Design Decisions Report

Final Project

CS321

Noé Poulain

Overview

This program is a simulation of an airport. It is simulated by creating a variety of Planes and Passengers for our airport. These two objects will be our Units and will therefore, each run on a different Thread.

The Passengers buy one Ticket when they perform their first action. Once they own a Ticket, valid for one flight, they will not be able to buy another one. There are three different options for seats, to which can be added a number of extras.

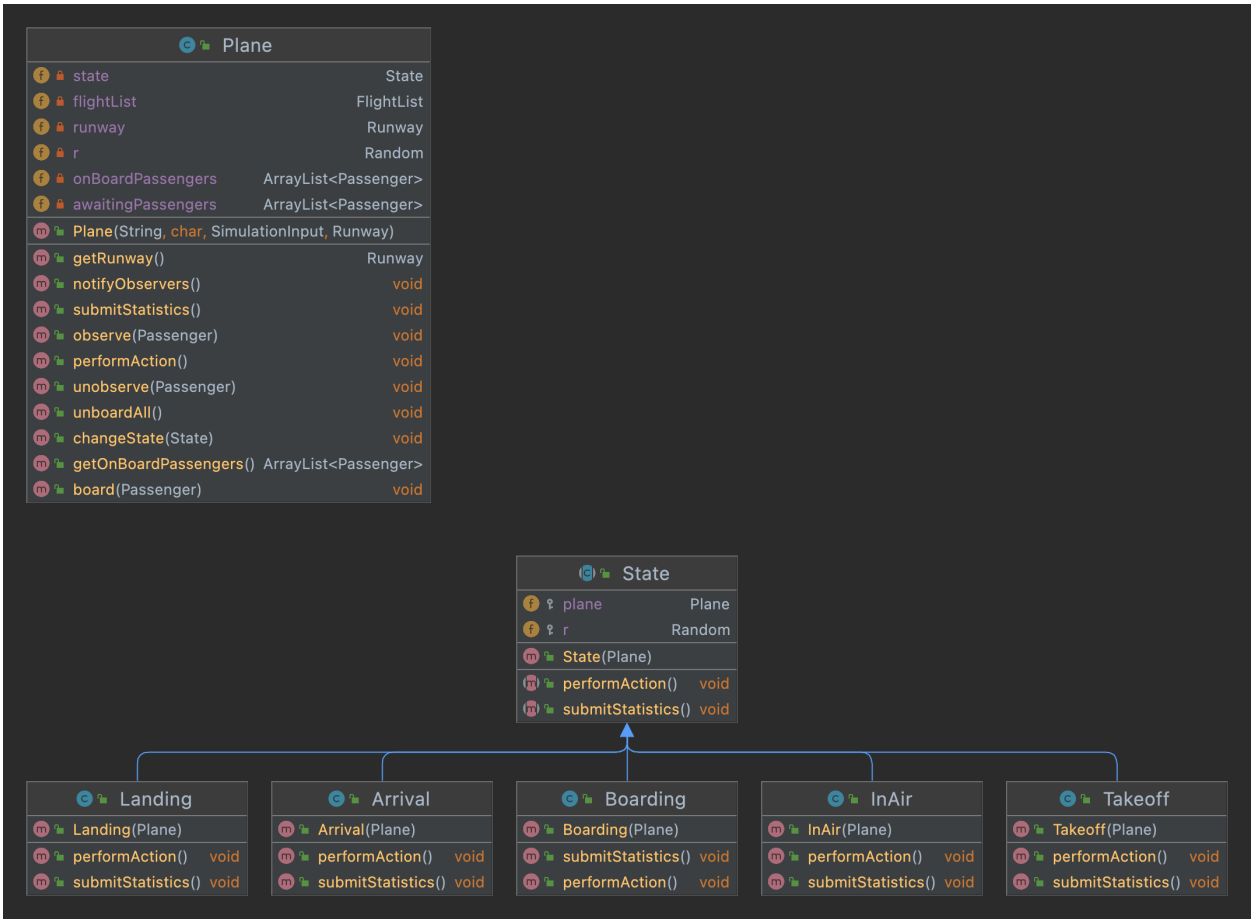
The Passengers will track the Plane corresponding to their Ticket in order to know when the Plane is available for boarding. When it is the case, the Passengers will get into the Plane and get off once the Plane arrives at its destination. The Planes will go through a cycle of flying from our airport to a destination and then coming back.

The simulation does not simulate arrival airports, therefore the Arrival state represents the arrival at an imaginary airport and Landing state the landing at the origin airport and therefore the end of a cycle.

Design Patterns

This program is implemented using five different design patterns (one creational, one structural, two behavioural and one concurrency) in order to facilitate modifications and to prevent bugs from the usage of threads.

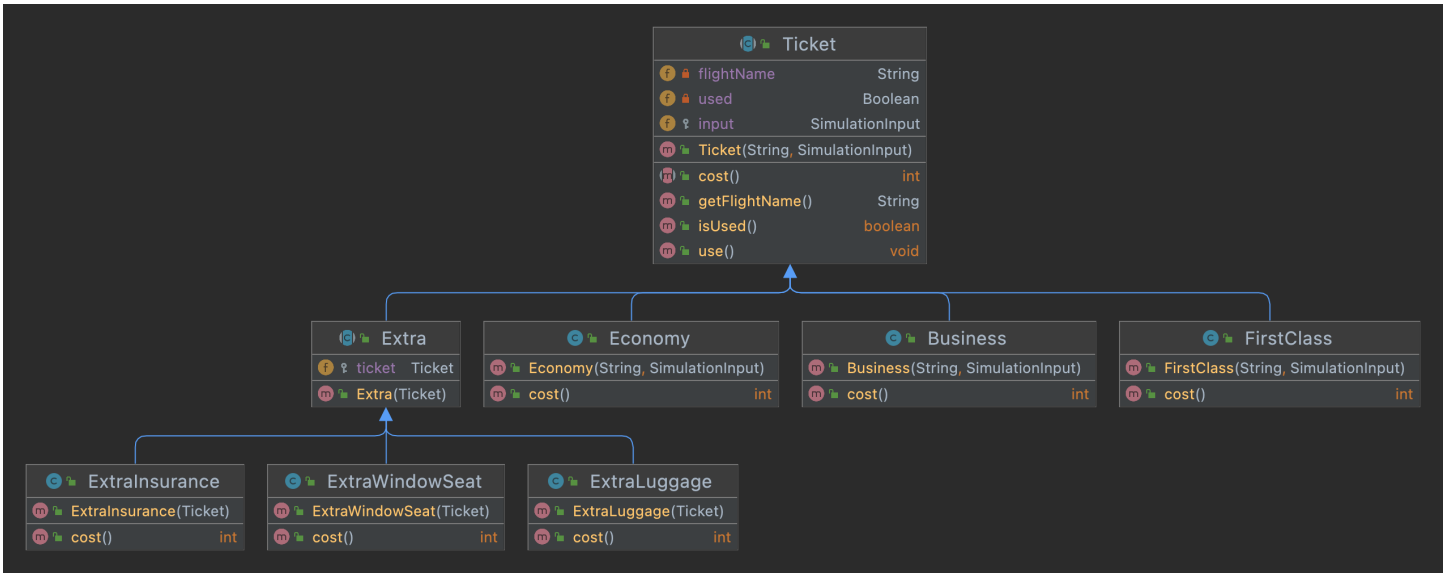
The first design pattern implemented in the simulation was the State design pattern. It is used by the Plane object. It allows the Plane to have different performAction() method depending on the current State of the Plane. For example if the Plane is in Boarding state, it will notify the passengers that it is ready for boarding. One major advantage of this design pattern is that it allows us to easily modify the behaviour of the Plane simply by modifying one or two lines of code to insert the new State or removing an old State.
The structure of this design pattern is showed in the UML diagram below. All the states inherit from the abstract class State so that they all implement performAction() and submitStatistics().



UML diagram for the State design pattern

The second design pattern used is the Observer pattern. It allows us to create a link between a plane and the passengers that have a ticket for the said plane. This pattern is a huge part of the simulation as we can have many objects (passengers) that depend on one object (plane) to do something, in this case boarding and leaving the plane. The pattern uses an Observer interface so that we can easily add different types of Observers to the plane (crew, baggage loaders, etc…).

The third design pattern is the Decorator pattern. It is used in the implementation of Ticket objects. It allows us to combine multiple options for the tickets without having a class for every combination possible. Therefore, by using the decorator pattern we respect the DRY principle as much as possible.
This pattern also allows the passengers to buy different ticket depending on one of their parameters, the amount of money they own. Hence, we can simulate different types of seats and gather global data for the money owned and spent by passengers. The pattern is structured as shown in the UML diagram that follows.



UML diagram for the Decorator design pattern

The last two patterns are tightly linked together. We use the Singleton design pattern to store a list of all the flights available. Therefore we obtain and use the list of all the flights in any part of the program.

While performing tests, some bugs appeared due to the multi-threaded nature of the environment. Hence the usage of a Double-checked Locking design pattern. Without the double-checked locking, there was a risk of creating two instances of FlightList despite it being a Singleton and therefore break the whole purpose of the FlightList class.

Implementation Decisions

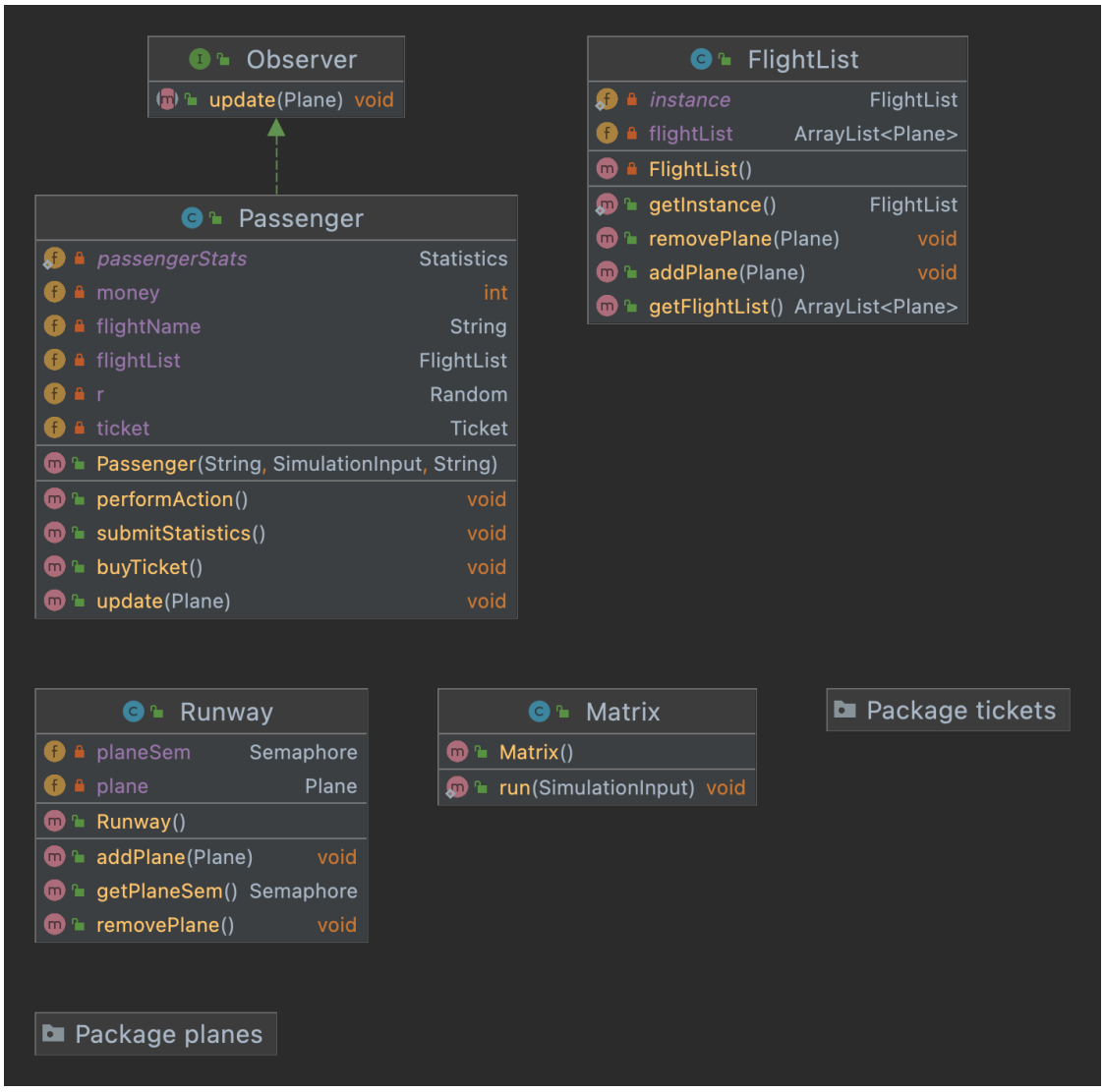
The main data structure used for the simulation is Java's built-in ArrayList. As the simulation runs, passengers have to be stored, added and removed. Consequently, it is required to use a data structure that allows for changes in size. Another data structure that could have been used for this program would be LinkedLists since our operations consists of adding and removing. However we often use the get() method which is more efficient using ArrayList.

The program tries to respect the DRY principle as much as possible. To do that we use three abstract classes, Ticket, Extra, and State. Most classes extend one of them.

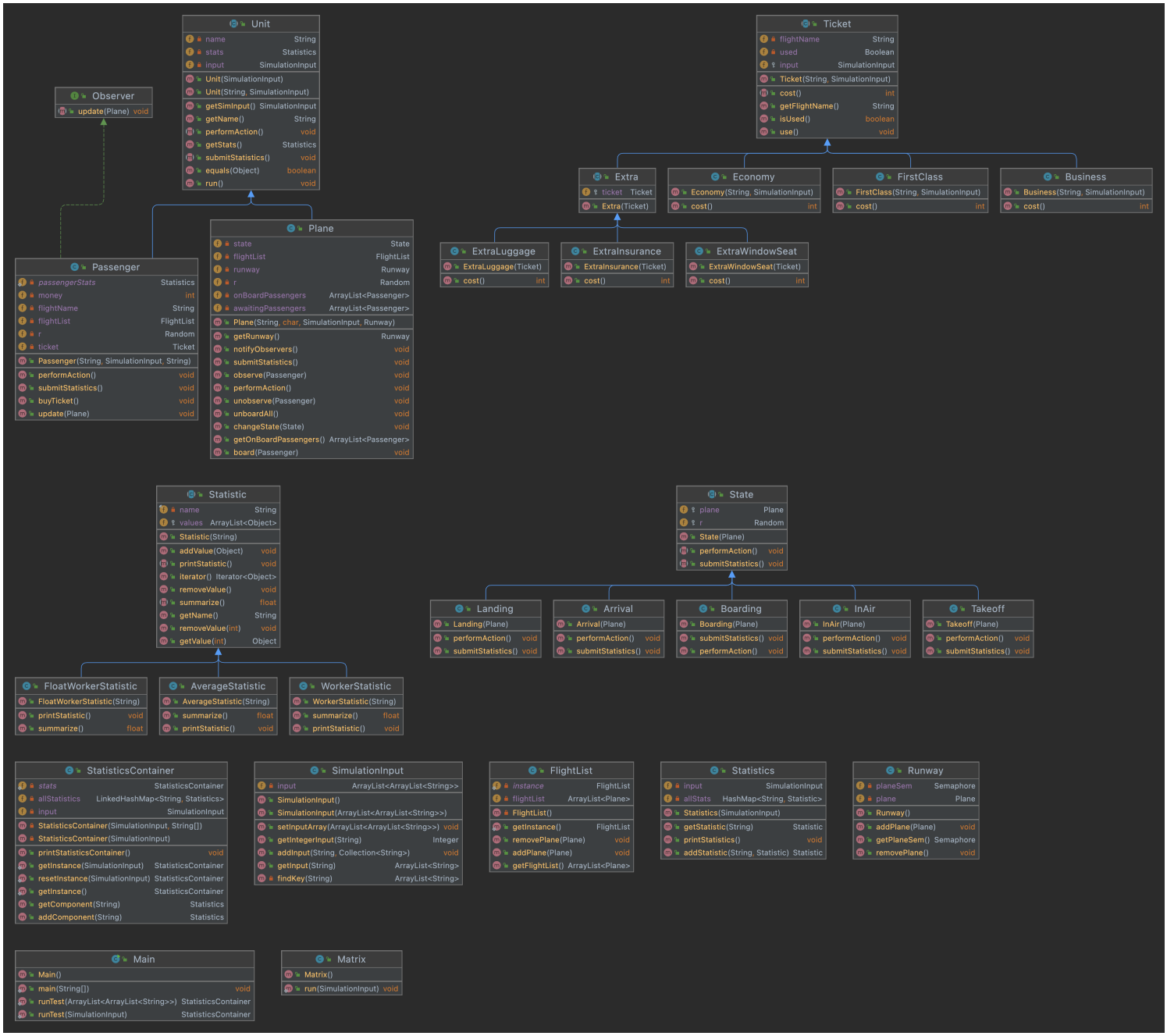
In terms of synchronization, the program uses a semaphore to control the Runway the planes need to takeoff. This semaphore does not allow more than one plane on the runway at the same time. Thus, when a plane enters Takeoff state or Landing state it acquires all the permits and releases them upon before going to the next state.

In addition to the Double-checked lock pattern used in FlightList, we the methods are synchronized in order to avoid concurrent modification of the ArrayList and printing issues.

UML Diagram



UML diagram of the simulation package



UML diagram of the whole program