Design document

CSC207 project - Group 0291

# Specification

Purpose: To create a program that allows a customer (user) to reserve tickets for an airplane ride. Each user will log-in to the system using their credentials in the program shell: Username + Password. If they are a new user, they will need to sign up for a new account - there will be a user tracker that should keep track of usernames and passwords for different users as well as other pertinent info such as address, DOB, payment methods, previous login dates and previous flights. A user will also be able to purchase an exclusive membership if they wish.

There will also be a Flight tracker available to give the customer a list of available flights based on the customer’s search. Each flight will have an ID (flight number), date (including the local time of departure and arrival), fare class, and class of travel. Each flight will also have an expandable seat map that shows the occupied/vacant seats on the flight (represented by x’s and o’s). After a user has bought a (valid) ticket, their seat will no longer be available in the seat map (vacant → occupied). The seat will be assigned to a user, unless the user cancels their flight.

The user will be presented with a menu upon login with options that allow the user to buy a ticket, cancel a ticket, or search available flights based on their flight number. They can also view their account details or set a payment method for their account. There will also be an option to log out, when the program will terminate.

Users can have exclusive memberships which will grant them access to special flights. These flights will not appear to a non-member. A user can be an administrator, granting them access to a different menu than the user’s. This menu will have options to add or remove a flight from the available list of flights. An administrator should also be able to see the list of users in a flight, as well as the list of loyalty members. They should also be able to manually cancel a user’s flight in case of a security threat.

When the user selects an available flight & seat that they wish to purchase, a new ticket will be made. The ticket will keep track of the flight details and its user’s name. The ticket will also have a price that differs based on details of the flight/seat, or the user’s loyalty for any added bonuses - the tickets have two types: normal and exclusive depending on the flight of the user. The ticket will expire after the flight has occurred - this would mean that the program will keep track of the current time and date.

# Clean Architecture

# Based on the dependency rule, source code should always point inwards, i.e.: from less stable to more stable classes/policies. At the core of our program lies the innermost (domain) layer of code, which contains entities such as User/Flight/Ticket (the entities are essentially the basis of our program/higher level dependencies). The layer directly outside of the domain layer contains the application business rules which house the Use Cases, such as BuyTicket or SeeAccountDetails which manipulate entities. Keeping the dependency rule in mind, the entities have no knowledge of or dependency on the Use Cases. The source code always points inwards, as does the level of abstraction.

# SOLID principles

The Single Responsibility principle was applied through separating large classes that had multiple responsibilities into smaller, single-reason classes (e.g., git revision number bb17362f separating admin menu’s methods into separate classes). The Open/Closed principle was applied by having the instance variables in Entity classes be private (but with getters/setters), as well as having controllers for each menu that allow for easier addition of other options. The Liskov Substitution principle was applied by having classes be abstract, so that the “child” class would instead extend the parent class by implementing the abstract methods (e.g., User → Admin/Customer classes and Database → Flight/UserConnector classes).

# Design patterns

# The main design pattern that was implemented was the iterator pattern, implemented in git revision 50680faf. It was implemented for the UserTracker class to improve the retrieval of the User database present in UserTracker (as it would give easier access to each individual User as opposed to the previous design). Other design patterns such as the Facade pattern were considered for the implementation of Controller classes, however ultimately the pattern was merged with clean architecture by having the controller and presenter both be a sort of "facade", which would dedicate work to Use Case classes and respect the principles of clean architecture and SOLID.

# Phase 2 refactoring + code smells

# Universal Design

A description about how your program does or COULD follow the 7 principles of universal design (from the ethics lecture)

# Personal Contribution (for phase 2)

* Billy:
* Daniel:
* Karl:
* Mani:
* Taymoor: