Flight Control Tower Program

JUXT Technical Test

Haaris Iqbal - December 2022

Overview

This application is a maven project made in IntelliJ and written in Java. It contains a POM file, seven classes, two test suites, three interfaces, an enumerated class and a README.md file. The project has been developed and organised in an MVC design pattern.

Test Driven Development methodology was used for multiple components, with a total of 47 JUnit tests. Appropriate Javadoc has been added to every functional file, and all Java code conforms to Google Checkstyle.

Thoughtful design decisions were made throughout the process of development. A lot of effort was put into both robustness of the backend, and UI neatness / helpfulness.

Program features all functionality required from specification (and more):

- Events can be added and updated.
- Events can be deleted.
- Appropriate error handling for incorrect inputs, duplicate inputs, and invalid inputs.
- Status table displays status in required format.
- Status table will always update to reflect additions, detail changes, deletions.
- Status table can display a status for any time stamp in the past or future and will take updates into account.
- Custom commands for help, guide, listing all logged events, quitting the program.

Running Project

The project is titled "FlightControlTower" The dependencies needed for this project are downloaded automatically. The main entry point of the application is the "App.java" class. Running the main method from this class will start the program.

Running Tests

All tests were written with JUnit 5 Jupiter – thus for each JUnit test suite, all tests can be run individually or at once using JUnit 5. Earlier versions of JUnit should also work.

Use of Program

All user interaction with the program occurs through the console.

When running the program, it will conform to the functionality required in the specification sheet. However, there are some reserved commands that have been included, which can be typed at any stage of the program loop:

- "help": To display commands, and other possible inputs.
- "events": To display all events that have been logged.
- "guide": A display a user guide on how inputs work for this program.
- "quit": To quit the program.

Other than these commands, the program works as follows:

1. An event can be added or updated by entering in the following format:

Fxxx xxx ORIGIN DESTINATION Event-Type Time-Stamp Fuel-Delta

To break this down:

- 'Fxxx' is the PlaneID and must be four characters long.

- 'xxx' is the Plane Model and must be three characters long.
- 'ORIGIN' is the origin city of the flight. It may not be the same as the destination city.
- 'DESTINATION' is the destination city of the flight. It may not be the same as the origin city.
- 'Event-Status' is the type of event that has taken place. There may only be three event types:
  - 'Re-Fuel' indicating that the plane is now awaiting takeoff.
  - 'Take-Off' indicating that the plane is now in-flight.
  - 'Land' indicating that the plane has landed.
- 'TimeStamp' is the time stamp of when the event has taken place.
  - Time stamps must be in the format of: yyyy-MM-ddThh:mm:ss
  - An example of a valid time stamp is: 2021-03-29T14:00:00
- 'Fuel-Delta' is the change in fuel that has occurred due to the event. It must be a number.

2. Next, an event can be removed by providing a valid Plane ID and time stamp. The format should be as follows:

```
Fxxx yyyy-MM-ddThh:mm:ss
```

3. Finally, a status table can be viewed for any timestamp simply by entering in the following format:

```
yyyy-MM-ddThh:mm:ss
```

All of these formats conform to the inputs described in the project specification.

Notable Features and Functionality

- The project has been designed in the MVC design pattern, allowing for an evenly distributed amount of responsibility to multiple key components. Backend and frontend functionality are distinct, and this pattern allows for the potential of vast expansion.

- Dependency injection has been incorporated throughout.

- Required functionality has been developed. For example, Status tables can be produced for any time stamp, and this will reflect a correct series of events.

- Extra functionality has been included, such as ability to log events in any order, preventing duplicate events, and custom commands to view all events logged thus far, quit, etc.

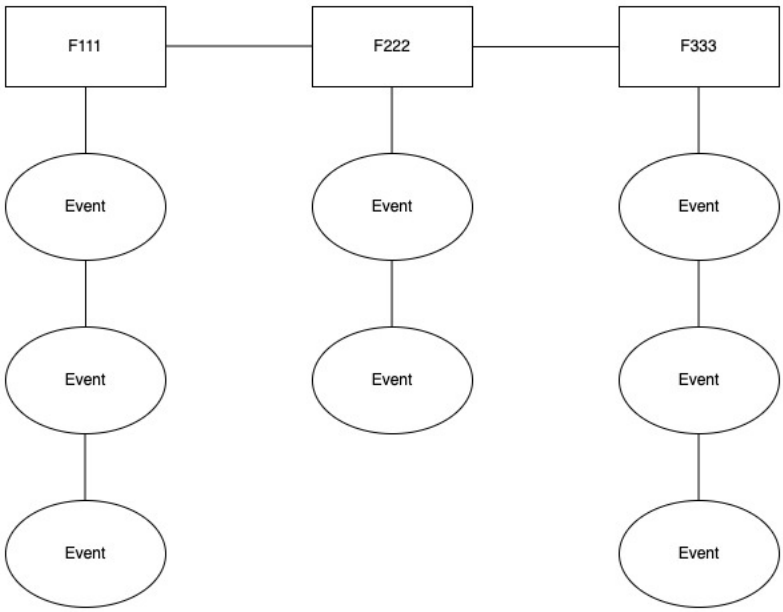
- Error handling taken into consideration for all elements and use cases.

- Included appropriate Interfaces, Enums, Tests.

- For tests, every feature has been tested and demonstrated to be working as expected. All features are also shown to respond to errors appropriately.

- All Events are handled as objects. The reason this was done was because an event incorporates multiple details of different types, and events need to be transferred, accessed, manipulated throughout the program. Thus, it makes for an appropriate DTO.

- All event storage is handled by the DAO. A HashMap of PlaneID to Array List of Events acts as the pseudo-database of this program. A visual representation of what this looks like is as follows:



- This program allows for non-linear entry of events, and thus events exist unordered without causing issues.

- Responsibility is distributed appropriately according to the MVC design pattern. The logic of interpreting events and constructing a status table, as well as validating details are handled by the Service Layer because these are business needs. The program is orchestrated by the controller, and all interaction is handled by View and UserIO.

- Using LocalDateTime from Java Time library to handle all time stamp related processing in events.

- In status table, time is formatted so seconds are displayed, even when zero.

- Javadoc and appropriate comments included throughout.

Improvements

There are numerous improvements and expansions that can be made to this program.

- Code clarity could be improved. In particular, Service Layer tests are dense and could be broken up into cleaner components.

- In some areas, TDD could be broken up a bit more. For example, in removal of an event, deletion of an entire flight's history could warrant a separate test case. Or in updating, updates of multiple elements could be separate from a single element update. Most critically overall, could have more tests to check event details outside of those needed for time stamps (e.g., more tests for destination, origin).

- Some aspects of the program could have neater implementations. For example, in calculating status table, 'if' checks are long and could be implemented in simpler ways.

- When calculating which event details to use for the production of the status table, one second is added to requested time stamp to ensure that results are inclusive. There may be more elegant solutions to solving this particular corner case problem.

- Could incorporate a database. The way storage is handled currently is basic. It exists as a HashMap in the DAO. Would ordinarily have incorporated a separate storage package.

- Could incorporate Spring Dependency Injection. In this case, the decision was made not to do so for the sake of reducing dependencies.

- In view, adding duplicate events currently uses the default success message despite being unsuccessful. This is an issue that should be resolved.

- Could incorporate more complex error handling (for example custom errors) to allow for more helpful error messages.

- Documentation could be improved. Should have produced a UML Diagram for the project and should have given a better high-level overview of the project. Additionally, certain key elements should be explained in more detail, along with the justification of actions taken (should be describing more of *why*, rather than *what* was done).

- Could use a HashMap with a HashSet of Events as the values, rather than an ArrayList of Events. The reason an ArrayList is used is currently because in original design, order was taken into consideration. In new design this is not the case.

Reflection

I do feel that given the requirements from the specification, this solution is unnecessarily detailed. However, beyond just correctness my goal was to use this opportunity to demonstrate good software engineering practice.

In development, I chose a horizonal slicing approach, in which I built each section layer by layer. I feel that this allowed for a more robust application, buy the process ended up being slower and a lot more work was required in planning the design of the application before programming.

I took a very imperative approach to designing and constructing this solution for the sake of the test, but I'm really interested in trying a simpler, more concise, and more functional approach. I am very eager to learn alternative methodologies, and I'm also looking to improve my current skills. I feel that there are many obvious errors in my work right now and numerous improvements that could be made to this program.

My two biggest compromises with this project are:

1. I didn't incorporate a database. This was because I felt that this would take a bit too much time, and a lot of my time was taken on testing the robustness of the service layer.
2. I didn't incorporate custom error messages and custom throwable errors, which was in my original design, due to time limitations.

Overall, I've thoroughly enjoyed working on this task, and I hope the pseudo client is satisfied!