Software Testing Project Report

Airline Management System (Team 9)

Done by:
Mohamed Ahmed Esmat 21P0144

Abdallah Ahmed Hamdy 21P0333

Ahmed Hossam Eldin Ibrahim 21P0271

Saleh Ahmed Saleh 21P0324

Youssef Mohammed Alsayyd 21P0094

Presented to:

Dr. Yasmine Afify

Eng. Mahmoud Wagih

Eng. Amr Hassan

Date: April 26, 2024

Contents

List of Figures

List of Tables

1 Introduction

Our Airline Management System simplifies the process for passengers to sign up, search for flights, select seats, and cancel bookings, while administrators can manage flight details through their panel. We've ensured the system's reliability and smooth performance through comprehensive testing with JUnit, FSM for GUI, white box methods, and pairwise integration testing.

2 Requirement 1: Unit Testing

2.1 Introduction to Unit Testing

The unit testing is developed using JUnit 5 and it applied many concepts like the Nested testing and the Parametarized testing. Applied the Test Suites as well.

2.2 Airline Test

To begin with, we are going to define create objects from various classes to use in our test under the @BeforeEach such as creating flight objects and airline objects. While in the @AfterEach we teared down these objects. Below is the description for the test cases developed to test the functionalities of Airline:

1. **testBookSeat**:
    - **Description**: This test verifies the functionality of the `bookSeat` method with different seat types (Economy and First Class). It ensures that the method successfully books a seat for a given flight and passenger when there are available seats of the specified type.
    - **Expected Outcome**: The `bookSeat` method should return `true` for all provided seat types, indicating successful booking.
2. **testBookSeatEconomyClass**:
    - **Description**: This test focuses specifically on booking Economy class seats. It checks whether the number of available Economy class seats decreases by 1 after each booking.
    - **Expected Outcome**: The number of available Economy class seats should decrease by 1 after each booking.
3. **testBookSeatFirstClass**:
    - **Description**: Similar to the previous test, this one focuses on booking First Class seats. It verifies whether the number of available First Class seats decreases by 1 after each booking.
    - **Expected Outcome**: The number of available First Class seats should decrease by 1 after each booking.

4. **testBookSeatNoAvailableSeats**:
    - **Description**: This test checks the behavior when there are no available seats left for booking. It simulates booking all seats for both Economy and First Class and then attempts to book additional seats.
    - **Expected Outcome**: The `bookSeat` method should return `false` for both Economy and First Class seats, indicating that no seats are available for booking.
5. **testNotAvailableFlight**:
    - **Description**: This test verifies the behavior of the `bookSeat` method when the flight object is null.
    - **Expected Outcome**: The `bookSeat` method should return `false` when attempting to book a seat for a null flight, indicating that the operation is not possible.
6. **testCancelBooking**:
    - **Description**: This test case validates the functionality of the cancelBooking method in the Airline class. It first books a seat for a passenger on a specific flight in the economy class and verifies that the available economy seats are updated accordingly. Then, it cancels the booking by passing the passenger's ticket, flight, and passenger objects to the cancelBooking method. After cancellation, it checks that the available economy seats are restored to their original count, and the passenger's ticket is set to null, indicating the successful cancellation of the booking.
    - **Expected Outcome**: The test should pass if the booking is successfully canceled, and the available economy seats are correctly updated, restoring the previous count. Additionally, the passenger's ticket should be set to null after cancellation.
7. **testCancelBookingNoBooking**:
    - **Description**: This test case validates the behavior of the cancelBooking method when attempting to cancel a booking without providing a valid ticket object. It calls the cancelBooking method with null parameters for the ticket, flight, and passenger objects. This scenario represents an attempt to cancel a booking when no booking has been made, as indicated by the absence of a valid ticket object.
    - **Expected Outcome**: The test should pass if the cancelBooking method returns false, indicating that no booking was canceled due to the absence of a valid ticket object. This ensures that the cancelBooking method handles invalid input gracefully and does not affect the state of the system when no booking exists.

From the above data we can see the relation the Airline and Flight classes share.

2.3 Flight Test

1. **testCreateFlight**:
   - **Description**: The method tests the creation of flights in the setup function `createFlights`. It uses `assertNotNull` to verify that the flights are created successfully.
   - **Expected Outcome**: The test should pass, indicating that the flights are successfully created.
2. **testGenerateFlightId**:
   - **Description**: The method checks the integrity of the flight ID generation and assignment process. It asserts that the flight IDs are correctly assigned and remain consistent even after modifying flight IDs.
   - **Expected Outcome**: The test should pass, ensuring that flight IDs are generated and assigned correctly.
3. **testAddTicket**:
   - **Description**: This test method evaluates the addTicket function's ability to correctly add a Ticket object to multiple Flight objects. It uses a range of ticket indices to add tickets to two different flights and then verifies that the tickets have been added correctly.
   - **Expected Outcome**: The test should confirm that the same Ticket object is present in the ticket lists of both flights, ensuring that the addTicket method functions properly across different instances of Flight. The test should pass, indicating that tickets are being accurately tracked and recorded for each flight.
4. **testSearchTicket**:
   - **Description**: This method is a test case that validates the functionality of the searchTicket method within the Flight class. The first assertion (assertSame) checks whether the Ticket object retrieved by searching for the ticket ID "fc123" is the same as the first ticket in the flights.get(0)'s list of tickets. The second assertion (assertNotSame) verifies that the Ticket object retrieved for the same ticket ID is not the same as the second ticket in the same Flight's list of tickets.
   - **Expected Outcome**: By passing this test, we confirm that the searchTicket method correctly associates ticket IDs with their corresponding Ticket objects in the context of a specific flight.
5. **testRemoveTicket**:
   - **Description**: The testRemoveTicket method is a parameterized test that checks the removeTicket function of the Flight class. It uses a

CSV source to provide pairs of ticket IDs and expected sizes of the ticket list after removal. The test removes a ticket from the first Flight object in the flights list using the provided ticket ID. It then asserts that the size of the Flight object's ticket list matches the expected number.

- **Expected Outcome**: The test should pass, ensuring that flight IDs are generated and assigned correctly.

6. **testTickets**:
   - **Description**: The method ensures that the `setTickets` method of the Flight class correctly associates a list of Ticket objects with a flight. It then verifies that the `getTickets` method returns the same list of tickets.
   - **Expected Outcome**: The test should pass, confirming that tickets are correctly associated with the flight and retrieved accurately.
7. **testEconomySeatsAvailable**:
   - **Description**: The method checks that the `setEconomySeatsAvailable` and `getEconomySeatsAvailable` methods in the Flight class accurately update and retrieve the number of available economy seats for a flight.
   - **Expected Outcome**: The test should pass, ensuring that the economy seat count is accurately tracked and updated.
8. **testFirstClassSeatsAvailable**:
   - **Description**: The method verifies the functionality of updating and retrieving the number of available first-class seats in a Flight object.
   - **Expected Outcome**: The test should pass, confirming that the first-class seat count is accurately tracked and updated.
9. **testFrom**:
   - **Description**: The method ensures that the `setFrom` and `getFrom` methods of the Flight class correctly set and retrieve the departure location of a flight.
   - **Expected Outcome**: The test should pass, confirming that the departure location is accurately set and retrieved.
10. **testTo**:
    - **Description**: The method validates the functionality of the `setTo` and `getTo` methods in the Flight class. It checks that the destination of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, ensuring that the destination is correctly updated and retrieved.

11. **testEconomyPrice**:
    - **Description**: The method checks that the `setEconomyPrice` and `getEconomyPrice` methods in the Flight class correctly set and retrieve the price of economy seats for a flight.
    - **Expected Outcome**: The test should pass, confirming that the economy pricing is accurately set and retrieved.
12. **testFirstClassPrice**:
    - **Description**: The method validates the functionality of the `setFirstClassPrice` and `getFirstClassPrice` methods within the Flight class. It checks that the price of first-class seats is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, ensuring that the first-class pricing is correctly updated and retrieved.
13. **testDate**:
    - **Description**: The method verifies the functionality of the `setDate` and `getDate` methods within the Flight class. It ensures that the departure date of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, confirming that the departure date is correctly updated and retrieved.
14. **testTime**:
    - **Description**: The method validates the functionality of the `setTime` and `getTime` methods within the Flight class. It checks that the departure time of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, ensuring that the departure time is correctly updated and retrieved.
15. **testFlightId**:
    - **Description**: The method ensures the `setFlightId` and `getFlightId` methods of the Flight class are functioning correctly. It verifies that the flight ID of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, confirming that the flight ID is correctly set and retrieved.
16. **testHighDemand**:
    - **Description**: The method validates the functionality of the `setHighDemand` and `getHighDemand` methods within the Flight class. It checks that the high demand status of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, ensuring that the high demand status is correctly updated and retrieved.

17. **testCompetitors**:

    - **Description**: The method checks the `setCompetitors` and `getCompetitors` methods of the Flight class. It verifies that the competitors' status of a flight is accurately set and retrieved.
    - **Expected Outcome**: The test should pass, confirming that the competitors' status is correctly updated and retrieved.

2.4 Admin Test

1. createFlightByAdmin Test Case

    - Description: Verifies the functionality of the createFlightByAdmin method in the Airline class by creating a new flight.
    - Inputs: Details of the new flight (e.g., economy seats available, first class seats available, origin, destination, distance, time, date, demand, competitors).
    - Expected Output: The method should create a new flight with the provided details and ensure that the flight object is not null.
    - Output: Confirmation that the flight has been created successfully and that the flight object is not null.

2. economyFareCalculation Test Case

    - Description: Verifies the correctness of the economyFare method in the Airline class for various input scenarios.
    - Inputs and Expected Outputs:
        - Input: Distance = 800, High Demand = true, Competitors = true, Expected Output: 153.6
        - Input: Distance = 1506, High Demand = false, Competitors = true, Expected Output: 240.96
        - Input: Distance = 919, High Demand = false, Competitors = false, Expected Output: 183.8
        - Input: Distance = 1506, High Demand = true, Competitors = false, Expected Output: 361.44

3. firstClassFareCalculation Test Case

    - Description: Verifies the correctness of the firstclassFare method in the Airline class for various input scenarios.
    - Inputs and Expected Outputs:
        - Input: Distance = 800, High Demand = true, Competitors = true, Expected Output: 312

- Input: Distance = 1506, High Demand = false, Competitors = true, Expected Output: 451.8
- Input: Distance = 919, High Demand = false, Competitors = false, Expected Output: 367.6
- Input: Distance = 1506, High Demand = true, Competitors = false, Expected Output: 783.12

4. delayFlightTest

   - Description: Verifies the functionality of the delayFlight method in the Airline class.
   - Inputs: Flight ID, new date, and new time for delaying the flight.
   - Expected Output: The method should update the date and time of the specified flight and ensure that the changes are reflected in the flight object.
   - Output: The date and time of the specified flight are checked against the expected values after the delay operation.

5. cancelFlightTest

   - Description: Verifies the functionality of the cancelFlight method in the Airline class.
   - Inputs: Flight ID for cancelling the flight.
   - Expected Output: The method should cancel the specified flight and ensure that it is removed from the list of flights.
   - Output: The cancellation status of the flight is verified, and the list of flights is checked to ensure that the specified flight is no longer present. If the flight is cancelled, a confirmation message is printed.

2.5 LoginSignup Test

1. **Signup Failure Test Description:** This test checks if the signup process fails when providing existing user credentials.
   Input:

   - Username: "Ahmed24"
   - Password: "AO@76"
   - Name: "Ahmed Hatem Amr"
   - Passport: "P123456"
   - Email: "ahmed24@example.com"
   - Phone: "1234567890"
   - Address: "123 Street, City, Country"

- Date of Birth: January 1, 1990
- Nationality: "Egyptian"
- Terms: true

Expected Output: Signup fails (`assertFalse`)

2. **Signup Success Test Description:** This test checks if the signup process succeeds with unique user credentials.
   Input:
   - Username: "NewUser"
   - Password: "NU@123"
   - Name: "New User"
   - Passport: "P654321"
   - Email: "newuser@example.com"
   - Phone: "9876543210"
   - Address: "321 Lane, City, Country"
   - Date of Birth: April 4, 2000
   - Nationality: "British"
   - Terms: false

   Expected Output: Signup succeeds (`assertTrue`)

3. **Invalid Login Test**
   - **Description:** Tests if the login fails with incorrect credentials.
   - **Input:** Username: "Ahmed24", Password: "xx@xx"
   - **Expected Output:** Login returns -1 (assertEquals)

4. **Regular User Login Test**
   - **Description:** Tests if a regular user can log in successfully.
   - **Input:** Username: "JohnDoe", Password: "JD@123"
   - **Expected Output:** Login returns 1 (assertEquals)

5. **Admin Login Test**
   - **Description:** Tests if an admin user can log in successfully.
   - **Input:** Username: "admin1", Password: "admin1"
   - **Expected Output:** Login returns 0 (assertEquals)

6. **New User Login Test**
   - **Description:** Tests if a newly signed up user can log in successfully.
   - **Input:** Username: "NewUser", Password: "NU@123"
   - **Expected Output:** Login returns 1 (assertEquals)

2.6 Search Test

1. **Search Index Flight By Id Test Method:**

   - **Description:** this method tests the functionality of searching for a flight index by its ID.
   - **Test Cases:**
   - Searches for a non-existent flight ID, expecting -1.
     - **Input:** The method tests the searchIndexFlightById functionality of the Airline class.It calls the searchIndexFlightById method with none existing flight ID (9999).
     - **Expected Output:** assertEquals(-1, index) The assert makes sure that the non existing flight id returns -1 that means False Id .(assertTrue)
   - Searches for existing flight and asserts their corresponding indexes.
     - **Input:** The method tests the searchIndexFlightById functionality of the Airline class.It calls the searchIndexFlightById method with existing flight IDs(10001,10002,10003).
     - **Expected Output:** The assert makes sure that the given Index of the flight id (10001 and 10002) is matching to the index of flights assertEquals(0, airline.searchIndexFlightById("10001")) and assertEquals(1, airline.searchIndexFlightById("10002")) and return (assertTrue)

       the assert checks that if the flight id do exist but not to its corresponding index assertFalse(airline.searchIndexFlightById("10003") == 3) and return (assertTrue) if they aren't matching.

2. **Search Flight By Id Test Method:**

   - **Description:** this method tests the functionality of searching for a flight by its ID.
   - **Test Cases:**
   - Searches for a non-existent flight ID , expecting null.
     - **Input:** It calls the searchFlightById method from class Airline and sets The id to be searched to be a wrong id number ("9999"). ".
     - **Expected Output:** For the flight id ("9999") that is wrong The expected output is NULL.
   - Searches for existing flight IDs and asserts their corresponding flight objects.
     - **Input:** It calls the searchFlightById method from class Airline and sets The id to be searched with existing IDs (10001)(10002)(10003).

- **Expected Output:** If the id matches the index of flight the assert returns True assertEquals(airline.flights.get(0), airline.searchFlightById("10001")) and assertEquals(airline.flights.get(0), airline.searchFlightById("10002")) (assertTrue).

  and if the id doesn't match the index assertNotEquals(airline.flights.get(3), airline.searchFlightById("10003")) (assertTrue) .

3. **Test Seacrh Flights Method:**

- **Description:** This method tests the functionality of searching for flights based on departure and destination cities and a specific date.

  Test Cases:

- It specifies the departure city (from), destination city (to), and date. Then searches for flight in the object of Airline.Finally, validates that the search results match the expected flights from the setup.
  - **Input:** Set 'from' to Cairo , Set 'to' to Aswan , Set Date to 20/4/2024.
  - **Expected Output:** If the search operation successfully finds a flight that matches the specified criteria (departure city, destination city, and date), and if that flight is the first flight in the list of flights stored in the airline object (airline.flights.get(0)), then the assertion will pass (assertTrue),If one of the criteria is not matching (assertFalse).

2.7 PassengerTicket Test

1. **Ticket Setter and Getter Test:**

- **Description:** Tests the setter and getter methods for various attributes of the Ticket class.
- **Test Cases:**
- Set and Get Ticket ID Test
  - **Input:** Set Ticket ID to "t123".
  - **Expected Output:** Ticket ID should be "t123".
- Set and Get Seat Type Test
  - **Input:** Set Seat Type to "First Class".
  - **Expected Output:** Seat Type should be "First Class".
- Set and Get Price Test
  - **Input:** Set Price to 1000.
  - **Expected Output:** Price should be 1000.
- Set and Get Flight Test
  - **Input:** Set Flight object.

- **Expected Output:** Flight object should be returned.

- Set and Get Passenger Test
  - **Input:** Set Passenger object.
  - **Expected Output:** Passenger object should be returned.

2. **Passenger Setter and Getter Test:**

- **Description:** Tests the setter and getter methods for various attributes of the Passenger class.
- **Test Cases:**
- Set and Get Username Test
  - **Input:** Set Username to "Ahmed123".
  - **Expected Output:** Username should be "Ahmed123".
- Set and Get Password Test
  - **Input:** Set Password to "ahmed123".
  - **Expected Output:** Password should be "ahmed123".
- Set and Get Name Test
  - **Input:** Set Name to "Ahmed Hossam ElDin".
  - **Expected Output:** Name should be "Ahmed Hossam ElDin".
- Set and Get Passport Number Test
  - **Input:** Set Passport Number to "P123456".
  - **Expected Output:** Passport Number should be "P123456".
- Set and Get Email Test
  - **Input:** Set Email to "ahmed123@example.com".
  - **Expected Output:** Email should be "ahmed123@example.com".
- Set and Get Phone Number Test
  - **Input:** Set Phone Number to "1234567890".
  - **Expected Output:** Phone Number should be "1234567890".
- Set and Get Address Test
  - **Input:** Set Address to "123 Street, Cairo, Egypt".
  - **Expected Output:** Address should be "123 Street, Cairo, Egypt".
- Set and Get Date of Birth Test
  - **Input:** Set Date of Birth to January 1, 1990.
  - **Expected Output:** Date of Birth should be January 1, 1990.
- Set and Get Nationality Test
  - **Input:** Set Nationality to "Egyptian".
  - **Expected Output:** Nationality should be "Egyptian".
- Set and Get Adult Test
  - **Input:** Set Adult to true.
  - **Expected Output:** Adult status should be true.

3 Requirement 2: GUI Testing

3.1 Introduction

Welcome to the documentation for the Airline Management System's Graphical User Interface (GUI) Testing. This system serves as a comprehensive tool for managing various aspects of airline operations, ranging from passenger bookings to administrative tasks. In this document, We explore the nuances of GUI testing, exploring the states, events, and transitions within both the Passenger Pages and Admin Panel interfaces. By analyzing the system's behavior and interactions, we aim to ensure robust functionality and an excellent user experience for both passengers and administrators.

3.2 Login, Signup, and Admin Panel

3.2.1 States and Events (Part 1)

State	**Events**
Login	1. Enter data 2. Press Signup 3. Press Login
Signup	1. Enter data 2. Press Signup
Validation	1. Invalid Data or Already Existing Username 2. Valid Data
Alert	1. Press OK
Validation and Auth	1. Correct Admin Credentials 2. Correct Passengers credentials 3. Wrong Credentials
Admin Page	1. Double Click on a Flight 2. Press Back 3. Enter data for Adding
Adding Flight	1. Enter data 2. Press Add
Editing Flight	1. Enter data 2. Press Edit
Flight Selected	1. Press delete 2. Enter data for editing

Table 1: Event States and Associated Events

3.2.2 Description

The following is a detailed description of the states and transitions within the Passenger Pages section of the Airline Management System:

1. **Login State**: users enter the system's initial state, where they can press sign up and move to the state of sign up or enter data and move to Validate and Authenticate the data entered is matched to any user .

2. **Signup State**: This state allows users to create a new account by providing necessary information and submitting their signup data to be validated by Validation State.

3. **Validation State**: Here, the system validates the entered data, ensuring it meets required criteria and ensure that all needed information about the user is filled . If the data is invalid or the username already exists it moves to Alert State that Shows the user the kind of errors that are triggered, guiding users to rectify their inputs or move to the Login page if the user already exists.

4. **Alert State**: An alert message is displayed to the user describes the problem and what is wrong, Alert messages are always connected to scenarios where data validation or Authentication fails . Users must acknowledge the alert by pressing "OK" to proceed .

5. **Validation and Authentication State**: This state involves verifying user credentials for authentication. Depending on whether the provided credentials are correct and belong to an admin so it shall continue to admin Page State or belong To a passenger and so move to Passenger Page State .If the credentials are wrong move to Alert State to inform the user on what is wrong .

6. **Admin page State**: upon successful authentication as an admin, admin is directed to the admin panel. This state provides access to administrative functionalities such as managing flights, adding new flights, editing existing flights, deleting flights .If admin double-clicked on a flight that means that the system should transition to Flight selected State transition and if chose to enter data of a flight the system transition to Adding flight State.

7. **Flight Selected State**: When an admin selects a flight from the admin panel, this state is entered. Here, admin can perform actions such as deleting the selected flight by pressing delete or editing its details by Entering Data for editing .

8. **Editing Flight State**: Admins can modify the details of a selected flight in this state. They input updated data for the flight.By pressing Edit the system transitions to Validate State to validate entered data is correct or not and move to other states according to the output of the validation process.

9. **Adding Flight State**: In this state, admins can input data for adding a new flight to the system. They provide necessary flight details and confirm the addition,By pressing add the system transitions to Validate State to

validate entered data is correct or not and move to other states according to the output of the validation process.

3.3 Passenger Pages

3.3.1 States and Events

State	Events
Search Flights	1. Enter Data 2. Press Back 3. Press Confirm
Search	1. Available Flights 2. No Available Flights
Alert	1. Press OK
Choosing Flight	1. Double Click on a Flight 2. Press Back
Flight Selected	1. Press the Seat Type
Seat Type Selected	1. Press Confirm Ticket
Checking Availability	1. Seats Available 2. No Available Seats for this class
Fail Alert	1. Press OK
Success Alert	1. Press OK
Ticket Status	1. Press Cancel

Table 2: Event States and Associated Events

3.3.2 Description

The following is a detailed description of the states and transitions within the Passenger Pages section of the Airline Management System:

1. **Search Flights State (Post-Login Process Initiation)**: Upon successful login, the user enters the 'Search Flights' state. This state provides input fields for Departure, Arrival, and Date, along with 'Back' button to return to the Login and 'Confirm' button to proceed to search process.

2. **Search State (Flight Search Outcomes)**: If the search yields no matching flights, the system transitions to the 'Alert' state (State 3). The user must press "OK" in the alert to return to the 'Search Flights' state. If one or more matching flights are found, the system transitions to the 'Choosing Flights' state (State 4).

3. **Alert State (No Match Scenario)**: An alert is displayed, prompting the user to modify their search criteria. The user must press "OK" to return to the 'Search Flights' state.

4. **Choosing Flights State (Flight Selection Interface)**: In this state, a table lists all matched flights. Users double-click on their preferred flight, leading them to the 'Flight Selected' state (State 5).

5. **Flight Selected State (Seat Class Decision)**: Users select their preferred seat class. This decision triggers a transition to the 'Seat Type Selected' state (State 6).

6. **Seat Type Selected State**: The 'Seat Type Selected' state will allow the user to press 'Confirm Ticket' button to continue to 'Checking Availability State' (state 7)

7. **Checking Availability State (Availability Verification)**: If no seats are available in the chosen class, the system transitions to the 'Fail Alert' state (State 8). The user must press "OK" in the alert to return to the 'Flight Selected' state. If seats are available, the system transitions to the 'Success Alert' state (State 9).

8. **Fail Alert State (Unavailable Seats)**: An alert informs the user that no seats are available in the chosen class. The user must press "OK" to return to the 'Flight Selected' state.

9. **Success Alert State (Seat Booked)**: An alert confirms seat availability. The user must press "OK" to proceed to the 'Ticket Status State' state (State 10).

10. **Ticket Status State (Ticket Details)**: The 'Ticket Status' state displays details of the selected flight ticket. A 'Cancel' button allows users to return to the 'Choosing Flights' state (state 4).
    The FSM could be checked here: 1

4 Requirement 3: White Box Testing

4.1 Introduction

The following sections of the white box testing will provide test cases that ensure 100% coverage for statement, decision, and condition testing.

4.2 Airline Class

4.2.1 bookSeat Method

```
public boolean bookSeat(Flight flight, String seatType, Passenger passenger) {
// Instruction 1
double final_price;

// Instruction 2
String ticketid;

// Instruction 3
for (Flight flighttocheck : flights) {

    // Instruction 4
    if (flighttocheck == flight) {

        // Instruction 5
        if ((seatType.equalsIgnoreCase("Economy") && flight.getEconomySeatsAvailable() > 0)
            (seatType.equalsIgnoreCase("First Class") && flight.getFirstClassSeatsAvailable(

            // Instruction 6
            if (seatType.equalsIgnoreCase("Economy")) {
                ticketid = "e " + flight.getEconomySeatsAvailable();
                // Instruction 7
                flight.setEconomySeatsAvailable(flight.getEconomySeatsAvailable() - 1);
                // Instruction 8
                final_price = flight.getEconomyPrice();
            } else {
                // Instruction 9
                ticketid = "fc " + flight.getFirstClassSeatsAvailable();
                // Instruction 10
                flight.setFirstClassSeatsAvailable(flight.getFirstClassSeatsAvailable() - 1)
                // Instruction 11
                final_price = flight.getFirstClassPrice();
            }

            // Instruction 12
            Ticket ticket = new Ticket(ticketid, seatType, final_price);
```

```
            // Instruction 13
            ticket.setFlight(flight);
            // Instruction 14
            ticket.setPassenger(passenger);
            // Instruction 15
            passenger.setTicket(ticket);
            // Instruction 16
            flight.addTicket(ticket);
            // Instruction 17
            return true;
        } else {
            // Instruction 18
            System.out.println("Sorry, no seats available for this flight.");
            // Instruction 19
            return false;
        }
    }
}
// Instruction 20
return false;
}
```

Statement Coverage Test

1. Test Case 1: Book a seat in Economy class when seats are available.
    - Input:
        - `Flight: flight1`
        - `Seat type: "Economy"`
        - `Passenger: passenger1`
    - Instructions executed: 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 17
2. Test Case 2: Book a seat in First Class when seats are available.
    - Input:
        - `Flight: flight1`
        - `Seat type: "First Class"`
        - `Passenger: passenger1`
    - Instructions executed: 1, 2, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15, 16, 17
3. Test Case 3: Attempt to book a seat when no seats are available.
    - Input:
        - `Flight: full_flight1`
        - `Seat type: "First Class"`

- Passenger: passenger1
- Instructions executed: 1, 2, 3, 4, 5, 18, 19

4. Test Case 4: Attempt to book a seat on a non-existing flight.
   - Input:
     - Flight: non_existing_flight
     - Seat type: "Economy"
     - Passenger: passenger1
   - Instructions executed: 1, 2, 3, 20

Condition Coverage Test

1. Test Case 1: Condition for booking a seat in Economy class is true.
   - Input:
     - Flight: flight1
     - Seat type: "Economy"
   - Conditions checked: - flighttocheck == flight, (seatType.equalsIgnoreCase("Economy") && flight.getEconomySeatsAvailable() > 0)
2. Test Case 2: Condition for booking a seat in First Class is true.
   - Input:
     - Flight: flight1
     - Seat type: "First Class"
   - Conditions checked: - flighttocheck == flight, (seatType.equalsIgnoreCase("First Class") && flight.getFirstClassSeatsAvailable() >
3. Test Case 3: Condition for no seats available is true.
   - Input:
     - Flight: full_flight1
     - Seat type: "First Class"
   - Conditions checked: - flighttocheck == flight, (seatType.equalsIgnoreCase("Economy") && flight.getEconomySeatsAvailable() > 0), (seatType.equalsIgnoreCase("First Class") && flight.getFirstClassSeatsAvailable() >
4. Test Case 4: Condition for flight not found is true.
   - Input:
     - Flight: non_existing_flight
     - Seat type: "Economy"
   - Conditions checked: - flighttocheck == flight

Decision Coverage Test

1. Test Case 1: Decision for booking a seat in Economy class is true.
   - Input:
     - `Flight: flight1`
     - `Seat type: "Economy"`
   - Branches checked: - Seat booking branch (Instruction 17)
2. Test Case 2: Decision for booking a seat in First Class is true.
   - Input:
     - `Flight: flight1`
     - `Seat type: "First Class"`
   - Branches checked: - Seat booking branch (Instruction 17)
3. Test Case 3: Decision for no seats available is true.
   - Input:
     - `Flight: full_flight1`
     - `Seat type: "First Class"`
   - Branches checked: - No seats available branch (Instruction 19)
4. Test Case 4: Decision for flight not found is true.
   - Input:
     - `Flight: non_existing_flight`
     - `Seat type: "Economy"`
   - Branches checked: - Flight not found branch (Instruction 20)

4.2.2 cancelBooking Method

```
public boolean cancelBooking(Ticket ticket, Flight flight, Passenger passenger) {
    // Instruction 1
    if (ticket != null) {
        // Instruction 2
        if (ticket.getSeatType().equalsIgnoreCase("Economy")) {
            // Instruction 3
            flight.setEconomySeatsAvailable(flight.getEconomySeatsAvailable() + 1);
          //Instruction 4
        } else if (ticket.getSeatType().equalsIgnoreCase("First Class")) {
            // Instruction 5
            flight.setFirstClassSeatsAvailable(flight.getFirstClassSeatsAvailable() + 1);
        }
        // Instruction 6
        passenger.setTicket(null); // Remove ticket reference from passenger
```

```
        // Instruction 7
        flight.removeTicket(ticket.getTicketId());
        // Instruction 8
        ticket = null; // Delete the ticket object
        // Instruction 9
        return true;
    } else {
        // Instruction 10
        System.out.println("No booking found for this passenger.");
        // Instruction 11
        return false;
    }
}
```

Statement Coverage Test

1. Test Case 1: Canceling Economy booking successfully.
   - Input:
     - `Ticket: ticket1`
     - `Flight: flight1`
     - `Passenger: passenger1`
   - Instructions executed: 1, 2, 3, 5, 6, 7, 8
2. Test Case 2: Canceling First Class booking successfully.
   - Input:
     - `Ticket: ticket1`
     - `Flight: flight1`
     - `Passenger: passenger1`
   - Instructions executed: 1, 4, 5, 6, 7, 8, 9
3. Test Case 3: No booking found scenario.
   - Input:
     - `Ticket: null`
     - `Flight: flight1`
     - `Passenger: passenger1`
   - Instructions executed: 1, 10, 11

Condition Coverage Test

1. Test Case 1: Condition for canceling booking successfully.
   - Input:

- `Ticket: ticket1`
- Conditions checked: - `ticket != null`

2. Test Case 2: Condition for no booking found scenario.
   - Input:
     - `Ticket: null`
   - Conditions checked: - `ticket == null`

Decision Coverage Test

1. Test Case 1: Decision outcomes for canceling booking successfully.
   - Input:
     - `Ticket: ticket1`
   - Branches checked: - Canceling booking branch (Instructions 1-8)
2. Test Case 2: Decision outcomes for no booking found scenario.
   - Input:
     - `Ticket: null`
   - Branches checked: - No booking found branch (Instructions 9-10)

4.2.3 Signup Method

```
public boolean signup(String username, String password, String name,
                      String passportNumber, String email, String phoneNumber,
                      String address, LocalDate dateOfBirth, String nationality,
                      Boolean isAdult) {
    // Instruction 1
    for(Passenger passenger : passengers) {
        // Instruction 2
        if(passenger.getUsername().equals(username)) {
            // Instruction 3
            System.out.println("Username already exists. Please try again.");
            return false;
        }
    }
    // Instruction 4
    Passenger passenger = new Passenger(username, password, name, passportNumber,
                                        email, phoneNumber, address, dateOfBirth,
                                        nationality, isAdult);
    // Instruction 5
    passengers.add(passenger);
    // Instruction 6
    System.out.println("Signup successful.");
```

```
    // Instruction 7
    return true;
}
```

Statement Coverage Test

1. Test Case 1: Signing up with a new username.
    - Input:

```
signup("newuser", "password", "John Doe", "AB123456",
       "john@example.com", "1234567890", "123 Main St",
       LocalDate.of(1990, 1, 1), "USA", true)
```

    - Instructions executed: 1, 4, 5, 6, 7
2. Test Case 2: Attempting to signup with an existing username.
    - Input:

```
signup("existinguser", "password", "Jane Doe", "CD789012",
       "jane@example.com", "9876543210", "456 Elm St",
       LocalDate.of(1985, 5, 10), "UK", false)
```

    - Instructions executed: 1, 2, 3

Condition Coverage Test

1. Test Case 1: Condition for checking if the username already exists is false.
    - Input:

```
signup("newuser", "password", "John Doe", "AB123456",
       "john@example.com", "1234567890", "123 Main St",
       LocalDate.of(1990, 1, 1), "USA", true)
```

    - Conditions checked: - No repeated username
2. Test Case 2: Condition for checking if the username already exists is true.
    - Input:

```
signup("existinguser", "password", "Jane Doe", "CD789012",
       "jane@example.com", "9876543210", "456 Elm St",
       LocalDate.of(1985, 5, 10), "UK", false)
```

    - Conditions checked: - Repeated username

Decision Coverage Test

1. Test Case 1: Decision for signing up with a new username is true.
   - Input:

```
signup("newuser", "password", "John Doe", "AB123456",
       "john@example.com", "1234567890", "123 Main St",
       LocalDate.of(1990, 1, 1), "USA", true)
```

   - Branches checked: - Signup successful branch (Instructions 6, 7)
2. Test Case 2: Decision for attempting to signup with an existing username.
   - Input:

```
signup("existinguser", "password", "Jane Doe", "CD789012",
       "jane@example.com", "9876543210", "456 Elm St",
       LocalDate.of(1985, 5, 10), "UK", false)
```

   - Branches checked: - Username already exists branch (Instructions 2, 3)

4.2.4 Login Method

```
public int login(String username, String password) {
    // Instruction 1
    for (Passenger passenger : passengers) {
        // Instruction 2
        if (passenger.getUsername().equals(username) &&
            passenger.getPassword().equals(password)) {
            // Instruction 3
            System.out.println("Passenger Login successful.");
            this.p = passenger;
            // Instruction 4
            return 1;
        }
    }
    // Instruction 5
    for (Admin admin : admins) {
        // Instruction 6
        if (admin.getUsername().equals(username) &&
            admin.getPassword().equals(password)) {
            // Instruction 7
            System.out.println("Admin Login successful.");
            // Instruction 8
            return 0;
        }
```

```
    }
    // Instruction 9
    System.out.println("Invalid username or password. Please try again.");
    // Instruction 10
    return -1;
}
```

Statement Coverage Test

1. Test Case 1: Passenger login with correct credentials.
    - Input: `login("passenger", "password")`
    - Instructions executed: 1, 2, 3, 4
2. Test Case 2: Admin login with correct credentials.
    - Input: `login("admin", "password")`
    - Instructions executed: 1, 5, 6, 7, 8
3. Test Case 3: Login with incorrect credentials.
    - Input: `login("invalid", "password")`
    - Instructions executed: 1, 9, 10

Condition Coverage Test

1. Test Case 1: Condition for passenger login is true.
    - Input: `login("passenger", "password")`
    - Conditions checked: - passenger.getUsername().equals(username) && passenger.getPassword().equals(password)
2. Test Case 2: Condition for admin login is true.
    - Input: `login("admin", "password")`
    - Conditions checked: - admin.getUsername().equals(username) && admin.getPassword().equals(password)
3. Test Case 3: Condition for invalid login is true.
    - Input: `login("invalid", "password")`
    - Conditions checked: - No passenger or admin with matching credentials

Decision Coverage Test

1. Test Case 1: Decision for passenger login is true.
    - Input: `login("passenger", "password")`
    - Branches checked: - Passenger login successful branch (Instructions 3, 4)
2. Test Case 2: Decision for admin login is true.
    - Input: `login("admin", "password")`
    - Branches checked: - Admin login successful branch (Instructions 7, 8)
3. Test Case 3: Decision for invalid login is true.
    - Input: `login("invalid", "password")`
    - Branches checked: - Invalid username or password branch (Instructions 9, 10)

4.2.5 createFlight Method

```
public void createFlight(int economySeatsAvailable, int firstClassSeatsAvailable,
                         String from, String to, double distance, LocalTime time,
                         LocalDate date, boolean highDemand, boolean competitors) {
    // Instruction 1
    double economyprice = economyFare(distance, highDemand, competitors);
    // Instruction 2
    double firstclassprice = firstclassFare(distance, highDemand, competitors);
    // Instruction 3
    Flight flight = new Flight(flights, economySeatsAvailable, firstClassSeatsAvailable,
                               from, to, economyprice ,firstclassprice , time, date,
                               highDemand, competitors);
    // Instruction 4
}
```

Statement Coverage Test

1. Test Case 1: All instructions executed.
    - Input: createFlight(100, 50, "New York", "London", 500, LocalTime.of(12, 30), LocalDate.of(2024, 5, 1), true, false)
    - Instructions executed: 1, 2, 3, and 4

Condition Coverage Test

1. Test Case 1: Condition for economy fare calculation.
    - Input: highDemand = true, competitors = false
    - Conditions checked: Instructions 1, 2, 3, 4, 5, 6
2. Test Case 2: Condition for first class fare calculation.
    - Input: highDemand = false, competitors = true
    - Conditions checked: Instructions 1, 2, 3, 4, 5, 6

Decision Coverage Test

1. Test Case 1: Decision outcome for calling economyFare and firstclassFare.
    - Input: createFlight(100, 50, "New York", "London", 500, LocalTime.of(12, 30), LocalDate.of(2024, 5, 1), true, false)
    - Branches checked: Instructions 1, 2, 3, and 4 executed.
2. Test Case 2: Decision outcome for creating a new Flight object.
    - Input: createFlight(100, 50, "New York", "London", 500, LocalTime.of(12, 30), LocalDate.of(2024, 5, 1), true, false)
    - Branches checked: Instructions 1, 2, 3, and 4 executed.

4.2.6 economyFare Method

```
public double economyFare(double distance, boolean highDemand, boolean competitors) {
    // Instruction 1
    double economyprice = distance * 0.2;
    // Instruction 2
    if(highDemand)
        // Instruction 3
        economyprice = economyprice * 1.2;
        // Instruction 4
    if(competitors)
        // Instruction 5
        economyprice = economyprice * 0.8;
        // Instruction 6
    // Instruction 7
    return economyprice;
}
```

Statement Coverage Test

1. Test Case 1: All instructions executed.
    - Input: economyFare(500, true, false)
    - Instructions executed: 1, 2, 3, 4, 5, 6, and 7

Condition Coverage Test

1. Test Case 1: Condition for high demand is true and competitors is false.
   - Input: highDemand = true, competitors = false
   - Conditions checked: Instructions 1, 2, 3, 4, 5, 7
2. Test Case 2: Condition for high demand is false and competitors is true.
   - Input: highDemand = false, competitors = true
   - Conditions checked: Instructions 1, 2, 4, 5, 6, 7
3. Test Case 3: Condition for high demand and competitors are both true.
   - Input: highDemand = true, competitors = true
   - Conditions checked: Instructions 1, 2, 3, 4, 5, 6, 7
4. Test Case 4: Condition for high demand and competitors are both false.
   - Input: highDemand = false, competitors = false
   - Conditions checked: Instructions 1, 2, 7

Decision Coverage Test

1. Test Case 1: Decision outcome for high demand condition.
   - Input: economyFare(500, true, false)
   - Branches checked: Instructions 1, 2, 3, 4, 5, and 7 executed. Instruction 6 not executed.
2. Test Case 2: Decision outcome for competitors condition.
   - Input: economyFare(500, false, true)
   - Branches checked: Instructions 1, 2, 3, 5, 6, and 7 executed. Instruction 4 not executed.

4.2.7 firstClassFare Method

```
public double firstClassFare(double distance, boolean highDemand, boolean competitors) {
    // Instruction 1
    double firstclassprice = distance * 0.4;
    // Instruction 2
    if(highDemand)
        // Instruction 3
        firstclassprice = firstclassprice * 1.3;
        // Instruction 4
    if(competitors)
        // Instruction 5
```

```
        firstclassprice = firstclassprice * 0.75;
        // Instruction 6
    // Instruction 7
    return firstclassprice;
}
```

Statement Coverage Test

1. Test Case 1: All instructions executed.
    - Input: firstClassFare(500, true, false)
    - Instructions executed: 1, 2, 3, 4, 5, 6, and 7

Condition Coverage Test

1. Test Case 1: Condition for high demand is true and competitors is false.
    - Input: highDemand = true, competitors = false
    - Conditions checked: Instructions 1, 2, 3, 4, 5, 7
2. Test Case 2: Condition for high demand is false and competitors is true.
    - Input: highDemand = false, competitors = true
    - Conditions checked: Instructions 1, 2, 4, 5, 6, 7
3. Test Case 3: Condition for high demand and competitors are both true.
    - Input: highDemand = true, competitors = true
    - Conditions checked: Instructions 1, 2, 3, 4, 5, 6, 7
4. Test Case 4: Condition for high demand and competitors are both false.
    - Input: highDemand = false, competitors = false
    - Conditions checked: Instructions 1, 2, 7

Decision Coverage Test

1. Test Case 1: Decision outcome for high demand condition.
    - Input: firstClassFare(500, true, false)
    - Branches checked: Instructions 1, 2, 3, 4, 5, and 7 executed. Instruction 6 not executed.
2. Test Case 2: Decision outcome for competitors condition.
    - Input: firstClassFare(500, false, true)
    - Branches checked: Instructions 1, 2, 3, 5, 6, and 7 executed. Instruction 4 not executed.

4.2.8 cancelFlight Method

```
public boolean cancelFlight(String ID) {
    // Instruction 1
    int index = searchIndexFlightById(ID);
    // Instruction 2
    if (index == -1)
    // Instruction 3
        return false;
    // Instruction 4
    flights.remove(index);
    // Instruction 5
    return true;
   // Instruction 6
}
```

Statement Coverage Test

1. Test Case 1: Decision outcome for index is not -1.
    - Input: cancelFlight("ABC123")
    - Instructions executed: 1, 2, 3, 5, and 6
2. Test Case 2: Decision outcome for index is -1.
    - Input: cancelFlight("XYZ456")
    - Instructions executed: 1, 2, 3, and 4

Condition Coverage Test

1. Test Case 1: Index is -1.
    - Input: ID = "NonExistingID"
    - Conditions checked: Instructions 1, 2, 3
2. Test Case 2: Index is not -1.
    - Input: ID = "ExistingID"
    - Conditions checked: Instructions 1, 2, 4, 5

Decision Coverage Test

1. Test Case 1: Decision outcome for index is not -1.
    - Input: cancelFlight("ABC123")
    - Branches checked: Instructions 1, 2, 3, 5, and 6 executed. Instruction 4 not executed.

2. Test Case 2: Decision outcome for index is -1.

    - Input: cancelFlight("XYZ456")
    - Branches checked: Instructions 1, 2, 3, and 4 executed. Instructions 5 and 6 not executed.

4.2.9 delayFlight Method

```
public boolean delayFlight(String ID, LocalDate date, LocalTime time) {
    // Instruction 1
    int index = searchIndexFlightById(ID);
    // Instruction 2
    if (index == -1)
    // Instruction 3
    return false;
    // Instruction 4
    flights.get(index).setDate(date);
    // Instruction 5
    flights.get(index).setTime(time);
    // Instruction 6
    return true;
   //Instruction 7
}
```

Statement Coverage Test

1. Test Case 1: All instructions executed.

    - Input: delayFlight("ABC123", LocalDate.of(2024, 5, 1), LocalTime.of(14, 30))
    - Instructions executed: 1, 2, 3, 4, 5, 6, and 7

2. Test Case 2: Index is -1.

    - Input: delayFlight("XYZ456", LocalDate.of(2024, 5, 1), LocalTime.of(14, 30))
    - Instructions executed: 1, 2, and 3

Condition Coverage Test

1. Test Case 1: Index is -1.

    - Input: ID = "NonExistingID"
    - Conditions checked: Instructions 1, 2, 3

2. Test Case 2: Index is not -1.

- Input: ID = "ExistingID"
- Conditions checked: Instructions 1, 2, 4, 5

Decision Coverage Test

1. Test Case 1: Index is not -1.
   - Input: delayFlight("ABC123", LocalDate.of(2024, 5, 1), LocalTime.of(14, 30))
   - Branches checked: Instructions 1, 2, 3, 5, 6, and 7 executed. Instruction 4 not executed.
2. Test Case 2: Index is -1.
   - Input: delayFlight("XYZ456", LocalDate.of(2024, 5, 1), LocalTime.of(14, 30))
   - Branches checked: Instructions 1, 2, 3, and 4 are executed. Instructions 5, 6, and 7 not executed.

4.3 Flight Class

4.3.1 generateFlightNumber Method

```
private static String generateFlightNumber(List<Flight> flights, Flight flight) {
    // Instruction 1:
    int number = 10001;

    // Instruction 2:
    if (flights.isEmpty()) {
        // Instruction 3:
        flights.add(0, flight);
        return String.valueOf(number); // Instruction 4:
    } else {
        // Instruction 5:
        int lastNumber = Integer.parseInt(flights.get(flights.size() - 1).getFlightId());

        // Instruction 6:
        if (lastNumber == 99999) {
            // Instruction 7:
            for (int i = 1; i < flights.size(); ++i) {
                // Instruction 8:
                int currentNumber = Integer.parseInt(flights.get(i).getFlightId());
                int previousNumber = Integer.parseInt(flights.get(i - 1).getFlightId());

                // Instruction 9:
                if (currentNumber != previousNumber + 1) {
```

```
                    // Instruction 10:
                    flights.add(i, flight);
                    return String.valueOf(previousNumber + 1); // Instruction 11:
                }
            }
            // Instruction 12:
            flights.add(flight);
            return String.valueOf(100001); // Instruction 13:
        } else {
            // Instruction 14:
            flights.add(flight);
            return String.valueOf(lastNumber + 1); // Instruction 15:
        }
    }
}
```

Statement Coverage Test

1. **Test Case 1:** Testing when the list of flights is empty.
   - **Input:** $flights$ is empty.
   - **Instructions executed:** 1, 2, 3, 4
2. **Test Case 2:** Testing when the last flight number is 99999.
   - **Input:** Last flight number is 99999.
   - **Instructions executed:** 1, 5, 6, 7, 8, 9, 10, 13, 14
3. **Test Case 3:** Testing when there's a gap in flight numbers.
   - **Input:** There's a gap in flight numbers.
   - **Instructions executed:** 1, 5, 6, 7, 8, 9, 10, 13, 14
4. **Test Case 4:** Testing when the last flight number is within range.
   - **Input:** Last flight number is within range.
   - **Instructions executed:** 1, 5, 15, 16

Condition Coverage Test

1. **Test Case 1:** Testing when the list of flights is empty.
   - **Input:** $flights$ is empty.
   - **Conditions checked:** $flights.isEmpty()$
2. **Test Case 2:** Testing when the last flight number is 99999.

- **Input:** Last flight number is 99999.
- **Conditions checked:** $lastNumber == 99999$

3. **Test Case 3:** Testing when there's a gap in flight numbers.
   - **Input:** There's a gap in flight numbers.
   - **Conditions checked:** $currentNumber \neq previousNumber + 1$
4. **Test Case 4:** Testing when the last flight number is within range.
   - **Input:** Last flight number is within range.
   - **Conditions checked:** None

Decision Coverage Test

1. **Test Case 1:** Testing when the list of flights is empty.
   - **Input:** $flights$ is empty.
   - **Branches checked:** Flight list empty branch (Instructions 1, 2, 3, 4)
2. **Test Case 2:** Testing when the last flight number is 99999.
   - **Input:** Last flight number is 99999.
   - **Branches checked:** Last flight number equals 99999 branch (Instructions 1, 5, 6, 7, 8, 9, 10, 13, 14)
3. **Test Case 3:** Testing when there's a gap in flight numbers.
   - **Input:** There's a gap in flight numbers.
   - **Branches checked:** Gap in flight numbers branch (Instructions 1, 5, 6, 7, 8, 9, 10, 13, 14)
4. **Test Case 4:** Testing when the last flight number is within range.
   - **Input:** Last flight number is within range.
   - **Branches checked:** Last flight number within range branch (Instructions 1, 5, 15, 16)

5 Requirement 4: Integeration Testing

5.1 Introduction to Pairwise Testing

Pairwise testing, also known as all pairs testing, is a software testing technique that focuses on efficiently covering various combinations of input parameters. Instead of exhaustively testing all possible combinations, pairwise testing selects a subset of test cases that ensures each pair of input parameters is tested together at least once. It strikes a balance between comprehensive coverage and testing efficiency while optimizing testing effort and ensuring robust defect detection.

Advantages

- **Reduced Test Case Execution:** Pairwise testing significantly reduces the number of test cases compared to exhaustive testing. By covering all possible pairs of input values, it achieves thorough coverage without excessive test case proliferation.
- **Increased Test Coverage:** Pairwise testing aims to achieve maximum coverage by testing all pairs of input parameters. It ensures that interactions between different parameters are adequately explored, even when the number of parameters is large.
- **Enhanced Defect Detection:** By focusing on pairs of input values, pairwise testing increases the likelihood of detecting defects arising from interactions between parameters. It helps uncover issues that might not be apparent through individual testing.
- **Efficient Execution:** Pairwise testing takes less time to complete the execution of the test suite. It's especially beneficial when dealing with complex systems or large input spaces.
- **Cost-Effective Approach:** Pairwise testing reduces the overall testing budget for a project. It balances thoroughness with practical constraints, making it an economical choice.

5.2 Pairwise Relations

Flight - Airline

To begin with, we are going to show samples from the system's code to prove the relation between the Airline and the Flight class.

1. **Flights List**: In the Airline class, there's a list created mainly to store the available flights. The list stores flight as objects.
2. **bookSeat Method**: This method shows the strong connection between the Airline and Flight classes, since this method is based in the Airline class, and it takes the flight object as a parameter. The flight object used in this method to extract the flight details for the ticekt to be generated for this flight.
3. **cancelBooking Method**: This method is similar to the bookSeat method in terms of the relation between the Airline and the Flight, since it also takes the flight object as a parameter. It is used in the method to match the flight details with the ticket to cancel it.
4. **searchFlight Method**: We have this method that iterates through the Flight list to search for a specific flight given some of the data.

5. **createFlight Method**: This method shows the relation in which in the Airline mehtod, this method creates a new flight object according to the data given to the method.

From the above data we can see the relation the Airline and Flight classes share.

Airline-Passenger

1. **Function Name**: Why this function? What is the relation?

Flight-Ticket

1. **bookSeat Method**: In this function an object from Flight is passed as an argument to the setter of the Ticket (setTicket), and an object from Ticket is passed as an argument to the function (addTicket) to add the Ticket to the list of tickets in the Flight.

2. **cancelBooking Method**: In this function we used the tickedId to pass it as an argument to the funciton (removeTicket) to remove the ticket from the list of tickets in the Flight.

3. **tickets List**: This list is in the Flight class and each instance from the flight has its tickets list.

4. **addTicket method**: In this function an instance of Ticket is add to the list of tickets in the Flight

5. **searchTicket method**: In this function a ticketId is used to search for an object of Ticket and return it, this happen by iterating over the list of tickets and searching for the Ticket that has the same tickerId.

6. **removeTicket method**: In this function a ticketId is used to search for a Ticket that has the same ticketId to remove it from the tickets list.

Airline-Admin

1. **login method**: In this function we used two functions from the Admin class which are getUsername and this used to get the username of the admin and compare it to the username the user entered and the second function is getPassword which used to get the password of the admin to compare it to the password the user entered.

Ticket-Passenger

the relation between the Ticket and the Passenger class.

1. **bookSeat Method**:

   This function bookSeat,That is found in Airline Class ,Show us the connection made between two the two classes Ticket and the Passenger class

in the system .The function is designed to book a seat on a flight for a passenger. It takes three parameters: a Flight object, a seatType string, and a Passenger object.

In the booking process, a unique ticket ID is generated based on the seat type and the number of available seats of that type. The number of available seats of the requested type is then decreased by one.

Next, a new Ticket object is created with the generated ticket ID, the seat type, and the price corresponding to the seat type. This Ticket is then associated with the Flight and the Passenger.

The Ticket is added to the Flight's list of tickets, and the Passenger's ticket attribute is set to this Ticket. This establishes a two-way connection between the Passenger and the Ticket.

If the requested seat type is not available, the function prints a message indicating that there are no seats available and returns false.

2. **cancelBooking Method**: In the cancelBooking function, the connection between the Passenger and the Ticket is depicted as part of the cancellation process.

   If a Ticket object exists, the function first checks the seat type of the ticket. If it's an "Economy" or "first class" and increase accordingly the available seat numbers. This is done to reflect the newly available seat due to the cancellation.

   Next, the Ticket reference from the Passenger object is removed by setting it to null. This disconnects the Passenger from the Ticket.

   The Ticket is then removed from the Flight's list of tickets using the removeTicket method, which takes the ticket ID as a parameter. This disconnects the Flight from the Ticket.

   Finally, the Ticket object itself is deleted by setting it to null. This completes the cancellation process, and the function returns true. If no Ticket object was found for the Passenger, the function prints a message indicating this and returns false.

Airline-Passenger

1. bookSeat Method: Once again, this class establishes a connection between the "Airline" class and another class, in this case it's the "Passenger" class. If the flight the passenger wishes to go on is available, and there are available seats for this passenger, the function creates a new ticket with this seat type and assigns this ticket to the passenger using the "setTicket" method from the Passenger class.
The dependency between the two classes here is that airline manages the passenger data by assigning a ticket to them, and passenger holds the ticket information created by the bookSeat method in Airline class.

2. cancelBooking Method: Similar to the bookSeat method, there is a dependancy between the airline class and the passenger class, if the ticket the passenger wants to cancel is available, the cancelling is done by using the "setTicket" method from the passenger class as null.
The dependency between the two classes here is that airline also manages the passenger data by setting the ticket information in the passenger class as null. The airline classes needs the function "setTicket" from the passenger class in order for the "cancelBooking" function to run successfully.

3. Sign up Method: The sign up method is used purely by the passenger. It takes in the passenger information for creating a new account and traverses through the list of passengers that it has stored as an attribute in the Airline class. If the user does not exist, the function creates a new object from class Passenger using its constructor and appends this new passenger to the existing list of passengers in airline class.
This is a clear dependency from the Airline class to the Passenger class as the existence of Passenger class is crucial for the functionality of the "Sign Up" method.

4. Login Method: The dependency between the class "Airline" and the class "Passenger" is also shown in the function "login" as the passenger enters his login information and the function uses the existing passenger list in the Airline class to find out if the passenger exists in this list.

5.3 Pairwise Integeration Testing

5.3.1 Organize Relations

From the above relations deduced, we can conclude that our pair relations for our program are:

- Flight - Passenger
- Airline - Passenger
- Flight - Ticket
- Airline - Admin
- Ticket - Passenger
- Airline - Passenger

Now, from the above relations, we can now plot the relation diagram between the above nodes here 2

Table 3: Pairwise Integration Testing Matrix for Airline System Functionalities

Pairs	**Batch 1**	**Batch 2**	**Batch 3**
Flight - Passenger	*X*		
Airline - Passenger		*X*	
Flight - Ticket	*X*		
Airline - Admin			*X*
Ticket - Passenger			*X*

This table outlines the planned pairwise integration testing for the core functionalities of an airline system. The functionalities are represented by pairs of classes or modules that interact with each other. The "X" symbol indicates which pairs will be tested together in each batch.

The rationale behind the chosen batching is as follows:

- **Batch 1:** Focuses on core functionalities related to flights and passengers. This includes testing the interaction between the `Flight` class and the `Passenger` class, ensuring proper functionalities like booking flights and associating them with passengers. Testing the `Flight - Ticket` pair is also included in this batch as ticket creation is often linked to flight bookings.
- **Batch 2:** Focuses on functionalities related to Airline management and passengers. This includes testing the interaction between the `Airline` class and the `Passenger` class, potentially covering passenger signup, login, and profile management functionalities within the airline system.
- **Batch 3:** Focuses on functionalities related to tickets and passengers. This includes testing the interaction between the `Ticket` class and the `Passenger` class, ensuring aspects like ticket issuance and passenger access to their tickets work as expected.

The separation of functionalities into batches allows for a more focused testing approach. It also enables efficient resource allocation during the testing process.

This testing plan can be further refined and adjusted as the development process progresses and a clearer understanding of system interactions emerges.

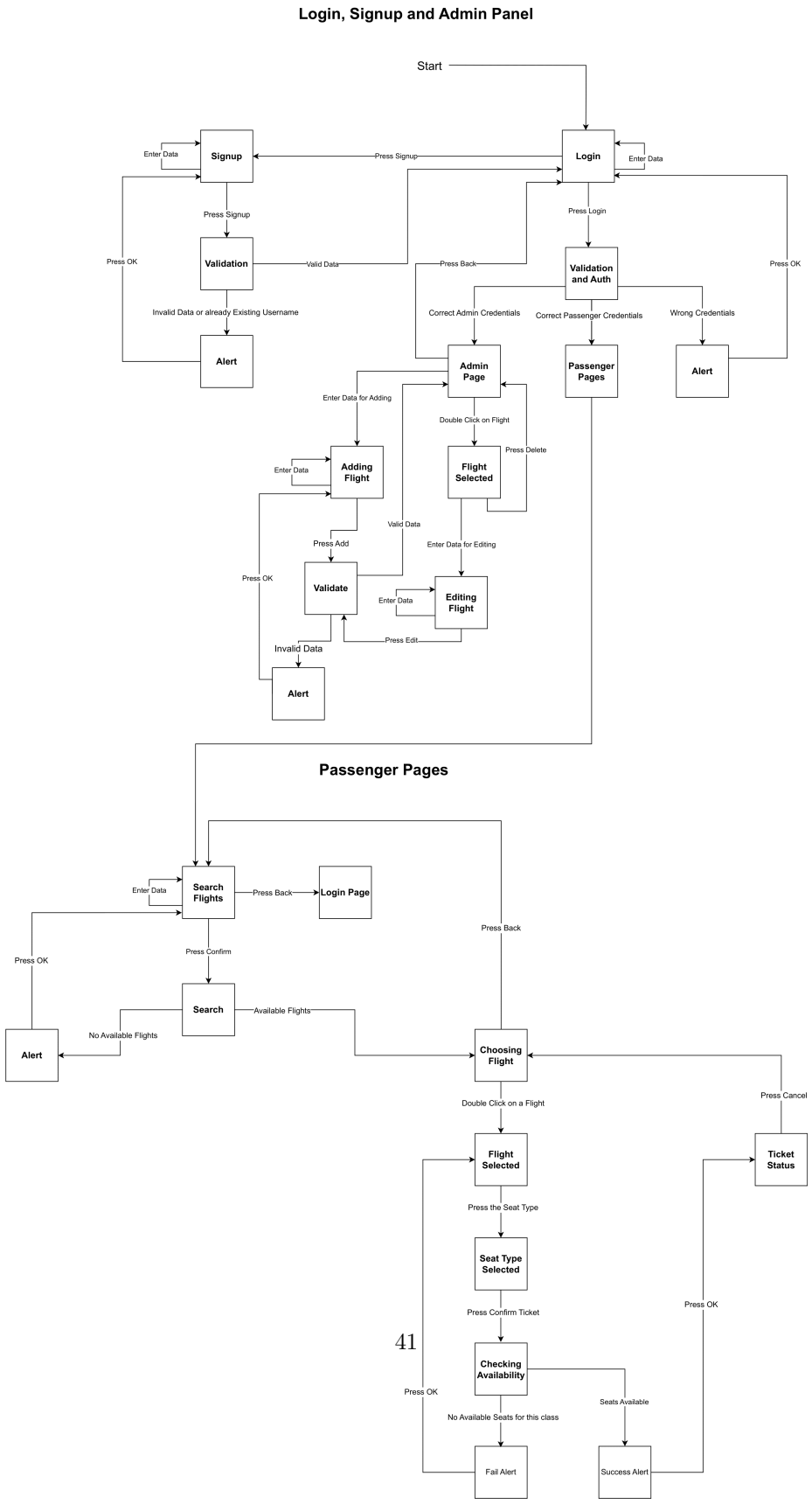

Figure 1: FSM GUI Testing

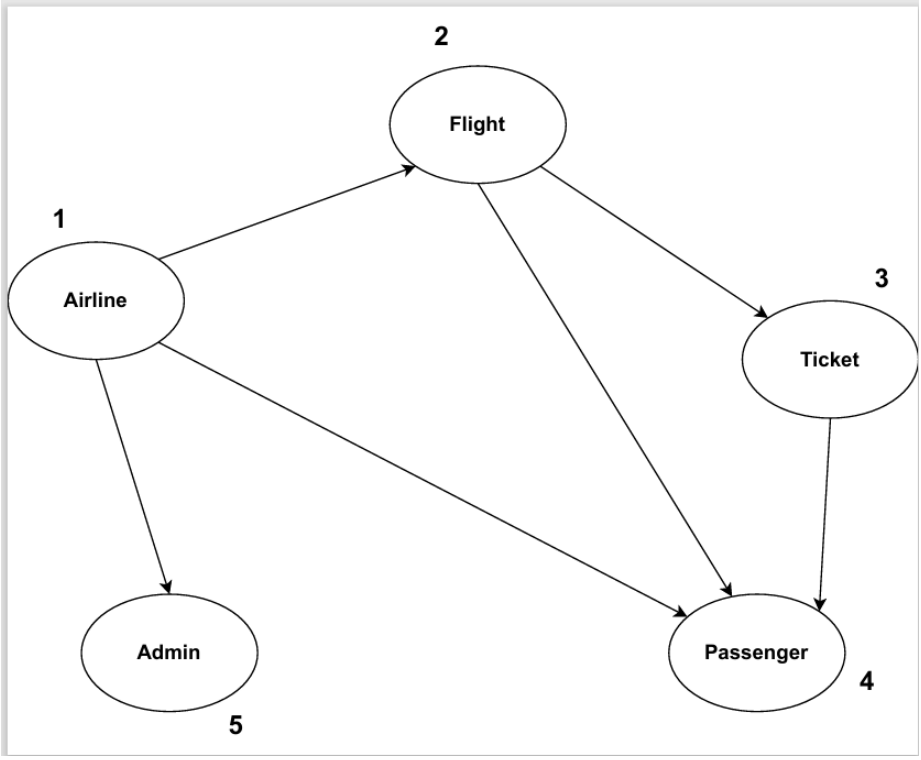

Figure 2: Relation Diagram