Software Quality
(SOFE 3980U)

Project: Deliverable 1 & Report

Group Members: (CRN 73385 Group 19)

Name	Student ID
Qamar Irfan	100785387
Rohan Radadiya	100704614
Sibi Sabesan	100750081
Syed Airaj Hussain	100789134

Due Date: Mar 15, 2024

Github: https://github.com/rohanradadiya/SQ-Project-Deliverable-1

Abstract:

This report outlines the initial phase of creating a flight ticket booking app for an airline, focusing on Test-driven development (TDD) and Continuous Integration/Continuous Deployment (CI/CD). The goal is to deliver a user-friendly web app allowing users to book direct or multi-stop flights, one-way or round trip, while considering user preferences. Functional requirements include a simple web interface, weekly direct flight listing, flight duration calculation, and user preference customization. The project structure centers on three main classes: Flight, Booking, and User Information, with added Preferences. Test-driven development is used, starting with failing tests to guide development. Detailed test cases ensure reliability. Moving forward, we'll progress to the green phase, where failing tests will guide implementation. Continuous integration will automate testing and deployment, aiding rapid development and improving code quality.

Project Structure:

Classes:

Class Diagram Structure:	
Flight:	● origin: String (the origin airport code) ● destination: String (the destination airport code) ● departureTime: DateTime (the departure time of the flight) ● arrivalTime: DateTime (the arrival time of the flight) ● duration: TimeSpan (the duration of the flight) ● isDirect: Boolean (indicates whether the flight is direct or multi-stop)
Booking:	● flight: Flight (the flight being booked) ● user: User (the user making the booking) ● type: BookingType (one-way or round trip) ● numberOfPassengers: int (the number of passengers for the booking) ● totalPrice: float (the total price of the booking) ● departureDate: DateTime (the departure date for the booking) ● returnDate: DateTime (the return date for round trip bookings)
User Information:	● username: String (the username of the user) ● email: String (the email address of the user) ● password: String (the password of the user) ● preferences: Preferences (the user's preferences)
Preferences:	● timeFormat: TimeFormat (the preferred time format for the user, 12-hour or 24-hour) ● seatPreference: SeatPreference (the preferred seating option for the user)
Other variables/enums:	● BookingType: Enum { OneWay, RoundTrip } ● TimeFormat: Enum { TwelveHour, TwentyFourHour } ● SeatPreference: Enum { Window, Aisle }

We designed our classes to cover functionalities for our flight ticket booking app. We created three main classes: Flight, Booking, and User Information.

Flight Class: Represents individual flights with details like origin, destination, departure, arrival time, duration, and direct or multi-stop status. This class manages flight information effectively.

Booking Class: Handles flight reservations, including booked flight, user details, booking type (one-way or round trip), number of passengers, total price, departure and return dates. It organizes booking-related data efficiently.

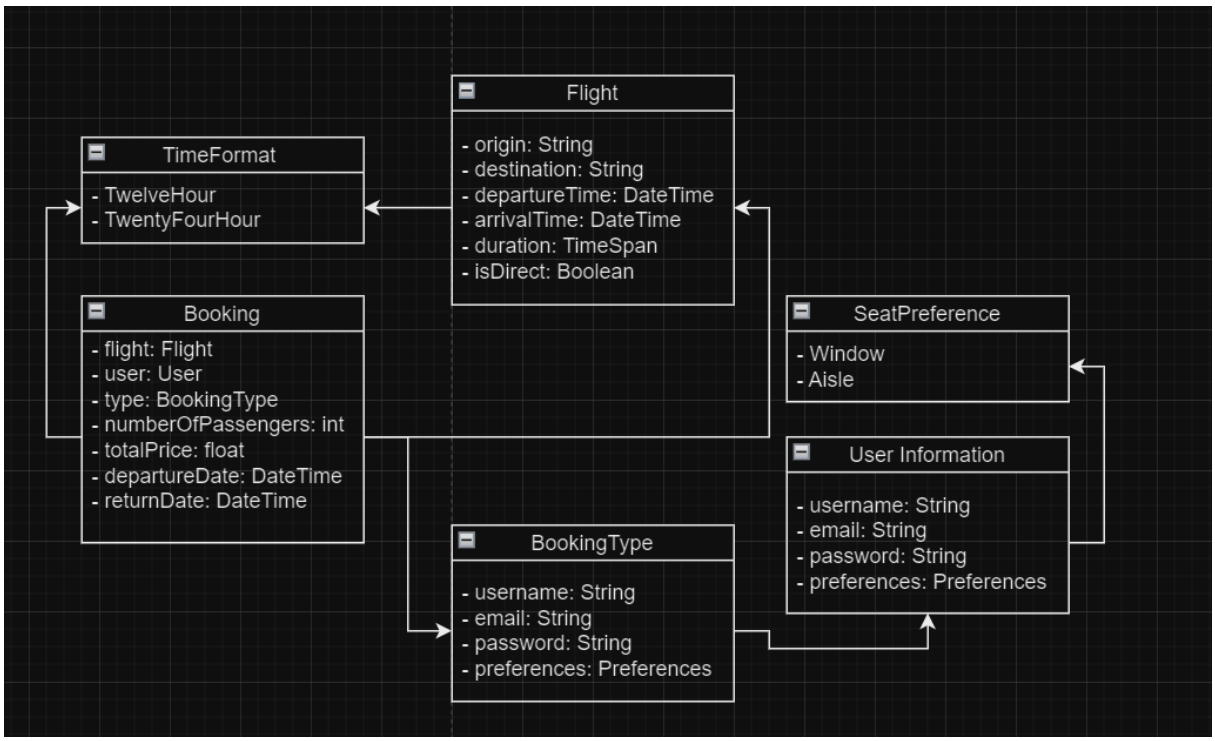
User Information Class: Manages user details and preferences such as username, email, password, and preferred time format and seating option. This class ensures personalized and secure user experiences.

We also introduced the Preferences class to manage user-specific settings like time format and seating preferences. This modular approach enhances code clarity and maintainability by separating user preference concerns.

Additionally, we defined two enums: TimeFormat (TwelveHour and TwentyFourHour) and SeatPreference (Window and Aisle) to standardize representation and management of user preferences.

These classes and enums collectively provide comprehensive functionality for our flight ticket booking app, enhancing flexibility and user-friendliness.

Class Diagram/UML:



To create our UML class diagram for the flight ticket booking application, relationships between classes were established. For instance, a Booking object is associated with a Flight object. We also used enumerations like TimeFormat and SeatPreference to represent predefined values.

After organizing all elements, a clear and concise UML class diagram was produced, serving as a blueprint for implementation.

Explanation of Project:

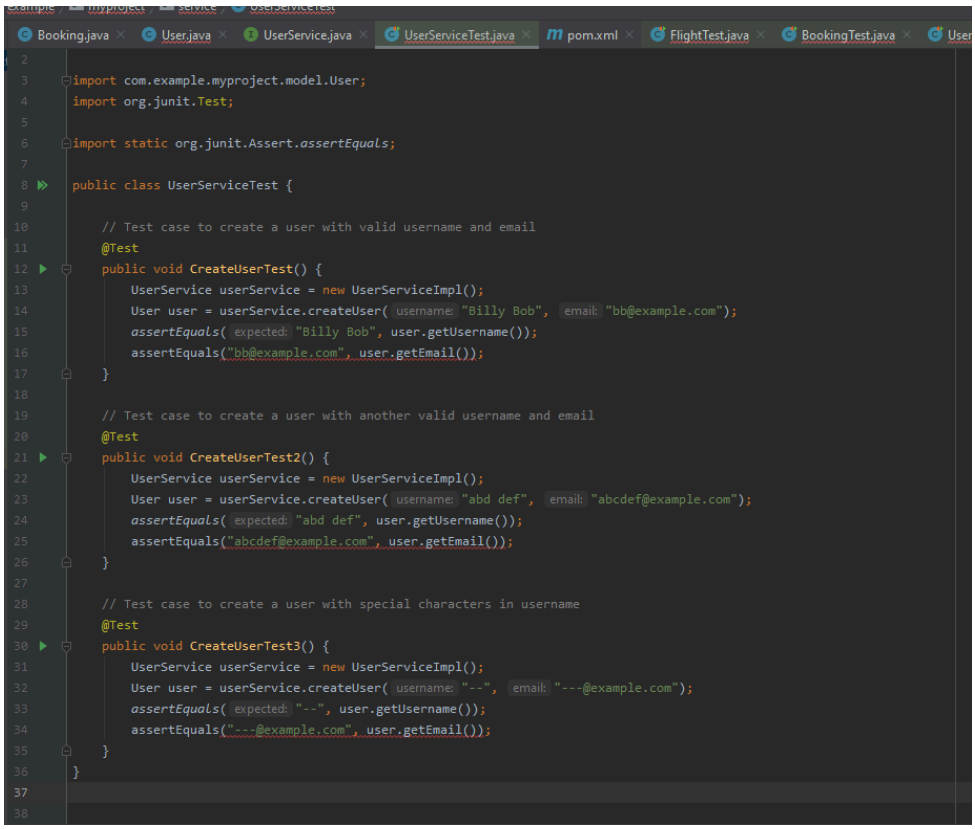
The overall project deals with the very early stages of the “Ticket Booking Application”. Before actually starting to implement the functionalities, it is very crucial to understanding the testing concepts at this stage. By conducting various, iterative and a wide variety of tests, it is

much more efficient and easier to implement a functional application afterwards. Errors are also very minimal or even nonexistent in the future stages.

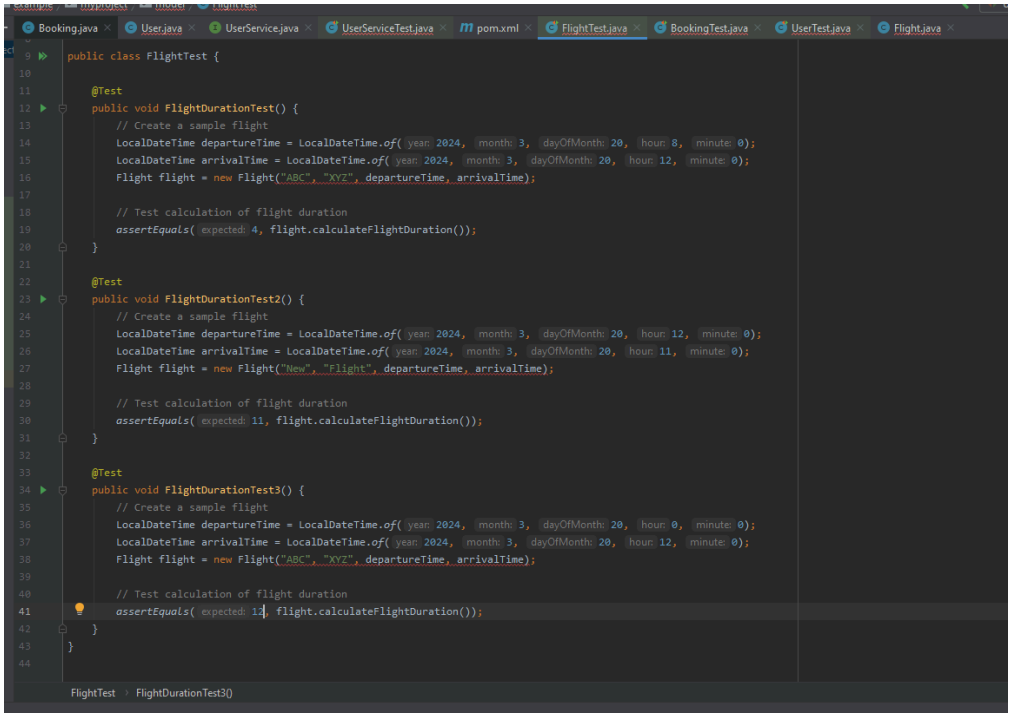
Furthermore, this type of approach, TDD, is what is being discussed here. Because the tests are being written before the actual functional code, it is going to show up and be calculated as failures/failed tests. That is why we were in the “red” phase of testing, where we write failing tests to ensure that the tests are being written properly, before moving onto the “green” phase, which is where we tend to write minimal code in order to pass the tests. And following that phase, we would then move onto the refactor phase where we would work on improving the overall code and design in order to also be able to improve the readability. Cleaning up the code would also be an essence at that stage.

All in all, at this stage of the project, we have outlined the implementations and what was done in the “red” phase of TDD, and the required tests which will be utilized in the later stages. The future stages will have functional code, and different stages of refactoring as well in order to make the overall ticket booking application functional. There will also be additional tests written and implemented into the code to make the program more efficient and user-friendly as well.

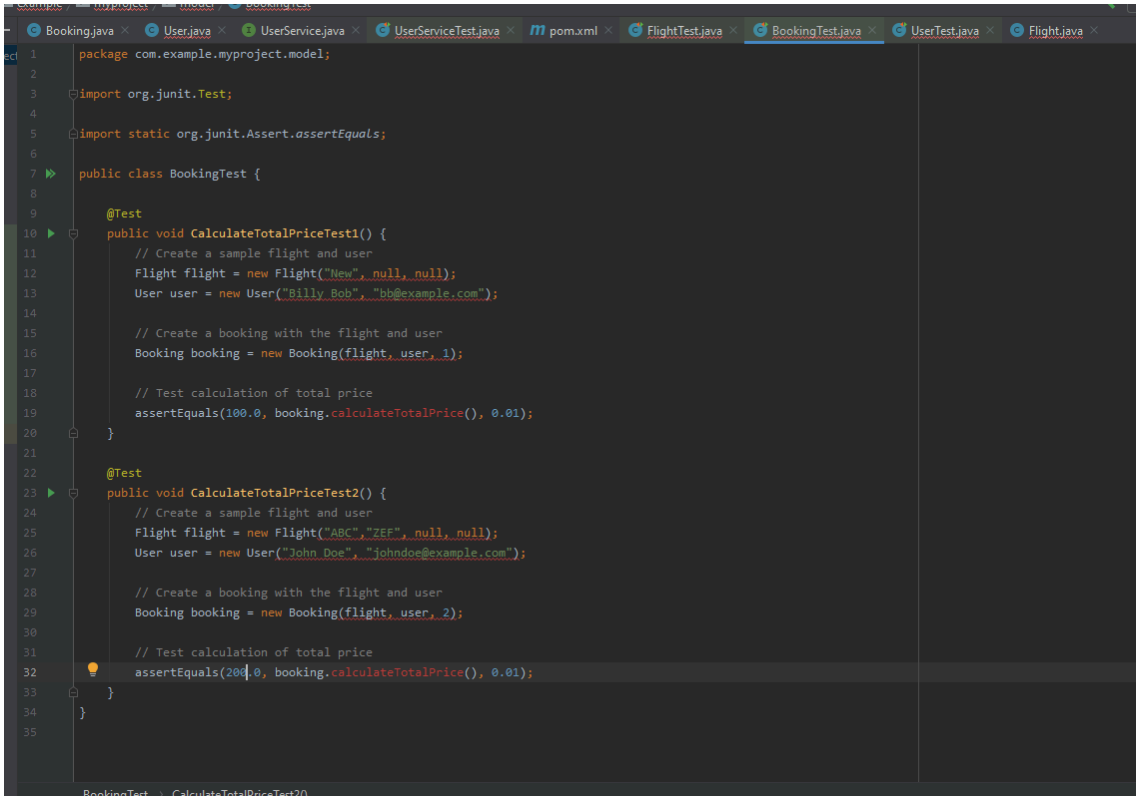
Screenshots of Test Cases:



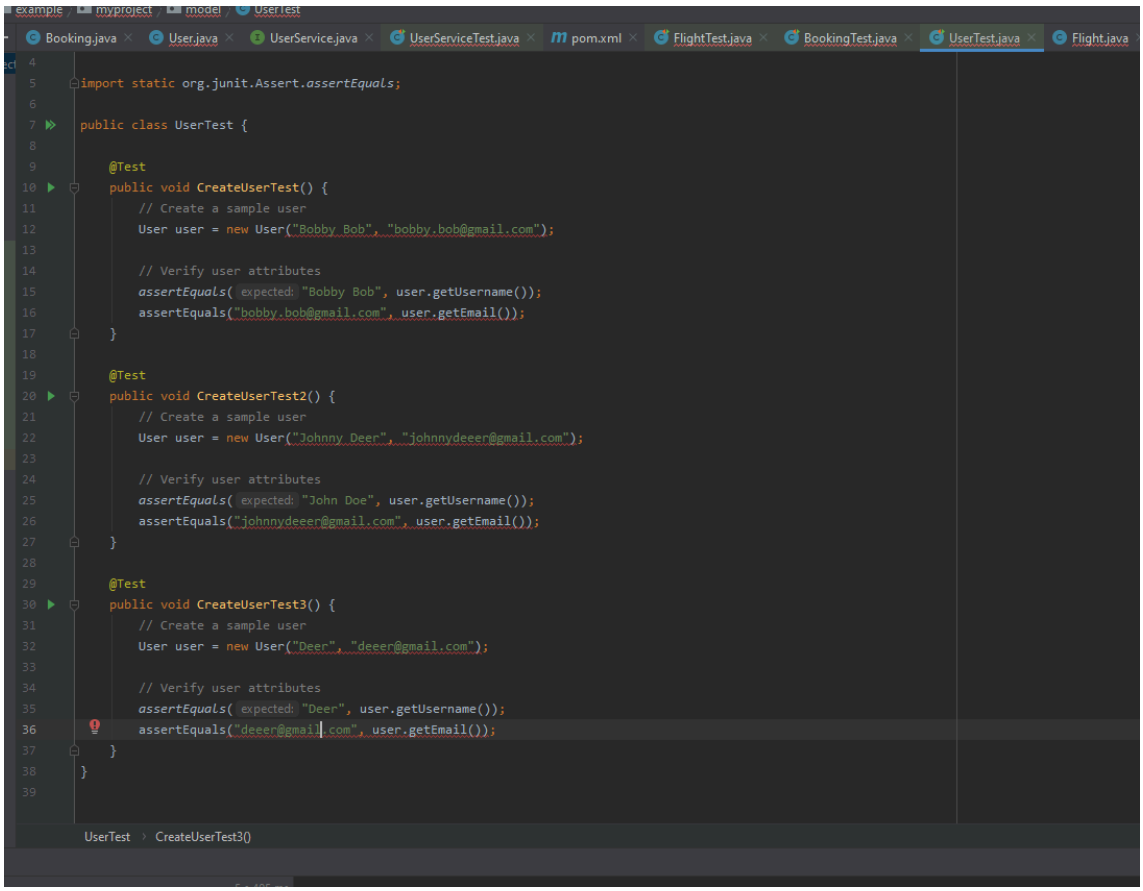
```java
import com.example.myproject.model.User;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class UserServiceTest {

    // Test case to create a user with valid username and email
    @Test
    public void CreateUserTest() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "Billy Bob",  email: "bb@example.com");
        assertEquals( expected: "Billy Bob", user.getUsername());
        assertEquals("bb@example.com", user.getEmail());
    }

    // Test case to create a user with another valid username and email
    @Test
    public void CreateUserTest2() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "abd def",  email: "abcdef@example.com");
        assertEquals( expected: "abd def", user.getUsername());
        assertEquals("abcdef@example.com", user.getEmail());
    }

    // Test case to create a user with special characters in username
    @Test
    public void CreateUserTest3() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "--",  email: "---@example.com");
        assertEquals( expected: "--", user.getUsername());
        assertEquals("---@example.com", user.getEmail());
    }
}
```



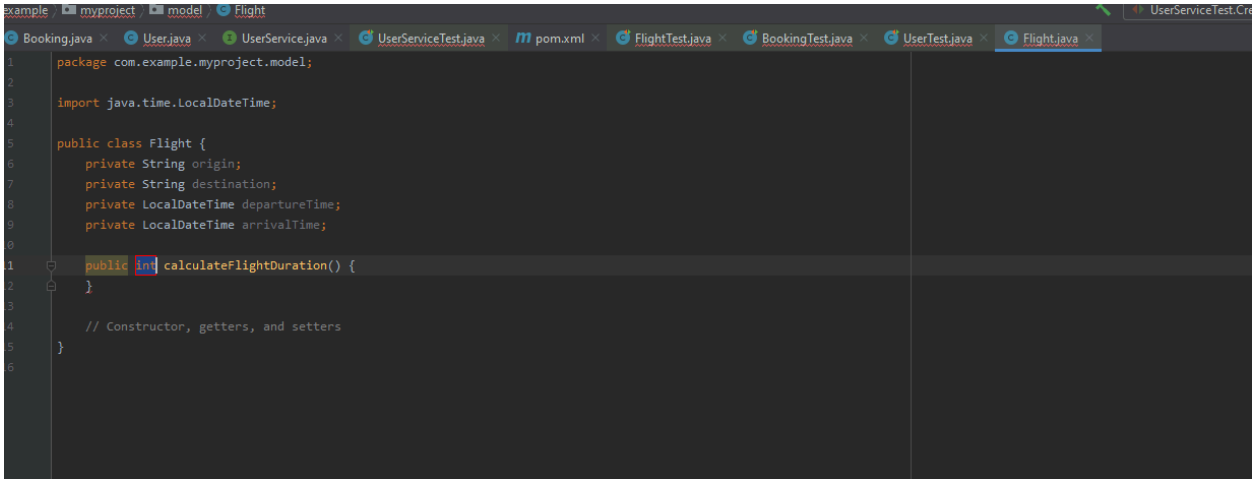
```java
public class FlightTest {

    @Test
    public void FlightDurationTest() {
        // Create a sample flight
        LocalDateTime departureTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 8,  minute: 0);
        LocalDateTime arrivalTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 12,  minute: 0);
        Flight flight = new Flight("ABC", "XYZ", departureTime, arrivalTime);

        // Test calculation of flight duration
        assertEquals( expected: 4, flight.calculateFlightDuration());
    }

    @Test
    public void FlightDurationTest2() {
        // Create a sample flight
        LocalDateTime departureTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 12,  minute: 0);
        LocalDateTime arrivalTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 11,  minute: 0);
        Flight flight = new Flight("New", "Flight", departureTime, arrivalTime);

        // Test calculation of flight duration
        assertEquals( expected: 11, flight.calculateFlightDuration());
    }

    @Test
    public void FlightDurationTest3() {
        // Create a sample flight
        LocalDateTime departureTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 0,  minute: 0);
        LocalDateTime arrivalTime = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 20,  hour: 12,  minute: 0);
        Flight flight = new Flight("ABC", "XYZ", departureTime, arrivalTime);

        // Test calculation of flight duration
        assertEquals( expected: 12, flight.calculateFlightDuration());
    }
}
```



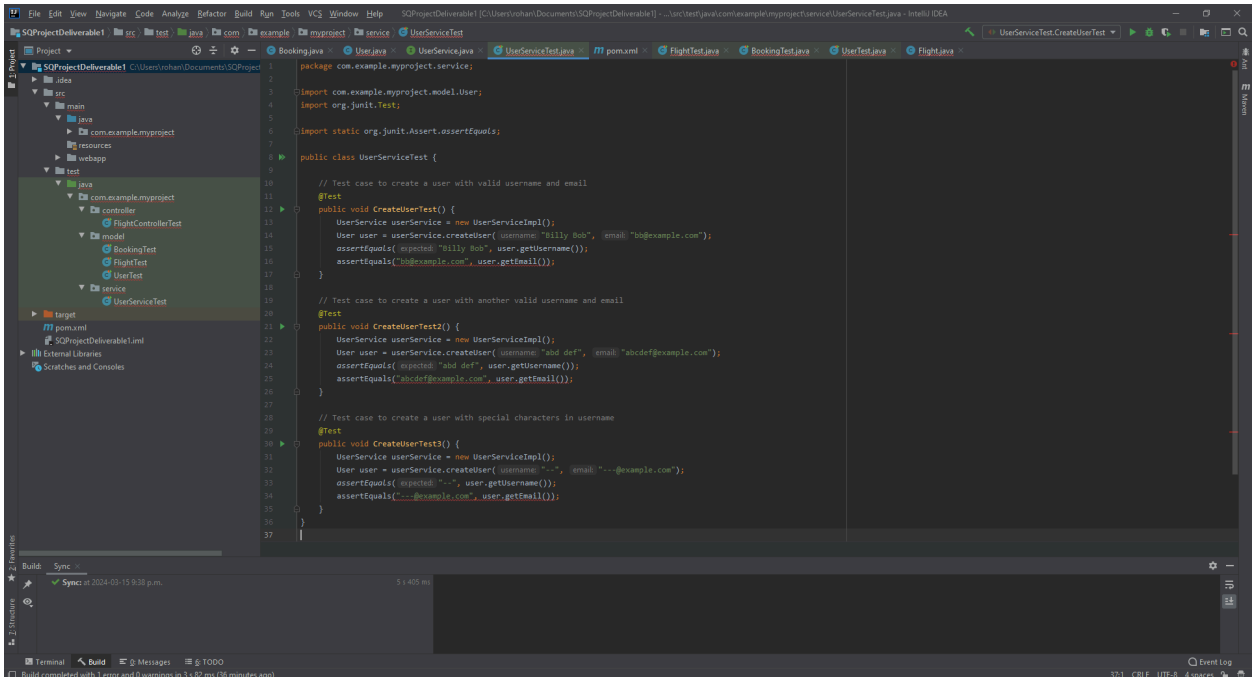
```java
package com.example.myproject.model;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class BookingTest {

    @Test
    public void CalculateTotalPriceTest1() {
        // Create a sample flight and user
        Flight flight = new Flight("New", null, null);
        User user = new User("Billy Bob", "bb@example.com");

        // Create a booking with the flight and user
        Booking booking = new Booking(flight, user, 1);

        // Test calculation of total price
        assertEquals(100.0, booking.calculateTotalPrice(), 0.01);
    }

    @Test
    public void CalculateTotalPriceTest2() {
        // Create a sample flight and user
        Flight flight = new Flight("ABC","ZEF", null, null);
        User user = new User("John Doe", "johndoe@example.com");

        // Create a booking with the flight and user
        Booking booking = new Booking(flight, user, 2);

        // Test calculation of total price
        assertEquals(200.0, booking.calculateTotalPrice(), 0.01);
    }
}
```



```java
import static org.junit.Assert.assertEquals;

public class UserTest {

    @Test
    public void CreateUserTest() {
        // Create a sample user
        User user = new User("Bobby Bob", "bobby.bob@gmail.com");

        // Verify user attributes
        assertEquals( expected: "Bobby Bob", user.getUsername());
        assertEquals("bobby.bob@gmail.com", user.getEmail());
    }

    @Test
    public void CreateUserTest2() {
        // Create a sample user
        User user = new User("Johnny Deer", "johnnydeeer@gmail.com");

        // Verify user attributes
        assertEquals( expected: "John Doe", user.getUsername());
        assertEquals("johnnydeeer@gmail.com", user.getEmail());
    }

    @Test
    public void CreateUserTest3() {
        // Create a sample user
        User user = new User("Deer", "deeer@gmail.com");

        // Verify user attributes
        assertEquals( expected: "Deer", user.getUsername());
        assertEquals("deeer@gmail.com", user.getEmail());
    }
}
```



```java
package com.example.myproject.model;

import java.time.LocalDateTime;

public class Flight {
    private String origin;
    private String destination;
    private LocalDateTime departureTime;
    private LocalDateTime arrivalTime;

    public int calculateFlightDuration() {
    }

    // Constructor, getters, and setters
}
```



```java
package com.example.myproject.service;

import com.example.myproject.model.User;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class UserServiceTest {

    // Test case to create a user with valid username and email
    @Test
    public void CreateUserTest() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "Billy Bob",  email: "bb@example.com");
        assertEquals( expected: "Billy Bob", user.getUsername());
        assertEquals("bb@example.com", user.getEmail());
    }

    // Test case to create a user with another valid username and email
    @Test
    public void CreateUserTest2() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "abd def",  email: "abcdef@example.com");
        assertEquals( expected: "abd def", user.getUsername());
        assertEquals("abcdef@example.com", user.getEmail());
    }

    // Test case to create a user with special characters in username
    @Test
    public void CreateUserTest3() {
        UserService userService = new UserServiceImpl();
        User user = userService.createUser( username: "--",  email: "---@example.com");
        assertEquals( expected: "--", user.getUsername());
        assertEquals("---@example.com", user.getEmail());
    }
}
```

As seen in the screenshots above, there were several test cases done in the various files in order to test the functionalities of not the code, but the tests specifically. Because the functional code is not included in the overall project right now, the tests aren't meant to pass, as we are currently in the red phase of TDD. Moving onto the next deliverable, we will be onto the green phase, and also refactor the code to make it more readable and functional as a whole.

Conclusion:

In conclusion, our flight ticket application provides an easy to use travel booking option. With concepts like Test-driven development and CI/CD concepts, we were able to properly structure the future plans for the code. Our main class design, including Flight, Booking, and User Information is supported by classes like Preferences and enums such as TimeFormat and SeatPreference for flexibility for users. The UML class diagram visually represents our application's structure, guiding implementation and team communication. We'll refine our app based on feedback to meet users to uphold a high-quality solution.

Group Member Tasks:

Name	Student ID	Detailed Tasks	Work %
Qamar Irfan	100785387	All the functional requirements for the application and any assumptions. Listed all the functional requirements for our application with assumptions and did in depth research of the system by following all the requirements, coordinated with the group and assigned tasks and helped in the overall report of the project.	25%
Rohan Radadiya	100704614	Design of project Modules, a GitHub repo with the initial implementation of the unit and integration	25%

		test. Was in charge of the design of the project modules and helped with the Github repo with implementation of testing (screenshots of test cases), also helped with the overall report of the project.	
Sibi Sabesan	100750081	The design of integration tests for modules, a GitHub repo with the initial implementation of the unit and integration test. Was in charge of the design of integration test of the modules as well as adding into the Github repo (screenshots of test cases), as well as helping with the overall report of the project.	25%
Syed Airaj Hussain	100789134	Design of unit tests for each method : Was in charge of the designs of the unit tests for each of the methods in the application,helped in making the Class UML diagram, and helped with the overall report of the project adding in key points.	25%

References

Ammann, P., & Offutt, J. (2017). *Introduction to software testing*. Cambridge University Press.