GROUP NAME : FUTURE BUILDERS

Github: https://github.com/PracDuckling/TravelSmartHub

PROBLEM STATEMENT :

TravelSmart aims to automate the search process and presentation of flight search results to its customers by developing an application that efficiently retrieves and displays flight information based on user input.

REQUIREMENTS:

1. Data Storage:
   - ➢ Flight information is stored in text files in a directory, with each file containing data in CSV format.
   - ➢ Each file corresponds to a specific airline carrier or company.
2. Data Parameters:
   - ➢ The flight data includes the following parameters: Flight Number, Source City, Destination City, Fare, and Duration.
3. Search Functionality:
   - ➢ The application should allow users to input a Source City and a Destination City.
   - ➢ The application should search all available flights between the specified cities.
4. Output:
   - ➢ The search results should display details of all flights available between the given Source and Destination cities.
   - ➢ The results should be sorted based on one of the following criteria:
     - ▪ Fare
     - ▪ Duration
     - ▪ Both Fare and Duration

5. Performance Considerations:
    - The application should be designed to handle multiple users simultaneously, ensuring efficient performance in a real-time environment.

SOLUTION:

The program adheres to key principles of object-oriented design (**OOD**), such as **encapsulation, abstraction, and modularity**. The program separates different concerns into methods, ensuring better maintainability and readability.

The code follows Java's **standard naming conventions and best practices**:

1. Class and method names: The class names are written in PascalCase (Flight, FlightSearch), and method names are in camelCase (loadFlightData, searchFlights), following Java conventions.
2. Variable names: Variable names are descriptive and written in camelCase (flightData, sourceCity, destinationCity, etc.), making the code easy to understand.

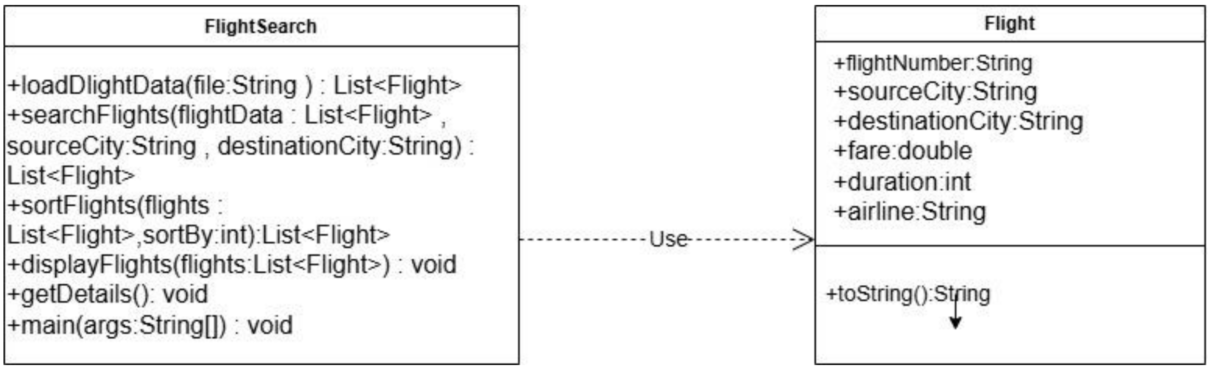
Design Patterns:
The most suitable design pattern for our current implementation is the **Strategy Pattern.**

Reasoning for Choosing Strategy Pattern

- Sorting Variability: The main dynamic aspect of our program is sorting the flight data. The user can choose between sorting by Fare, Duration, or Fare and Duration. These are interchangeable algorithms, and the Strategy Pattern is ideal for such situations where we want to define a family of algorithms and allow the user to select one at runtime.

- Decoupling the Sorting Logic: By applying the Strategy Pattern, we can decouple the sorting logic from the FlightSearch class and encapsulate it in separate classes, each representing a different sorting strategy. This not only makes the code more maintainable but also allows us to easily add new sorting strategies in the future without modifying the existing code.
- Maintainability and Extension: The Strategy Pattern allows us to add new sorting strategies without changing the core logic of the program. If, in the future, we decide to add more sorting criteria (like by Airline or Flight Number), we can simply create new strategy classes.

Class Diagram:


FlightSearch
+loadDlightData(file:String) : List<Flight>
+searchFlights(flightData : List<Flight> , sourceCity:String , destinationCity:String) : List<Flight>
+sortFlights(flights : List<Flight>,sortBy:int):List<Flight>
+displayFlights(flights:List<Flight>) : void
+getDetails(): void
+main(args:String[]) : void
Use
Flight
+flightNumber:String
+sourceCity:String
+destinationCity:String
+fare:double
+duration:int
+airline:String
+toString():String

Performance in real-time environment:

Multithreading in our flight search application allows multiple users to search for flights concurrently by running each user interaction in a separate thread. The UserTask class implements Runnable to define the task, and shared resources like flightData and cityMapping are accessed safely using synchronization. In the main method, 10 threads are created and started, each executing the run method of UserTask, ensuring efficient and concurrent user interactions while preventing data inconsistencies.

Time and Space complexity :

The overall **time complexity** for a typical flow (load -> search -> sort -> display) is dominated by the sorting step, which is **O(n log n).**

The overall **space complexity** for the program is **O(n)** as we store a list of n flights, which is the most significant space usage in our program.

Unit Testing for TravelSmart using JUnit5

The test cases for the ***FlightSearch*** class are designed to ensure that the methods for loading city mappings, loading flight data, searching for flights, and displaying flight information work correctly.

This approach allows us to focus on testing the logic of each method in isolation. For example, the ***testLoadCityMapping*** method verifies that the ***loadCityMapping*** method correctly parses city mappings from a CSV file and handles case insensitivity.

Similarly, the ***testLoadFlightData*** method ensures that the ***loadFlightData*** method accurately reads flight data and populates the list of flights.

In addition to verifying the correctness of the methods, these test cases also help identify potential issues and edge cases.

For instance, the ***testSearchFlights*** method checks that the ***searchFlights*** method returns the correct flights for a given source and destination, while the ***testDisplayFlights*** method ensures that the flight information is displayed in the expected format. By thoroughly testing these methods, we can be confident that the ***FlightSearch*** class will function as intended in various scenarios. This documentation provides a clear understanding of the purpose and functionality of each test case, making it easier to maintain and extend the codebase in the future.

CONCLUSION:

The Flight Search application is a well-structured, object-oriented solution for managing and querying flight data, leveraging modern Java features like streams and comparators for efficient searching and sorting. It follows good coding practices, including clear separation of concerns and extensibility through the Strategy Pattern, making it easy to add new sorting criteria in the future. While the program performs well for moderate datasets, it could benefit from improved error handling, validation, and performance optimizations for larger datasets, such as parallel processing. Overall, the program provides a solid foundation with opportunities for further refinement, particularly in terms of scalability and user experience.