Rayan Khales

Airline reservation system (Ryanair):

Scenario:

The Airline Reservation System (ARS) is a program that helps manage flight bookings for both passengers and admins. Passengers can search for flights, book tickets, and make changes to their bookings, like canceling or updating them. Admins check and confirm bookings, keep track of passenger details, and manage all the information about flights, pilots, crew members, and booking reports.

Expected Output:

- Passengers can view available flights, book, cancel, or update reservations.
- Admins can create, manage, and view passenger and flight data.
- The system provides confirmations, error messages, and system-generated reports.
- Data is stored and retrieved via text I/O.

Planned functionalities:

The ARS demonstrates the following functionalities:

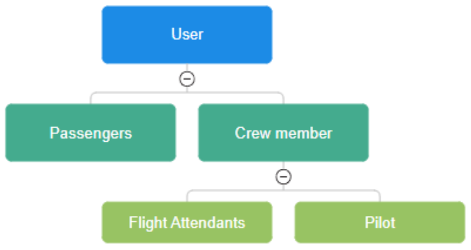
Passengers

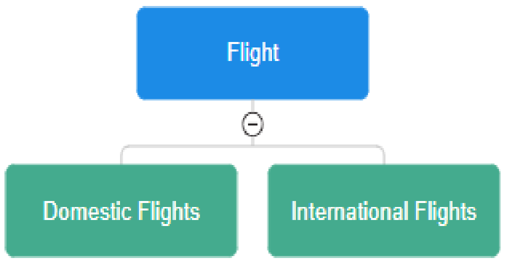
- Search flights by route, date, or airline.
- Book flights and receive a unique booking ID.
- Cancel or modify existing bookings.
- View personal booking history.

Administrators

- Add/edit/delete flight information.
- Manage booking confirmations.
- Maintain records of passengers, crew, and captains.
- Generate daily/weekly/monthly transaction reports.

Hierarchies:





Interface:

- **User Interface**: This interface will define common actions for all users in the system, such as viewing flight information and managing bookings. It is needed to make sure that both Passengers and Admins can perform basic user operations in a consistent and organized way. By implementing this interface, we ensure that shared behaviors between different types of users follow the same structure.

- **Bookable Interface**: This interface will define actions related to booking flights, such as booking a flight and canceling a reservation. It is needed to make sure any class that deals with reservations (like the Passenger class) can handle bookings in a clear and standard way. This also helps keep the booking-related logic separate from other parts of the program.

- **Searchable Interface**: This interface will define methods for searching flights by route or date. It is needed to allow classes like Passenger to perform flight searches using a standard approach. This makes the search functionality easy to manage and reuse across different parts of the system.

Runtime Polymorphism:

A getFlightDetails() is called, it will call the correct version of the method based on whether the flight is a DomesticFlight or InternationalFlight (Override).

Text I/O:

- **Admin Class**

The Admin class uses text IO to save the transaction report to a text file for record-keeping and future reference.

- Passenger Class

After a booking is made, the booking information is written in a .txt file using text IO. This ensures that all bookings are recorded and can be referred to later. Like a regular flight ticket.

Comparable / Comparator:

- The Comparable interface will be implemented in the Flight class to provide a sorting order for flights. Sorting by either departure date or flight number (have not decided).

- The Comparator interface is used when you need multiple ways to sort flights based on different criteria. Implemented in the FlightComparator Class where you could compare by price which would be useful for passengers or destination or flight number which is useful for admins.

Class diagram:

Class: Flight (abstract)
Implements: Comparable<Flight>, Searchable

- **Fields:**
  o flightNumber: String
  o departureTime: String
  o arrivalTime: String
  o origin: String
  o destination: String
  o price: double
- **Methods:**
  - compareTo(Flight):
  - search(String keyword):

Subclass: DomesticFlight extends Flight

- **Fields:**
  o region: String
- **Methods:**
  - search(String keyword): (overridden)

Subclass: InternationalFlight extends Flight

- **Fields:**
  o passportRequired: boolean
- **Methods:**
  - search(String keyword): (overridden)

Interface: Searchable

- **Methods:**
  - search(String keyword):

Interface: Bookable

- **Methods:**
  - book():
  - cancel():

Class: Booking
Implements: Bookable

- **Fields:**
  o bookingId: String
  o flight: Flight
  o passenger: Passenger
  o seatNumber: String
  o status: String
- **Methods:**
  - book():

- cancel():

Class: Passenger

- **Fields:**
  o name: String
  o email: String
  o passportNumber: String
- **Methods:**
  - bookFlight(Flight):
  - cancelBooking(String):
  - saveFlightTicket(): *(uses TextIO to write ticket info to a file)*

Class: Admin

- **Fields:**
  o name: String
  o employeeId: String
- **Methods:**
  - generateReport(): *(uses TextIO to write report data)*
  - confirmBooking(Booking):
  - managePassengerRecords():

Class: FlightComparatorByPrice
Implements: Comparator<Flight>

- **Methods:**
  - compare(Flight f1, Flight f2):

Class: FlightComparatorByFlightNumber
Implements: Comparator<Flight>

- **Methods:**
  - compare(Flight f1, Flight f2):

Implementations for deliverable 2:

Class Structures:
Implementation of the following classes:

- Flight (abstract class)
- DomesticFlight
- InternationalFlight
- Passenger
- Admin
- Booking

• Interface:

- Bookable: This interface will be implemented by the Booking class to ensure a standardized structure for booking and cancellation functionality.

• Text-Based Interaction:

- The write() method in the Admin class to generate a flight transaction report.
- The saveFlightTicket() method in the Passenger class to save a copy of the booked flight ticket.
  (Both use TextIO for reading/writing to files.)

• JUnit Tests:
The following methods will be unit tested:

- book() — ensures a flight can be successfully booked
- cancel() — ensures a booking can be canceled
- generateReport() — checks if the admin report is written correctly
- saveFlightTicket() — tests that the ticket file is saved properly
- search(String keyword) — verifies the search functionality in flights works as expected