PHASE 1: DOCUMENTATION

FLIGHT PLANNER

15/11/2021

Prepared For :

CSC207F

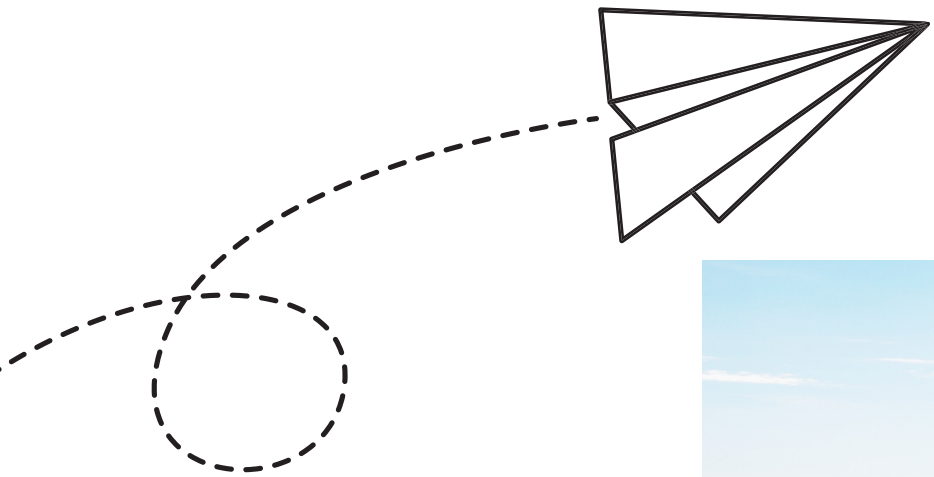
Prepared By :

Team KAMAMANS

UPDATED SPECIFICATIONS

PART ONE OF TWO

Create a web-based interface that lets users join a platform named Flight Planner. Users of Flight Planner are allowed to join at specific tiers that offer different benefits. **Unlike Phase 0, there is the option of upgrading and downgrading accounts without loss of information. This is achieved through the state design pattern of behavioural design patterns.** Each user has an ID (username), password, email, and phone number. Users should have access to their information and the ability to edit their own attributes. Running the platform starts the web-based interface that allows the user to interact with a login page. While running, the interface starts by letting the user log in or sign-up verifying credentials. Credentials are case sensitive to increase security. **Unlike the implementation through the terminal, we have created website pages using javascript and html to provide a layout for the GUI. The GUI will be completely implemented by Phase 2.**

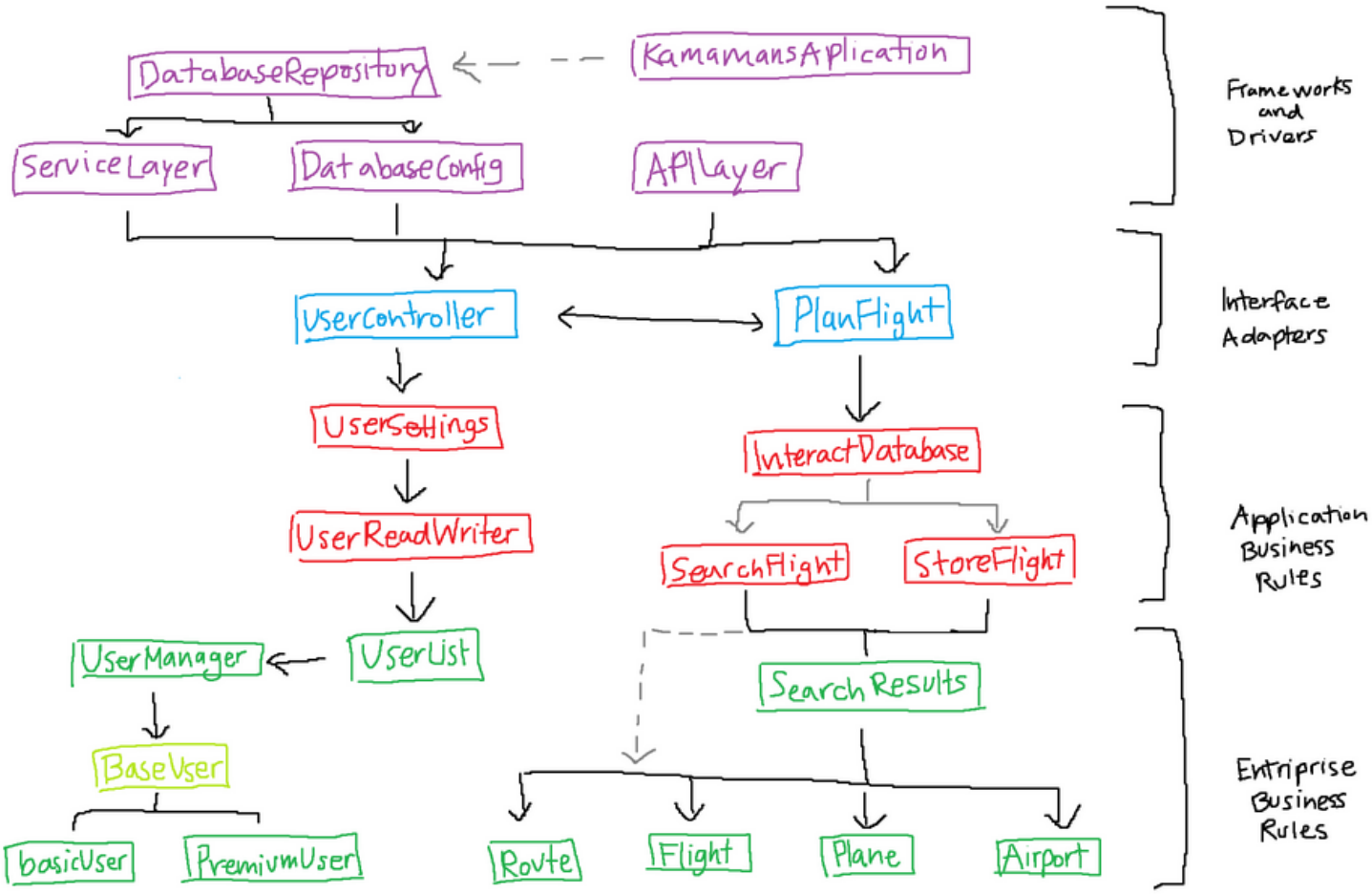


UPDATED SPECIFICATIONS

PART TWO OF TWO

Updated from phase 0, there is a connected database for planes, routes and airports. This ties in to while logged in, the users can search for and select a flight using data from an external API. Flights contain a date, plane entity, price, duration, departure location, and destination location. Plane entities contain a brand name and capacity for distinct types of passengers. Routes should combine multiple flights so that a user can get from their departure to their destination even using multiple flights. The platform will source its flight data from a current API called aviation stack and will use real-life, real-time flights. **The platform will save selected flights per users' choices to a centralized database (currently a csv), to allow for user's flight history to accumulate. The state of the user will be saved after logging out and closing the web-based interface.**

UML


DatabaseRepository
KamamansAplication
ServiceLayer
DatabaseConfig
APILayer
UserController
PlanFlight
UserSettings
InteractDatabase
UserReadWriter
SearchFlight
StoreFlight
UserManager
UserList
SearchResults
BaseUser
basicUser
PremiumUser
Route
Flight
Plane
Airport
Frameworks and Drivers
Interface Adapters
Application Business Rules
Entriprise Business Rules

OUR DESIGN DECISIONS

Major Design Decision

We had a few major design decisions our group had to make, and some will be expanded on in later sections.

- LOGGING IN/SIGNING UP
- SEARCHING/STORAGE
- INTERACTING WITH DATABASE

The first major design decision was how we handle both the logging in and signup up of users. We wanted to make sure that the user classes were adhering to a design structure that adhered to the SOLID design principles. From our frontend, UserController initiates the process and delegates tasks to userSettings. UserSettings then delegates more tasks such as serializing data and using the UserManager class. At this point, users are organized by individual UserManager's that store any relevant information and organize the structure of our BaseUser's, BasicUser's, and PremiumUser's.

The second major design decision was handling the searching and storage of flights. Similar to UserController, we implemented a class, PlanFlight, that controls how the different classes related to a complex Route behave. Here, the PlanFlight class keeps track of the current user and handles all their requests by using the different entity classes. Flight data is stored in the database and users can access this data by specifying the routes they want to take.

The last major design decision was how we would store, fetch, and save the relevant data to our database. There are two important parts to this. The first is when users create an account, they should be able to instantly access their account through the login page. So, when users create their account, their account is instantly serialized after being added to a list of all existing users. The next part is how users can both store their flight data and access flights from external data. Here, users get access to external data when they search for flights and on the SelectFlight page, then can then add a flight to their flightHistory to be viewed or referenced later. Keeping as much of this data retrieval and manipulation sourced in the backend was key to adhering to Clean Architecture.

CLEAN ARCHITETURE

Our project adheres to Clean Architecture as seen by our UML diagram. Our entities are objects that are used within our Flight Planner System to solidify what can be manipulated, recorded and changed. Our use cases are the only classes that manipulate our entities (with the exception of a not seen relationship between a database recording. However, this goes through the Interface Adapter which although it isn't recommended it is allowed. This is the closest we have to violating the dependency rule. We will now go over a walkthrough to show our updated clean architecture:

To enter the application, a user must first log in with the User Controller, which goes through the UserSettings and UserReadWriter to pull up existing users. In the UserList we store user's credentials and uses InteractDatabase to verify that the user's information is valid. If the user exists, then they have the option to book a new flight. UserController interacts with PlanFlight to begin inputting the user's search specifications. These search specifications are then used by SearchFlights to find any flights that meet the requirements. Through the backend we take the given information received by SearchFlight, and we search through a sorted list of all current flights taken from the API, using the APILayer, and return those that meet the specifications.

CLEAN ARCHITETURE

Once the user finds an appropriate flight, the selected flight is given to StoreFlight, which contains all the information related to the airport, plane, flight, and route. EditUser then takes this information and uses InteractDatabase to store it. Through the backend we take the given the information received by StoreFlight, we store the flight information to the associated user's account.

SOLID

DESIGN PRINCIPLES

One of the design principles we use is the Single Responsibility Principle. In our code we have the class UserManager which is responsible for common methods and attributes across all users. The SRP states that each class should have only one responsibility and therefore we had no way to store, add, remove or access users without violating the Single Responsibility Principle. Therefore, we ended up creating UserList which is a short concise class that allows for mapping of Users for easy access. Although this design is quite simple it achieves our goal for having concise classes/objects that are responsible for a single task only.

SRP

LSP

SOLID

DESIGN PRINCIPLES

Another design principle we originally used was the Liskov Substitution Principle. This will be discussed more later in our Design Pattern Summary. In essence the LSP states that if a program module uses a Base class (which is what we ended up using to differentiate between Basic/Premium Users) then it could potentially be replaced with a derived class without affecting the functionally of the program module. However, we violated this to use the State Design Pattern as both premium user and basic user are implementations of Base User. The SOLID principle used here is just an extention of the Open Close Principle since we are making sure the newly derived classes (users) are extending the base class (the simplest Base User) without changing its behaviour.

PACKAGING STRATEGIES

We considered two packaging strategies: Layer and Inside/Outside. The Inside/Outside packaging strategy was a serious consideration for us, as for phase 1, we split our team into two sub-teams: one sub-team tasked with focusing on backend, and another sub-team tasked with focusing on frontend. Since the Inside/Outside packaging strategy packages based on which classes are outward facing (frontend) and inward facing(backend), it seemed that this strategy would nicely reflect the way we chose to organize our teamwork. In addition, it would separate frontend classes from the backend ones, meaning no matter which sub team a member is on, everything would be easy to find and all files in the respective package would be relevant.

We landed on using the Layer packaging strategy, as we found it to be the most intuitive for our program so far. We also agreed that it helped us adhere to clean architecture rules and make the process easier to grasp, as each package can be found as a layer in the clean architecture diagram where the layers are visualized as concentric circles (Diagram can be found in Week 4, CleanArchitecture.pdf, slide 10).

PACKAGING STRATEGIES

Following clean architecture rules was our main priority for phase 1. Therefore, in having the constant reminder and visual cue, we believed it would lessen the chance of accidental violations of the clean architecture rules. In addition, we also completed phase 0 with the Layer packaging strategy and we found it efficient to continue with this strategy, as everyone was comfortable with how the packages were organized and what each package contained. Since interfaces don't necessarily belong in any packages, we decided as a team where to put them, based on what made the most sense to us and where we would expect to find them.

Although we decided not to use the Inside/Outside, the team agreed that as our program grows larger and more complex, we will re-visit the possibility of switching from Layer to the Inside/Outside strategy.

DESIGN PATTERN SUMMARY

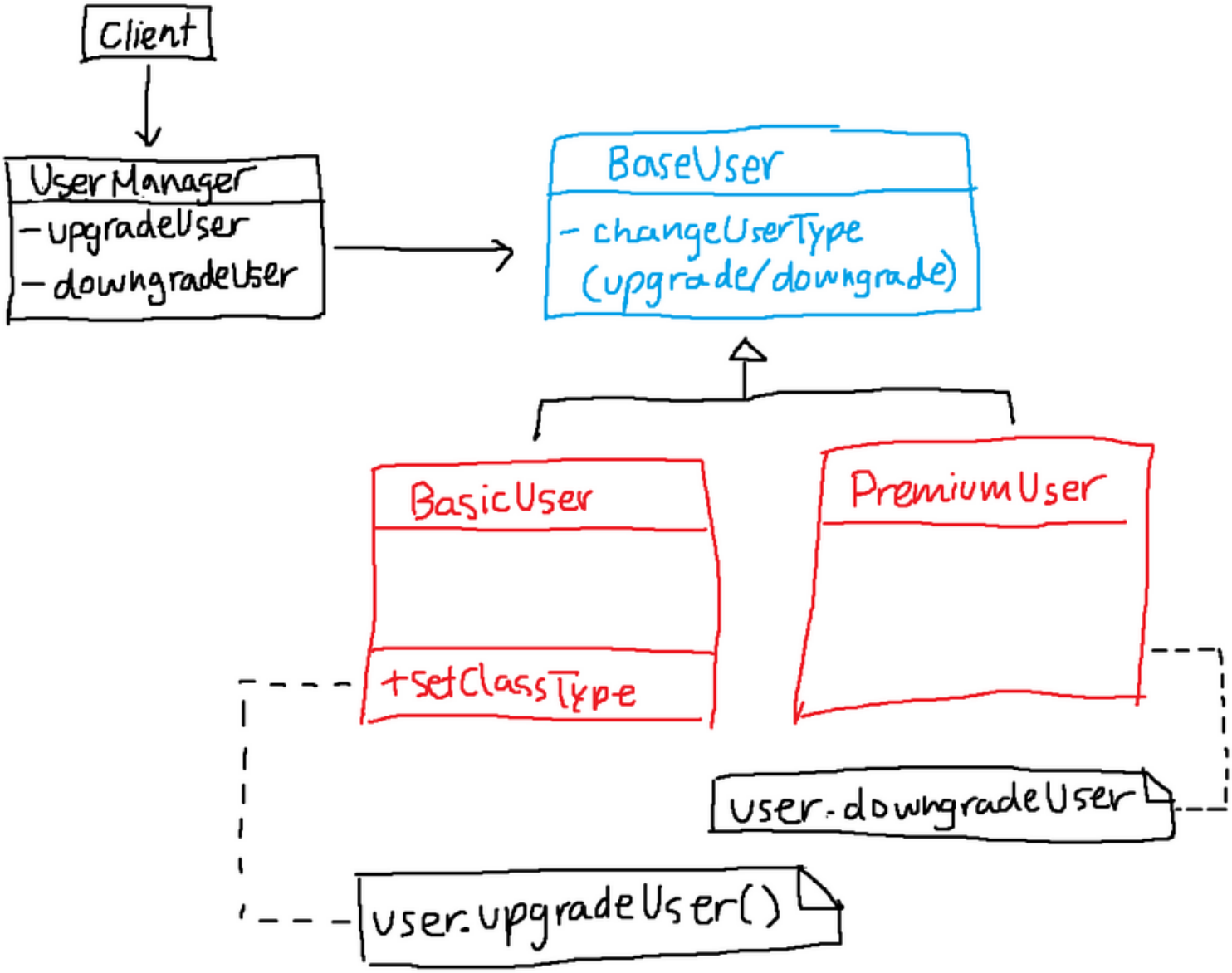
The most prominent design patter we have implemented is a structural design pattern called State. The state design pattern is meant to allow an object change its behaviour in a way that appears to switch classes. In essence, the object would be able to be characterized real-time and accomplish different things depending on its current state.

Currently our group has having an issue on how to control having a basicUser account and premiumUser account. We originally had two classes but ran into the issue that when a user upgrades and/or downgrades extra internal states must be created and/or are lost. Currently we have a Boolean inside our basicUser class that keeps track of whether or not an account is premium, and we manually check to allow for that specific user to access premium methods/settings. We are looking for a simpler method to encapsulate a user's information (when premium) so when they downgrade and later upgrade their settings/methods/bookmarks/capabilities are all restored to the internal state it had previous.

DESIGN PATTERN SUMMARY

Currently we have our User's encapsulated in a "wrapper" class which is the User Manager. This means that it's hierarchy mirrors everything with the exception of the single extra parameter, in this case it's whether it's a basic user or not. This means we do not lose any information, simply lose access to the functions that would allow us to access/change and therefore it will not allow any behavioural changes other than the ones we specify.

We will look into more design patterns for significant code sections when we begin incorporating the GUI more heavily (chain of responsibility for controlling the way of passing requests between a chain of objects) and when we implement more complicated search methods (template method design pattern to defer very exact steps of algorithms for searching).



Caption.

The UserManager is the Context. The BaseUser is the overall wrapper for the state. The BasicUser and PremiumUser are states that the BaseUser can be in. This shows how the structure of our code almost exactly mimics state design pattern.

GITHUB FEATURES

We utilized multiple github features throughout phase 1 of creating this project. Some of them include creating new branches (for each feature), pushing, creating pull requests, creating comments and reviews under pull requests, and merging. In addition, when we grouped up to create different features, we used the checkout feature to allow some of us to work on the same branch without merging to the main branch beforehand.

We had a few issues with merging, specifically some IDE-related files were pushed; this was solved by either choosing to overwrite or adding those files to the gitignore file. The other merge requests had a few slight issues due to code conflicts but those were solved quickly.

Code Style and Documentation

Currently we have a few warnings when the code is opened in IntelliJ. All of them however are unused methods, parameters, classes and/or commented out lines of code which will be used and fully incorporated by phase 2.
Overall, our code uses descriptive class names and method names that extensive commenting to explain what each thing accomplishes is not needed. Each class has a javadoc comment at the top summarizing its role and for methods that are slightly more complicated (in logistic understanding and not coding) they also have comments explaining how they accomplish what they do (e.g. downgradeUserType in PremiumUser).

TESTING

The testing classes that we created are for the main entities (Airport, Flight, Plane, Route, SearchResults, and User). These are all simply checking that the methods that are the basis of our program work and return the expected values. We also created the outline of the ApplicationTest (currently being made so is currently testing nothing).

We also tested the backend development as it was being created through InteractDatabase. It is currently commented out code (one of our causes for IntelliJ complaints) and it achieves testing serializing and object (post method) and retrieving it from its storage location (files). This was for ease of the backend team since we did not want to cause any unnecessary merge errors with entities and application tests (these were currently being developed and updated at the same time). Our goal is to collect these backend tests and create a respective test file in the test folder.

Overall, the majority of our components are fully tested. We do see testing with Application Business Rules slightly more complicated than normal due to the changing premises of users and having to make sure method changes do not change how they interact with the use cases.

To be incorporated in Phase 2 design documentation are code organization analysis and functionality demos.

PROGRESS REPORT

Majda: Part of the backend team. Implemented and tested the Plane Database. Worked with Salwa to create presentations. Reviewed other pull requests relevant to the backend. Helped with documentation and updated the CRC cards. Plans to keep the CRC cards up to date, keep reviewing pull requests relevant to the backend, and continue working on the backend, such as algorithms and helping with implementing the program to fetch data from the API of our choice.

Salwa: Part of the jack of all trades teams (just kidding not a real team). Created the basis of the frontend websites including starting the LoginPage, ProfileMakingPage and the SearchFlight. Created the design documentation, UML diagram, design pattern diagram and updated the CRC cards. Plans to keep the CRC cards up to date, continue to review pull requests and help with creating theoretical edge testing cases. Will work more focused on front-end and expanding the pages available. Worked with Majda to create presentations.

Kevin: Part of the frontend team. Created the web interface and added transitions/JS functions. Helped implement Spring application and linked the JS functions to our java classes. Helped connect endpoints for getting and posting data. Created method to turn searchResults into a JSON parse-able string. Worked on correcting class compatibility issues. Implemented search method in InteractDatabase to retrieve airports by city names.

PROGRESS REPORT

Marian: Part of the backend team. Refactored the user classes from basic subclassing to using the State design pattern so that shared features and attributes for Basic and Premium users are preserved while allowing easy customization for extra features for Premium Users. Handled serialization of users using the Java Serializable class by creating UserList and UserReadWriter. Plans to flesh out user classes (e.g. login information validation, creating new premium features such as favourite airport, saved flights, auto logout timer, etc.) and help with algorithms if needed.

Nathan: Part of the frontend team and worked a lot on the backend endpoints. Initialized the local server and worked on making the endpoints take and respond with the right data. Also worked on connecting the front end to the back end by writing HTTP requests and helped fix errors to make sure everything runs smoothly when the controllers are being called from the endpoints.

Andrew: Part of the backend team. Implemented and tested the route Database as well as updated the class interactDatabase to be able to store data on route into specific bin files. Also created updated route class to work with the new system. Planning to further help with anything that is needed in the backend as well as reviewing pull requests on github.

PROGRESS REPORT

Albert: Part of backend team. Investigated possibility of using spring boot server and postgresql database. Implemented Airport databases with GET and POST functionality through serialization. Organized tasks to be completed for other backend team members. Worked closely with teammates to learn the database system and structure. Helped other members solve issues during development. Began working on Route searching algorithm.

Andrei: Incomplete