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 (occupied seats are represented by “0”, while unoccupied seats are labeled by “X”). 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. Customers also can accumulate loyalty points from purchasing flights.

A user can be an administrator, granting them access to a different menu than the users. 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

# From phase 1: 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.

Additionally for phase 2: A new set of functionalities for payment methods was added that touched on all layers of Clean Architecture and not violate the dependency principle (git revisions 715cdbbd for entity, 12bb0dc4 for the use case, 3de04b44 for the presenter/controller, 52348d13 for gateway). The presenter class that accesses the main payment method entity was designed with dependency injection – it only depends on the object that the payment manager use case gives back, not the payment method itself. The use case also does not depend on the presenter menu but rather the payment method entity only. The program was also extended by enabling a Membership option for customers. To do so, a Membership menu presenter and controller (revision 27816c71) were implemented, which again, accessed the Customer entity through the object created by the GetUser use case, while the entity has no knowledge of the use case, and the use case has no knowledge of the menu. This also reinforces how the entities are the most abstract layer of the system, and as you move to the outer layers, the implementation becomes significantly more concrete.

# 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., Database → Flight/UserConnector classes).

Additionally for phase 2: The Liskov Substitution principle was extended and refined by having more functionality in the Customer Class than in the User class – Loyalty points (Git rev. 7ac21a35) and Payment methods are only available to Customers, which allows for Customer to be substituted for Users in Use Cases when needed. The Open/Closed principle was applied by making several “helper/director” methods in controllers/presenters be private (Git rev. 98097f95) which will allow for addition of new options to each without disrupting the previous options. The dependency inversion principle was applied in the Gateway classes by having the main Application rely more on the abstract “Database Connector” save/load methods, which would it dependent on the abstraction rather than the concrete Flight/User/Payment/Ticket connectors.

# Design patterns

# From phase 1: 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.

Additionally for phase 2: After Revaluating our code in this phase, we changed the design pattern of both UserTracker (git rev. d144da) and FlightTracker (git rev. 7f01252e) to now be the Singleton design pattern (Reasons for change in Refactor section). The Singleton Design Pattern instantiates only a single instance of each Entity class (hashmaps of both entities), ensuring all objects accesses the same single instance. Memory is not wasted creating new objects of the instance when there is already one present.

# Phase 2 refactoring + code smells

Our Phase 2 edition of our program is far more polished and complete with useful features. We added a flight generator (git rev. 213e7e5b) to populate the Flight Tracker when the program runs to give the Users more options. We also added data specifically for Customer subclass (not user), as there was not much of a difference between customer/admin before (git. rev 7ac21a35)

Furthermore, we greatly increased the number of tests; for phase 2, we added enough tests to encapsulate all UseCase methods and more (git rev. 320f891a, 7d24d01c, d605ebc4). This greatly boosted our confidence in the program’s performance and forced us to fix previous code smells.

Our group received feedback that we had too many static variables that change the state of the program within the UseCase layer. While a more “java-specific” code smell, we fixed this code smell through the implementation of the Singleton Design pattern. FlightTracker’s flightmap variable (where flights are stored) was formerly a public static final variable that was mutable (mutable map), and not a constant. By refactoring into the Singleton Design pattern, flightmap is now a constant, with the only static method of the UseCase being getInstance().

We made sure to fully follow camelCase conventions (git rev. 42affb53) per the feedback as well.

# Universal Design

We followed Principle 3(Simple and Intuitive Use) within the design of our menus by grouping similar information together; e.g the User Menu allows Users to view/change any information related to their personal information (viewing/changing birthdates, payment methods, etc.). Furthermore, all the User prompts and messages are basic/simple to understand, even with a limited English background. Throughout the use of the program, the User will be given simple feedback with task completion (i.e. “Account has been added!” after registering, “Your Payment has been saved” after adding a payment method). There is also feedback if the User enters an invalid input (e.g “This Seat is already Occupied. Please Try Again” if the User wants to purchase an occupied seat). Feedback helps with User intuition: the User knows immediately if their inputs are valid, and thus allows better understanding of the program and removes complexity.

We also followed Principle 4 (Perceptible Information) throughout the program. Within the flightMenu, essential information about the flights (flight number, destination) are uniquely presented from the rest of the console to maximize visibility of the information. By presenting flight information this way, it is easier for the User to choose a flight number and purchase a ticket, instead of the flight number being cluttered with other less important information.

The program also follows Principle 5(Tolerance for Error). As mentioned before, there is feedback given when a User performs an action. To prevent User from accidently/purposely crashing and disrupting the program with invalid input, we implemented regex expression matchers within the console class. This “Fail Safe” feature prevents the program from progressing (and potentially crashing) unless the User enters a valid input that matches the pattern. This can be seen when the User registers, where the User must enter a valid birth month that must be in the format MM, or an email that contains @).

One way we could have improved upon our program to accommodate Universal design is to have the program run automatically (perhaps like an app), instead of the User having to navigate to “MainApplication” in the SRC folder to run. This would better accommodate Users with little experience with java/IntelliJ.

# Personal Contribution (for phase 2)

* Billy: Graphic UI with login and registeration features, and main page for customer and admin users with other feature pages (developed from text UI version) for future developments. Tests for tickets entity and use cases. Fixed some errors (mainly Ticket ID, time-date representations) for entity class.
* Daniel: GenerateFlights, CancelAllFlights, ClearAllUsers. Repackaged Tests into appropriate test folders and wrote tests for nearly all methods in all UseCases.
* Karl:
* Mani: Update of UML diagram, design notes on Clean architecture/SOLID, implementation of Payment method (entity + use case + presenter + gateway), Database repackaging, Singleton Design pattern implementation for FlightTracker, User/Customer functionality refactoring
* Taymoor: Implemented the Membership features (MembershipMenu, MembershipMenuC, LoyaltyPoints) while refactoring related classes. Adjusted the remaining menus to streamline prompts and test for errors/lack of exit conditions. Design Document extension for Clean Architecture.