This package includes classes that manage the loading and storing of data for bookings, customers, and flights. By implementing a standardised interface, these classes ensure consistent and reliable data management, enabling the flight booking system to maintain accurate and up-to-date information across different sessions. The primary classes in this package are ‘**BookingDataManager**’, ‘**CustomerDataManager**’, ‘**FlightDataManager**’, and ‘**FlightBookingSystemData**’, each responsible for specific data types, contributing to the seamless operation of the entire system.

3.1 BookingDataManager

- Purpose:** Manages the loading and storing of booking data in the flight booking system.
- Fields:**
 - ‘**RESOURCE**’: The file path where booking data is stored.
- Methods:**
 - ‘**loadData(FlightBookingSystem fbs)**’: Reads booking data from the file and populates the ‘FlightBookingSystem’ object with ‘Booking’ objects.
 - ‘**storeData(FlightBookingSystem fbs)**’: Writes the current bookings from the ‘FlightBookingSystem’ object to the file.

```

package bcu.cmp5332.bookingsystem.data;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class BookingDataManager implements DataManager {

    public final String RESOURCE = "resources/data/bookings.txt";

    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        // TODO: implementation here
        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 name = properties[1];
                    String phone = properties[2];
                    String email = properties[3];
                    List<Booking> bookings = new ArrayList<>();
                    Customer customer = new Customer(id, name, phone, email, bookings);

                    int flightId = Integer.parseInt(properties[4]);
                    String flightNumber = properties[5];
                    String origin = properties[6];
                    String destination = properties[7];
                    LocalDate departureDate = LocalDate.parse(properties[8]);
                    int numberOfSeats = Integer.parseInt(properties[9]);
                    int price = Integer.parseInt(properties[10]);
                    Flight flight = new Flight(flightId, flightNumber, origin, destination, departureDate, numberOfSeats, price);

                    LocalDate bookingDate = LocalDate.parse(properties[11]);

                    Booking booking = new Booking(customer, flight, bookingDate);
                    fbs.addBooking(booking);

```

```

                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse book id " + properties[0] + " on line " + line_idx
                        + "\nError: " + ex);
                }
                line_idx++;
            }
        }
    }

    @Override
    public void storeData(FlightBookingSystem fbs) throws IOException {
        try (PrintWriter out = new PrintWriter(new FileWriter(RESOURCE))) {
            for (Booking booking : fbs.getBookings()) {
                Customer customer = booking.getCustomer();
                out.print(customer.getId() + SEPARATOR);
                out.print(customer.getName() + SEPARATOR);
                out.print(customer.getPhone() + SEPARATOR);
                out.print(customer.getEmail() + SEPARATOR);

                Flight flight = booking.getFlight();
                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.print(booking.getBookingDate() + SEPARATOR);
                out.println();
            }
        }
    }
}

```

Figure 7: BookingDataManager Code

3.2 CustomerDataManager

- a. **Purpose:** Manages the loading and storing of customer data in the flight booking

system.

b. Fields:

- **‘RESOURCE’**: The file path where customer data is stored.

c. Methods:

- **‘loadData(FlightBookingSystem fbs)’**: Reads customer data from the file and populates the ‘FlightBookingSystem’ object with ‘Customer’ objects.
- **‘storeData(FlightBookingSystem fbs)’**: Writes the current customers from the ‘FlightBookingSystem’ object to the file.

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

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class CustomerDataManager implements DataManager {

    public final String RESOURCE = "resources/data/customers.txt";

    @Override
    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException {
        // TODO: implementation here
        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 name = properties[1];
                    String phone = properties[2];
                    String email = properties[3];
                    List<Booking> bookings = new ArrayList<>();

                    Customer customer = new Customer(id, name, phone, email, bookings);
                    fbs.addCustomer(customer);
                } catch (NumberFormatException ex) {
                    throw new FlightBookingSystemException("Unable to parse book id " + properties[0] + " on line " + line_idx
                        + "\nError: " + ex);
                }
                line_idx++;
            }
        }
    }
}
```

```
@Override
public void storeData(FlightBookingSystem fbs) throws IOException {
    // TODO: implementation here
    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();
        }
    }
}
```

Figure 8:CustomerDataManager Code

3.3 DataManager Interface

- a. **Purpose:** Defines the contract for data management operations.
- b. **Fields:**
 - **‘SEPARATOR’:** A delimiter used to separate fields in the data files.
- c. **Methods:**
 - **‘loadData(FlightBookingSystem fbs)’:** Abstract method to be implemented for loading data into the ‘FlightBookingSystem’.
 - **‘storeData(FlightBookingSystem fbs)’:** Abstract method to be implemented for storing data from the ‘FlightBookingSystem’.

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

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public interface DataManager {

    public static final String SEPARATOR = "::";

    public void loadData(FlightBookingSystem fbs) throws IOException, FlightBookingSystemException;
    public void storeData(FlightBookingSystem fbs) throws IOException;

}
```

Figure 9: DataManager Code

3.4 FlightBookingSystemData

- a. **Purpose:** Manages the loading and storing of all data related to the flight booking system by coordinating multiple data managers.
- b. **Fields:**
 - **‘dataManagers’:** A list of ‘DataManager’ instances for different types of data (‘flights’, ‘customers’, ‘bookings’).
- c. **Methods:**
 - **‘load()’:** Creates a new ‘FlightBookingSystem’ object and populates it by calling ‘loadData()’ on each ‘DataManager’.
 - **‘store(FlightBookingSystem fbs)’:** Persists the current state of the ‘FlightBookingSystem’ by calling ‘storeData()’ on each ‘DataManager’.

```

package bcu.cmp5332.bookingsystem.data;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

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());
    }

    public static FlightBookingSystem load() throws FlightBookingSystemException, IOException {

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

    public static void store(FlightBookingSystem fbs) throws IOException {

        for (DataManager dm : dataManagers) {
            dm.storeData(fbs);
        }
    }
}

```

Figure 10: FlightBookingSystemData Code

3.5 FlightDataManager

- a. **Purpose:** Manages the loading and storing of flight data in the flight booking system.
- b. **Fields:**
 - **‘RESOURCE’:** The file path where flight data is stored.
- c. **Methods:**
 - **‘loadData(FlightBookingSystem fbs)’:** Reads flight data from the file and populates the ‘FlightBookingSystem’ object with ‘Flight’ objects.
 - **‘storeData(FlightBookingSystem fbs)’:** Writes the current flights from the ‘FlightBookingSystem’ object to the file.


```

package bcu.cmp5332.bookingsystem.data;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class FlightDataManager implements DataManager {

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

    @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]);
                    int price = Integer.parseInt(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++;
            }
        }
    }
}

```

```

    @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();
            }
        }
    }
}

```

Figure 11: FlightDataManager Code

4. GUI Package

4.1 AddBookingWindow

- a. **Purpose:** Provides a graphical interface for adding new bookings to the flight booking system. Extends JFrame and implements ActionListener to manage

button clicks.

b. Fields:

- JFrame parentFrame: The parent frame of this window.
- JTextField custIdField: Field to input customer ID.
- JTextField flightIdField: Field to input flight ID.
- JTextField bookingDateField: Field to input booking date.
- JButton addBtn: Button to add the booking.
- JButton cancelBtn: Button to cancel and close the window.

c. Constructor:

- AddBookingWindow(JFrame parentFrame): Initialises the window with the specified parent frame, sets up the layout and components via initialise(), and makes the window visible.

4.2 AddCustomerWindow

a. Purpose: Provides a graphical interface for adding new customers to the flight booking system. Extends JFrame and implements ActionListener to manage button clicks.

b. Fields:

- JFrame parentFrame: The parent frame of this window.
- JTextField custNameField: Field to input customer's name.
- JTextField phoneNumField: Field to input customer's phone number.
- JTextField emailField: Field to input customer's email.
- JButton addBtn: Button to add the customer.
- JButton cancelBtn: Button to cancel and close the window.

c. Constructor:

- AddCustomerWindow(JFrame parentFrame): Initialises the window with the specified parent frame, sets up the layout and components via initialize(), and makes the window visible.

4.3 AddFlightWindow

a. Purpose: Provides a graphical interface for adding new flights to the flight booking system. Extends JFrame and implements ActionListener to manage button clicks.

b. Fields:

- JFrame parentFrame: The parent frame of this window.
- JTextField flightNoText: Field to input flight number.
- JTextField originText: Field to input origin.
- JTextField destinationText: Field to input destination.
- JTextField depDateText: Field to input departure date.

- JTextField numberOfSeats: Field to input number of seats.
- JTextField price: Field to input price.
- JButton addBtn: Button to add the flight.
- JButton cancelBtn: Button to cancel and close the window.

c. Constructor:

- AddFlightWindow(JFrame parentFrame): Initialises the window with the specified parent frame.

d. Methods:

- initialize(): Sets up the layout and components.
- actionPerformed(ActionEvent ae): Handles button clicks.
- addFlight(): Adds a new flight based on the input fields.

4.4 AddWindow

- a. Purpose:** Provides an administrative GUI for managing the flight booking system, including viewing, adding, deleting, editing, and searching for flights, managing bookings, and handling customer details.

b. Fields:

- JMenuBar menuBar: The main menu bar. JMenu adminMenu, flightsMenu, bookingsMenu, customersMenu: Menus for various actions.
- JMenuItem adminExit, flightsView, flightsAdd, flightsDel, flightsEdit, flightsSearch, bookingsAdd, bookingsIssue, bookingsUpdate, bookingsCancel, custView, custAdd, custDel, custEdit: Menu items for actions within the menus.
- FlightBookingSystem fbs: The flight booking system instance.

c. Constructor:

- AdminWindow(FlightBookingSystem fbs): Initialises the window with the specified flight booking system instance.

d. Methods:

- getFlightBookingSystem(): Sets up the layout and components.
- actionPerformed(ActionEvent ae): Handles menu item clicks.
- displayFlights(): Displays a table of flights.
- displayCustomers(): Displays a table of customers.
- displayTableInFrame(JTable table): Displays a JTable in a new frame.

4.5 DeleteBookingWindow

- a. Purpose:** Provides a GUI for deleting bookings in the flight booking system.

b. Fields:

- JFrame parentFrame: The parent frame of the window.
- JTextField custIdField: Field to input customer ID.

- JTextField flightIdField: Field to input flight ID.
 - JButton confirmBtn: Button to confirm deletion.
 - JButton cancelBtn: Button to cancel and close the window.
- c. Constructor:**
- DeleteBookingWindow(JFrame parentFrame): Initialises the window with the specified parent frame and sets up the contents via initialize().
- d. Methods:**
- initialize(): Sets up the layout and components of the window.
 - actionPerformed(ActionEvent ae): Handles button clicks.
 - deleteBooking(): Executes the booking deletion.

4.6 DeleteFlightWindow

- a. Purpose:** Provides a GUI for administrators to delete flights from the flight booking system.
- b. Fields:**
- JFrame parentFrame: The parent frame of the window.
 - JTextField flightIdField: Field to input flight ID.
 - JButton confirmBtn: Button to confirm flight deletion.
- c. Constructor:**
- DeleteFlightWindow(JFrame parentFrame): Initializes the window with the specified parent frame and sets up the contents via initialize().
- d. Methods:**
- initialize(): Sets up the layout and components of the window.
 - actionPerformed(ActionEvent ae): Handles button clicks.
 - deleteFlight(): Executes the flight deletion.

4.7 EditCustomerWindow

- a. Purpose:** Provides a GUI for editing customer details in the flight booking system.
- b. Fields:**
- JFrame parentFrame: The parent frame of the window.
 - JTextField flightIdText: Field to input flight ID.
 - JTextField nameText: Field to input new name.
 - JTextField phoneText: Field to input new phone number.
 - JTextField emailText: Field to input new email address.
 - JButton editBtn: Button to confirm changes.
 - JButton cancelBtn: Button to cancel and close the window.
- c. Constructor:**
- EditCustomerWindow(JFrame parentFrame): Initializes the window with the specified parent frame and sets up the contents via initialize().
- d. Methods:**

- `initialize()`: Sets up the layout and components of the window.
- `actionPerformed(ActionEvent ae)`: Handles button clicks.
- `editCustomer()`: Executes the editing of customer details.

4.8 EditFlightWindow

- Purpose:** Provides a GUI for editing flight details in the flight booking system.
- Fields:**
 - `JFrame parentFrame`: The parent frame of the window.
 - `TextField flightIdText`: Field to input flight ID.
 - `TextField numberOfSeatsText`: Field to input number of seats.
 - `TextField priceText`: Field to input flight price.
 - `Button editBtn`: Button to confirm changes.
 - `Button cancelBtn`: Button to cancel and close the window.
- Constructor:**
 - `EditFlightWindow(JFrame parentFrame)`: Initialises the window with the specified parent frame and sets up the contents via `initialize()`.
- Methods:**
 - `initialize()`: Sets up the layout and components of the window.
 - `actionPerformed(ActionEvent ae)`: Handles button clicks.
 - `editFlight()`: Executes the editing of flight details.

4.9 LoginDialog

- Purpose:** Provides a GUI for administrators to log into the flight booking system, featuring fields for username and password, and login and cancel buttons. Includes hardcoded admin credentials for authentication.
- Fields:**
 - `TextField usernameField`: Field to input username.
 - `JPasswordField passwordField`: Field to input password.
 - `Button loginButton`: Button to initiate login.
 - `Button cancelButton`: Button to cancel login and close the dialog.
 - `boolean isAuthenticated`: Flag indicating whether the user is authenticated.
 - `User adminUser`: Hardcoded admin user with username "admin" and password "1234".
- Constructor:**
 - `LoginDialog(JFrame parent)`: Initializes the dialog with the specified parent frame and sets up the contents via `initialize()`.
- Methods:**
 - `initialize()`: Sets up the layout and components of the dialog.
 - `actionPerformed(ActionEvent e)`: Handles button clicks.

- `isAuthenticated()`: Returns the authentication status.

4.10 MainWindow

- Purpose:** Provides the main GUI interface for the flight booking system, including menus for admin and user logins, and options to view flights and customers.
- Fields:**
 - `JMenuBar menuBar`: The main menu bar.
 - `JMenu adminMenu, flightsMenu, customersMenu`: Menus for various actions.
 - `JMenuItem adminLogin, userLogin, adminExit`: Menu items for admin login, user login, and exiting the application.
 - `FlightBookingSystem fbs`: The core flight booking system instance.
 - `LoginDialog loginDialog`: Dialog for admin login.
 - `AdminWindow adminWindow`: The admin window.
 - `UserLoginDialog userLoginDialog`: Dialog for user login.
 - `UserWindow userWindow`: The user window.
- Constructor:**
 - `MainWindow(FlightBookingSystem fbs)`: Initialises the main window with the provided flight booking system.
- Methods:**
 - `getFlightBookingSystem()`: Returns the flight booking system instance.
 - `initialize()`: Sets up the GUI components and layout.
 - `actionPerformed(ActionEvent ae)`: Handles menu item actions.
 - `showLoginDialog()`: Displays the admin login dialog.
 - `openAdminWindow()`: Opens the admin window if authenticated.

4.11 UserLoginDialog

- Purpose:** Provides a GUI for users to log into the flight booking system with their username and password. This dialog includes fields for user credentials and login/cancel buttons.
- Fields:**
 - `TextField usernameField`: Field to input username.
 - `PasswordField passwordField`: Field to input password.
 - `Button loginButton`: Button to initiate login.
 - `Button cancelButton`: Button to cancel login and close the dialog.
 - `boolean isAuthenticated`: Flag indicating whether the user is authenticated.
- Constructor:**
 - `UserLoginDialog(JFrame parent)`: Initializes the dialog with the

specified parent frame and sets up the contents via initialize().

i. **Methods:**

- initialize(): Sets up the layout and components of the dialog.
- actionPerformed(ActionEvent e): Handles button clicks.
- isAuthenticated(): Returns the authentication status.

4.12 UserWindow

a. **Purpose:** Provides the user interface for viewing available flights and managing user-specific tasks in the flight booking system.

b. **Fields:**

- JFrame parentFrame: The parent frame of the window.
- JMenuBar menuBar: The main menu bar.
- JMenu flightsMenu, accountMenu: Menus for flight-related and account-related actions.
- JMenuItem viewFlights, manageBookings, accountDetails, logout: Menu items for viewing flights, managing bookings, viewing account details, and logging out.

c. **Constructor:**

- UserWindow(JFrame parentFrame): Initializes the window with the specified parent frame and sets up the contents via initialize().

d. **Methods:**

- initialize(): Sets up the layout and components of the window.
- actionPerformed(ActionEvent ae): Handles menu item clicks.
- displayFlights(): Displays available flights in a new window.
- manageBookings(): Opens a window to manage user bookings.
- viewAccountDetails(): Displays user account details in a new window.

5. Model Package

The model package within the Flight Booking System forms the fundamental framework for handling essential data structures and operational rules of the application. It acts as the central hub for overseeing flights, customers, bookings, and related entities. This package is structured to enhance clarity, facilitate modular design, and ensure ease of maintenance throughout the system.

5.1 Booking Class

Java class Booking that represents a booking made by a customer for a flight in a flight booking system.

5.1.1 Class Declaration and Javadoc

```
/**
 * Represents a booking made by a customer for a flight.
 *
 * This class encapsulates information about the customer making the booking,
 * the flight being booked, and the date when the booking was made.
 *
 * @author Sujal Manandhar
 */
public class Booking {
```

Figure 12: Class Declaration & Javadoc for Booking Class

Booking class represents a booking made by a customer for a flight. The class includes Javadoc comments describing its purpose, attributes, and authorship.

5.1.2 Instance Variables

```
private Customer customer; // The customer making the booking
private Flight flight; // The flight being booked
private LocalDate bookingDate; // The date when the booking was made
```

Figure 13: Instance Variables for Booking Class

- **customer:** Represents the customer who made the booking. It's of type Customer, which is assumed to be another class.
- **flight:** Represents the flight that is being booked. It's of type Flight, another assumed class.
- **bookingDate:** Represents the date when the booking was made. It's of type LocalDate from the Java date-time API.

5.1.3 Constructor

```
/**
 * Constructs a new Booking object with the specified customer, flight, and booking date.
 *
 * @param customer The customer making the booking
 * @param flight The flight being booked
 * @param bookingDate The date when the booking was made
 */
public Booking(Customer customer, Flight flight, LocalDate bookingDate) {
    this.customer = customer;
    this.flight = flight;
    this.bookingDate = bookingDate;
}
```


Figure 14: Constructor for Booking Class

The constructor initialises a Booking object with the provided customer, flight, and bookingDate.

5.1.4 Getter and Setter Methods

- **Getter for customer:** Retrieves the customer associated with the booking.
- **Setter for customer:** Sets a new customer associated with the booking.
- **Getter for flight:** Retrieves the flight associated with the booking.
- **Setter for flight:** Sets a new flight associated with the booking.
- **Getter for bookingDate:** Retrieves the date when the booking was made.
- **Setter for bookingDate:** Sets a new date for when the booking was made.

5.2 Customer Class

The Customer class is part of the bcu.cmp5332.bookingsystem.model package. It imports necessary classes from the Java standard library: DateTimeFormatter for date formatting, ArrayList and List for managing collections.

5.2.1 Class Declaration and Javadoc

```
package bcu.cmp5332.bookingsystem.model;

import java.time.format.DateTimeFormatter;

/**
 * Represents a customer of the flight booking system.
 *
 * This class encapsulates information about a customer including their ID, name,
 * phone number, email, and a list of bookings made by the customer.
 * It provides methods to retrieve and modify customer details, manage bookings,
 * and generate short and long-form textual representations of customer details.
 *
 * @author Sujal Manandhar
 */
public class Customer {
```

Figure 15: Class Declaration & Javadoc for Customer Class

Customer class represents a customer in a flight booking system. Javadoc comments describe its purpose, attributes, and functionality. The author tag identifies the author of this class.

5.2.2 Instance Variables

```
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<>(); // List of bookings made by the customer
```

Figure 16: Instance Variables for Customer Class

- **id, name, phone, email:** Basic attributes of the Customer.
- **bookings:** A list to store bookings made by this customer (Booking objects).

5.2.3 Constructor

```
/**
 * 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 bookings The list of bookings made by the customer
 */
public Customer(int id, String name, String phone, String email, List<Booking> bookings) {
    this.id = id;
    this.name = name;
    this.phone = phone;
    this.email = email;
    this.bookings.addAll(bookings); // Add all provided bookings to the customer's list
}
```

Figure 17: Constructor for Customer Class

Initialises a Customer object with provided id, name, phone, email, and bookings.

5.2.4 Getter and Setter Methods

- **Getters:** getId(), getName(), getPhone(), getEmail(), getBookings().
- **Setters:** setId(int newId), setName(String newName), setPhone(String newPhone), setEmail(String email).
- **Additional Methods:** addBooking(Booking booking), removeBooking(Booking booking).

5.2.5 Other Methods

- **getDetailsShort():** Generates a short textual representation of the customer's details.
- **getDetailsLong():** Generates a detailed textual representation of the customer's details, including booked flights. It uses DateTimeFormatter to format the date of each booked flight.

5.3 Flight Class

The Flight class encapsulates all the necessary information and functionality related to a flight within a booking system.

5.3.1 Constructors

Flight(int id, String flightNumber, String origin, String destination, LocalDate departureDate, int numberOfSeats, int price): Initialises a new Flight object with the provided attributes. Sets up the passengers set and initialises the remaining number of seats to the total number of seats.

```
/**
 * 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 airport
 * @param destination The destination airport
 * @param departureDate The departure date of the flight
 * @param numberOfSeats The total number of seats available on the flight
 * @param price The price per seat
 */
```

Figure 18: Constructor for Flight Class

5.3.2 Methods

- **Getters and Setters:** Provide access to private fields (getId(), getFlightNumber(), getOrigin(), etc.) and allow modification where necessary (setFlightNumber(String flightNumber), setNumberOfSeats(int numberOfSeats)).

5.4 FlightBookingSystem Class

The FlightBookingSystem class manages customers, flights, and bookings through various methods that handle adding, retrieving, updating, and deleting operations.

5.4.1 Key Components

- a. System Date Management:**
 - **systemDate:** Represents the current system date used by the booking system.

b. Data Structures:

- customers: A TreeMap to store customers.
- flights: A TreeMap to store flights.
- bookings: An ArrayList to store bookings.

c. Constructor:

- FlightBookingSystem(): Initializes the booking system.

d. Methods:

- Customers Management:
 1. addCustomer(Customer customer): Adds a new customer to the system.
 2. getCustomerByID(int id): Retrieves a customer by ID.
 3. updateCustomer(int id, String name, String phoneNum, String email): Updates customer details.
 4. deleteCustomer(int id): Deletes a customer from the system.
- Flights Management:
 1. addFlight(Flight flight): Adds a new flight to the system
 2. getFlightByID(int id): Retrieves a flight by ID.
 3. updateFlight(int id, int numOfSeats, int price): Updates flight details.
 4. deleteFlight(int id): Deletes a flight from the system.
- Booking Management:
 1. addBooking(Booking booking): Adds a new booking to the system.
 2. addBookingByIds(int cusId, int flightId, LocalDate bookingDate): Adds a booking based on customer ID, flight ID, and booking date.
 3. cancelBooking(int cusId, int flightId): Cancels a booking.
 4. updateBooking(int cusId, int flightId, int new_flightId, LocalDate newbookingDate): Updates an existing booking.
 5. getBookingByCusIDs(int id): Retrieves bookings associated with a specific customer ID.
 6. bookingListByCustomer(int cusId): Retrieves bookings associated with a specific customer ID.
 7. getBookings(): Retrieves all bookings in the system.
 8. listOfPassengersByFlight(int id): Retrieves passengers booked on a specific flight.

6. Main Package

6.1 Main Class

a. Purpose:

The **'Main'** class serves as the entry point for the Flight Booking System application. It initialises system components, manages user interaction, and controls the overall execution flow of the program.

b. Functionality:

- System Initialization: Loads initial system data using **'FlightBookingSystemData.load()'**.
- User Interface: Displays a command prompt for user input and continuously prompts for commands until termination.
- Command Execution: Utilises **'CommandParser'** to parse user input and execute corresponding commands.
- Exception Handling: Catches and displays **'FlightBookingSystemException'** messages for user-friendly error handling.
- Data Persistence: Stores updated system data using **'FlightBookingSystemData.store()'** before exiting the application.

6.2 CommandParser Class

a. Purpose:

The **'CommandParser'** class interprets user input to instantiate appropriate command objects based on predefined commands and their parameters. It facilitates interaction between users and the system by parsing commands and managing input/output operations.

b. Functionality:

- Parsing Commands: Splits user input into components to identify the command and its parameters.
- Command Execution: Instantiates specific command objects (e.g., **'AddFlight'**, **'ListBookings'**) based on user input.
- Interactive Input Handling: Manages user interaction through console input to gather necessary data for executing commands.
- Exception Handling: Catches and throws **'FlightBookingSystemException'** for invalid commands or input errors.

6. 3 FlightBookingSystemException Class

a. Purpose:

The '**FlightBookingSystemException**' class extends Java's Exception class to define a custom exception specific to the Flight Booking System. It provides a mechanism to handle and propagate errors or exceptional conditions encountered during system operation.

b. Functionality:

- Custom Error Messaging: Allows customization of error messages to provide clear feedback to users or log files.
- Error Propagation: Throws exceptions to signal errors in various parts of the system, such as command parsing or data manipulation.
- Enhanced Debugging: Aids in debugging by pinpointing issues related to command execution or system state.

```
package bcu.cmp5332.bookingsystem.main;

/**
 * FlightBookingSystemException extends {@link Exception} class and is a custom exception
 * that is used to notify the user about errors or invalid commands.
 */
public class FlightBookingSystemException extends Exception {

    public FlightBookingSystemException(String message) {
        super(message);
    }
}
```

Figure 19:FlightBookingSystemException

7. Creation Test

The '**CreationTest**' class is a unit test suite designed to verify the creation and properties of essential model classes in the Flight Booking System.

7.1 testFlightCreation()

a. **Objective:** Tests the creation and properties of the Flight class.

b. **Steps:**

- Defines parameters for a flight ('id', 'flightNumber', 'origin', 'destination', 'departureDate', 'numberOfSeats', 'price').
- Creates a new '**Flight**' object using these parameters.
- Asserts that the created '**Flight**' object is not null (assertNotNull).

- Validates that the attributes of the Flight object match the expected values (assertEquals assertions for 'id', 'flightNumber', 'origin', 'destination', 'departureDate', 'numberOfSeats', 'price').

7.2 testCustomerCreation()

- Objective:** Tests the creation and properties of the 'Customer' class.
- Steps:**
 - Defines parameters for a customer ('id', 'name', 'phoneNumber', 'email').
 - Creates a new 'Customer' object using these parameters.
 - Asserts that the created 'Customer' object is not null (assertNotNull).
 - Validates that the attributes of the Customer object match the expected values (assertEquals assertions for 'id', 'name', 'phoneNumber', 'email').

```
public class CreationTest {
    @Test
    public void testFlightCreation() {

        int id = 1;
        String flightNumber = "FL123";
        String origin = "London";
        String destination = "New York";
        LocalDate departureDate = LocalDate.now().plusDays(7);
        int numberOfSeats = 200;
        int price = 500;

        Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
        assertNotNull(flight);
        assertEquals(id, flight.getId());
        assertEquals(flightNumber, flight.getFlightNumber());
        assertEquals(origin, flight.getOrigin());
        assertEquals(destination, flight.getDestination());
        assertEquals(departureDate, flight.getDepartureDate());
        assertEquals(numberOfSeats, flight.getNumberOfSeats());
        assertEquals(price, flight.getPrice(), 0.001);
    }

    @Test
    public void testCustomerCreation() {
        int id = 1;
        String name = "John Doe";
        String phoneNumber = "1234567890";
        String email = "john.doe@example.com";
        List<Booking> bookings = new ArrayList<>();
        Customer customer = new Customer(id, name, phoneNumber, email, bookings );
        assertNotNull(customer);
        assertEquals(id, customer.getId());
        assertEquals(name, customer.getName());
        assertEquals(phoneNumber, customer.getPhone());
        assertEquals(email, customer.getEmail());
    }
}
```

Figure 20: CreationTest class

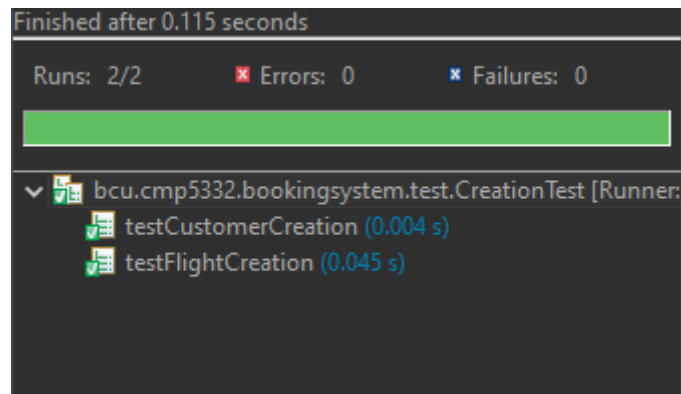


Figure 21:JUnit Testing for CreationTest Class

8. Outputs of commands and GUI

1. help command

```
Enter 'help' to see a list of available commands.
> help
Commands:
    listflights           print all flights
    listbookings          print all bookings
    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
    customerbookinglist [customer id] show customer's booking list
    deleteflight [flight id] delete flight details
    deletecustomer [customer id] delete customer details
    remainingseats [flight id] show flight's remaining seats details
    editcustomer [customer id] edit customer details
    addbooking [customer id] [flight id] add a new booking
    cancelbooking [customer id] [flight id] cancel a booking
    editbooking [customer id] [flight id] update a booking
    loadgui              loads the GUI version of the app
    help                 prints this help message
    exit                exits the program
```

2. listflights

```
> listflights
Flight #1 - ig56 - nepal to india on 10/02/2022 price 1500 Number of seats 40
Flight #2 - yu89 - nepal to usa on 10/03/2022 price 2000 Number of seats 45
Flight #3 - de23 - nepal to canada on 10/04/2022 price 3000 Number of seats 50
3 flight(s)
```

3. listbookings


```
> listbookings
Customer #1 Flight #2 on 2022-10-10
1 booking(s)
```

4. listcustomers

```
Customer #1 - Shyam - 98124362631 - shyam@gmail.com
Customer #2 - Ram - 98000000000 - ram@gmail.com
2 customer(s)
```

5. addflight

```
> addflight
Flight Number: sn24
Origin: nepal
Destination: uk
Departure: Date ("YYYY-MM-DD" format): 2022-05-29
Number of Seats: 60
Price: 5000
Flight #4 added.
```

6. showflight [flight id]

```
> showflight 1
Flight #1 - ig56 - nepal to india on 10/02/2022
```

7. showcustomer [customer id]

```
> showcustomer 2
Customer #2 - Ram - 98000000000 - ram@gmail.com
```

8. customerbookinglist [customer id]

```
> customerbookinglist 1
Customer #1 Flight #2 on 2022-10-10
```

9. deleteflight [flight id]

```
> deleteflight 3
Flight details Deleted successfully
```

10. remainingseats [flight id]

```
> remainingseats 2  
Number of seats remaining: 44
```

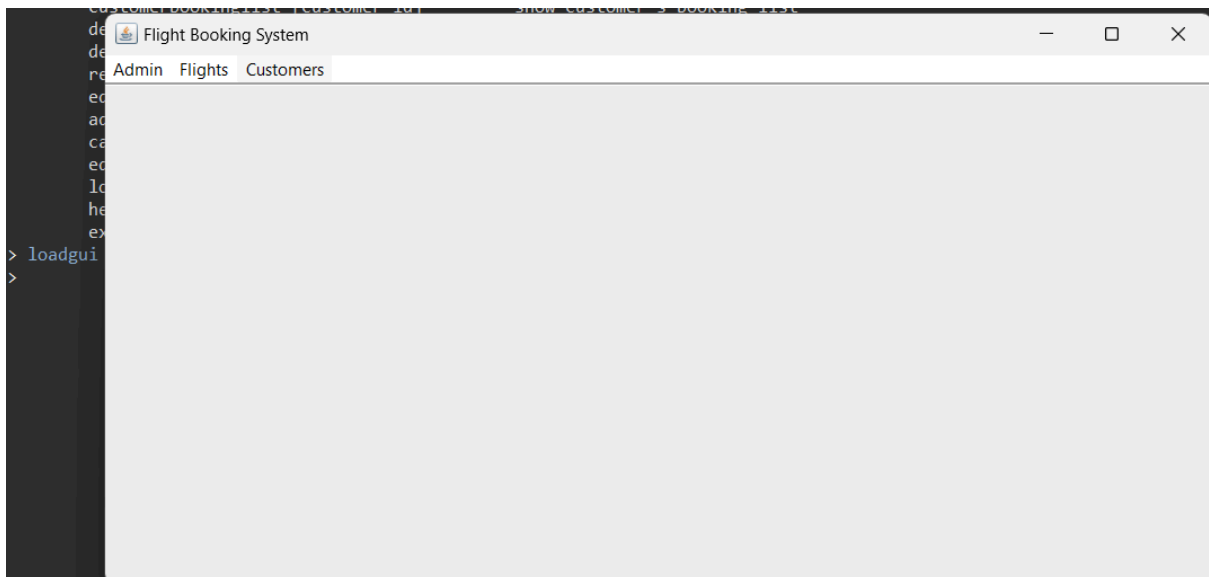
11. editcustomer [customer id]\=

```
> editcustomer 2  
Name: Sita  
Phone Number: 98212121212  
Email: sita@gmail.com  
Customer info updated successfully..
```

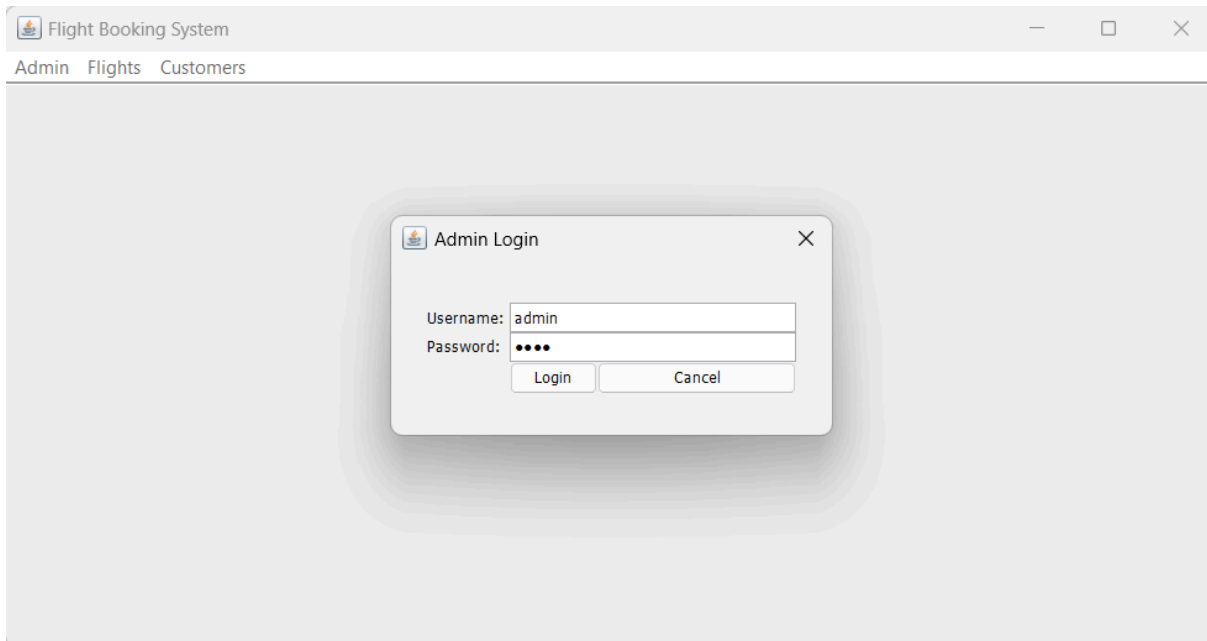
12. cancelbooking [customer id] [flight id]

```
> cancelbooking 1 2  
Booking cancelled successfully !!!
```

13. loadgui



14. adminlogin




15. Adding customer

The screenshot shows a dialog box titled "Add a New Customer". It has three input fields on the right side, each corresponding to a label on the left: "Customer Name :" with the value "Ramesh", "Phone Number :" with the value "987653424", and "Email :" with the value "ramesh@gmail.com". At the bottom of the dialog box are two buttons: "Add" and "Cancel".

Name	Phone Number	Email	No of Bookings
Shyam	98124362631	shyam@gmail.com	1
Ram	98000000000	ram@gmail.com	0
Ramesh	987653424	ramesh@gmail.com	0

16. Deleting customer

 Delete Customer with Id

Customer Id :

3

Confirm

Table			
Name	Phone Number	Email	No of Bookings
Shyam	98124362631	shyam@gmail.com	1
Ram	98000000000	ram@gmail.com	0