CSC207 Design Document (Phase 2)

- **Group ASOUL**

Members:

- TOM LI
- STEVEN LIU
- YANKE (VECTOR) MAO
- MIKE YUAN
- WILLIAM ZHANG
- HONGSHUO(TONY) ZHOU

Specification:

We designed a FoodTruck ordering program in which the user can create an account to interact with the program and place orders from a food truck through the program, as well as sell food items through their food truck.

The project essentially consists of two major parts: the backend and the frontend. The backend is the core of our program and includes all the **design patterns and functionalities** that one can expect from a food ordering system. Some examples include:

- In a **login system**, the user can register with an account, and log in with the username and password anytime after the account is created. The login system also checks for incorrect or non-existing credentials and returns error messages.
- In a **Rating system**, the buyer can rate an order anytime after the order is being marked as complete (which is executed by the seller of the order). Then the rating will be reflected in the rating of the food truck as an average of the total.
- A **market sorter**, the market which consists of food trucks can be displayed either by default (random), by rating, or by name.
- **Serialization**, all data of user account, food truck, and orders are saved to an external file, constructed upon program entering and saved upon program exiting

… or other functionaries such as being able to

- **manage & edit** food truck and its menu
- **change** user information (nickname, password, add money...)
- **review** order history lists

These are just some examples of the functionalities of our program. A particular note to our user account design is that an account can act as both buyer and seller provided

the user manually activates their food truck if they want to “become” a seller. Then, the user can freely edit their food truck.

Now proceeding to the front end of the program. Our front end also consists of two designs: the originally implemented fully-functional command-line interface and a partially implemented Android app that acts slightly differently. (All files of the Android app are contained in the branch “Android”)

- The command-line interface is our primary front-end display which is interacted with the command line in Intellij, in particular scenes, we have provided a **“help”** function that the user can call to see the list of commands available to use.
- The Android interface is our secondary front-end display and is interacted through an emulator. We have made the app almost fully functional with the exception of implementing dynamically viewing the market and viewing order history, and place order (so it includes everything the command line can do including serialization except the ones mentioned) .

Major Design Decisions:

- **CodeBase Simplification**
  The following changes are made to reduce code complexity and workload due to the time restriction on the size of the program.
  - **Dropped add-on entity fields**
    Some of the add-on entity fields were dropped for simplification, like the labels of the food.
  - **Reduced-order states from 4 to 2:**
    The order states used to have 4 states which represented “placed order”, “in progress”, “cancelled”, “completed”. Throughout the development process, 2 of them are dropped, and “in progress” as well as “completed” are left. This decision was made due to the same reason as above.
  - **Merged 2 user types into 1**
    The original specification allows for 2 user types, one represents the seller and the other one represents the customer. Each seller used to be able to hold multiple food trucks. Now each user starts with an empty idle food truck.

- **Code Structure Changes**
  - **Formalized Login Process and Enforced Access Control**

Controllers will not be able to modify existing entities without the proper access key. Such a key is generated each time the user logs in, and is discarded once the user logs out.

- **Use Case classes are made to be methods holders**
  Since the use case classes do not contain any instance fields, their methods are made static to allow other components to call them without instantiating use case instances.
- **Plugin Based Controller Packaging**
  The implementation of controllers are now separated from the controller package to allow other designs of the controller to be deployed

Clean Architecture:

The full UML Diagram can be accessed here.
Throughout the project, we've always implemented codes and created classes with the idea of clean architecture in mind, thus the structure of our program fully demonstrates clean architecture.

Firstly, all files can be categorized into the four layers of clean architecture, below is the list:

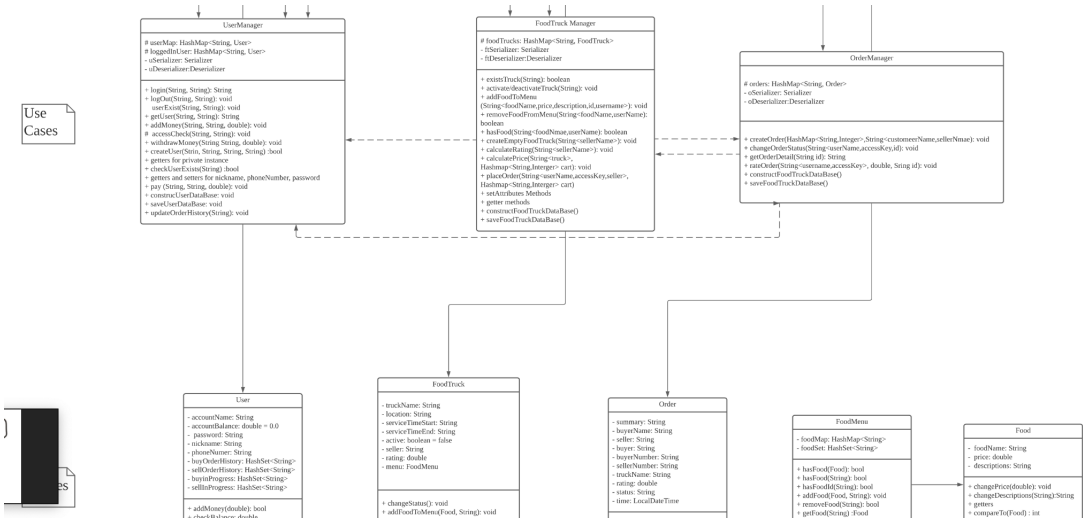
Entities: User, FoodTruck, Order, Food, FoodMenu

Use Cases: UserManager, FoodTruckManager, OrderManager

Interface Adapters: Scene (and its many subsequent subclasses of different scenes), Scene Booster. (And all the Android Activities are controllers & presenters)

Framework Drivers: CommandLine Interface, External database (data folder), the Android interface

1. The essence of clean architecture comes from the fact that "outer layers" can depend on "inner layers" but not vice versa. This is best demonstrated through our UML diagram that it follows a clear "top-down" structure: **arrows always point from the outer layers (which are located more towards the top) to the inner layers (located more towards the bottom).**
    - For example, from the UML diagram, only the managers have dependencies on the entities, no entities rely on any of the managers



2. Another way to show dependency is through the import statements from the java classes. It can be checked that **none of our files violates** the principle of clean architecture by importing classes that are either to the outer layer of itself. Furthermore, classes do not skip layers, meaning that interface adapters like controllers do not interact with entities directly.

SOLID:

Single responsibility principle:

Previously in Phase 1, our code has followed the principle of single responsibility. We have created several entity classes, which each of them stands for one single entity object (User, Order, FoodTruck etc.). We also implemented several different use-case (manager) classes each of which is responsible for creating or editing one single entity object. For both controllers and presenters (Scenes), they follow the SRP as well since each Scene class only represents one scene of the ordering process. Moving toward Phase 2, we have created different controllers & presenters (Activities), and UI using Android. These activity classes do the same functionality as Scenes except each activity class has an associated XML file that can display these Scenes on Android. Since each activity class still represents a single Scene, so it also follows the SRP.

Open/Close Principle:

Moving from Phase 1 to Phase 2, we have followed the Open/Closed Principle that each single responsibility class is open for extension and closed for modification. While

designing our Android Interface, we have never modified the existing methods since they are useful toward our Command-line Interface. Instead, to meet the need for new functionalities, we have extended our classes by creating new methods associated with the Android Interface

Liskov substitution principle:

As we have mentioned in Phase 1, all subclasses we have created in our code follow the LSP. For instance, we have created a parent class named “Scene”, the “Scene” class has some common functions such as switch Scene. All child classes of “Scene” (such as “OrderScene”) inherit these common functions and will not modify them. Instead, they have extended their class for more methods to meet their special behaviours.

Interface Segregation Principle:

Following the Interface Segregation Principle, the interfaces we have defined in our program are small. For example, we have only two necessary methods while writing our Observer/Observable interfaces. Thus, the classes that use these interfaces do not need to override some useless methods. Moreover, as an extension of this principle, while we are writing some parent classes, we only create methods that are necessary and share commons with all its children's classes.

Dependency Inversion Principle:

According to the Clean Architecture, we have divided our code into different layers including entities, use-cases, controllers, interfaces. The entity classes are the deepest-level classes that are created first by us. Use-cases are implemented after entity classes since they aim to manage the entity classes. Controllers are implemented at last since they aim to call manager methods only. Notice that our deeper-layer classes are never dependent on the outer-layer classes, and in our program, there is no necessary situation that we need to call outer-layer classes’ methods through an interface.

Packaging Strategies:

We packed our code mainly by layers and functionalities. We designed our packages based on the Clean Architecture, so you would find packages named like

- Entities
- Use_case
- controllers

Another group of packages serves a certain feature that could be used at any layer. For example

- Sorters
- Serializations
- Exceptions

We have also created packages for some design patterns, for example,

- Singleton pattern
- Observer pattern

We benefited a lot from this well-structured package. It helps us to find and locate a specific class or method during our development and debugging process.
It also helps us to separate work for our group members.

Design Patterns:

- **Observer:**
  - Observer and Observable are good tools helping us to implement two classes that interact consistently. That is, an update in one class will definitely affect the other. Below are some example usages.
  - Our Commandline interface implements Observer, and SceneBooter implements the Observable. After the booter, We implement this design pattern because the output String will update automatically.
- **Singleton:**
  - Our program only needs to instantiate exactly one instance from each class in the controller's package.
  - Controllers are used to connecting interfaces and use cases, which means there should be only one ‘Manager’ responsible for every connection.
  - Singleton pattern is a perfect method to reinforce this fact.
- **Template method:**
  - This design pattern is great for reinforcing the open/close principle.

    - For example, each sorter shares the same sorting steps but varies in detail. Other programmers can also easily implement a different kind of sort in the future.
- **Facade:**
    - SceneBooter is the facade that interacts with all the child classes of Scene classes.
    - This is a better encapsulation where clients of this code can only have to interact with one class.
    - It is easier to follow the single responsibility principle since changing one class will not affect others.
- **Simple Factory:**
    - This design pattern is implemented for getting the corresponded sorter from the user input.
    - We hide the construction of a new sorter from the user to make the code cleaner.
    - This perfectly aligns with the open/closed principle, which other programmers can also easily extend this feature.

Accessibility:

We considered a lot about accessibility in our project. When we did the design, we always followed the 7 Principles of Universal Design. As we implemented the project, we found that compared to the command line, Android App has better accessibility. This is one of the reasons why we choose to construct a second version of our project. Here are some, but not all, facts that support that our project has good accessibility.

Principle 1: Equitable Use

Our program is equivalent for all users. From the registering process to the ordering process, our program doesn't segregate any users, all users are treated as the same.

Principle 2: Flexibility in Use

We are going to implement a series of using options for users to choose from in the future. For example, we plan to add the option to switch the language of the program, switch between daily mode and night mode, change the font size and so on. These

functions will provide users with the flexibility to use the program in the way that they'd like to.

Principle 3: Simple and Intuitive Use

In our Android App, we designed a series of functions to eliminate unnecessary complexity in use. For example, when you click the Register button on the register page, if the registration successes, the app will automatically bring you back to the login page. Thus, you don't need to click the back button if you want to log in with your new account. Also, our program is consistent with users' expectations and intuition. In our Android App, we have the hint text in each input box and button that implies the function of these components. And in our command-line version, we provide a note of the most useful command in each scene, and we also have a "help" command to guide the user. What's more, our program always provides effective prompting and feedback during and after task completion. In our Android App, we designed several toasts to notice users what's going on, like the toast with the text "Successfully registered" after the user logon a new account. And in the command line version, there is also a text-feedback after the user input a command and the program completes the task.

Principle 4: Perceptible Information

Our program always provides adequate contrast between essential information and its surroundings. For example, in our command-line version, we left out a blank line between the most useful command note and the scene information, which makes it easy for users to catch the command information. And in our Android App design, there is adequate colour contrast between the background and words, thus making the text information very clear.

Principle 5: Tolerance for Error

When there may be an error in the usage, our program will provide users warning to avoid wrong options. For example, in the command-line version, when users enter an invalid command, our program will return an "invalid command" warning. And if the user enters a wrong old password while changing the password on our Android App, our program will pop up a toast with the text "The password is invalid or incorrect. Please try again." In the future, we want to develop more on Tolerance for Error for our Android

App. For example, we plan to add the serialization function to save data during the onDestroy() and onStop() life cycle of the Android App. This can effectively avoid data loss risks.

Principle 6: Low Physical Effort

In our Android App, we designed it to automatically jump to the next page after completing a few actions. For example, after the user successfully changed the account information, the App will automatically switch to the user information page, thus avoiding repetitive clicking.

Principle 7: Size and Space for Approach and Use

In our Android App, we set appropriate space between each button and message. And with the adequate colour contrast of our UI, users can get all information from our app from different aspects.

Target Users:

As the main purpose for our project, we want to create an app for all UofT students and staff to get information about the food-serving services at the in-campus Food Trucks. Thus, our main target users are UofT students and UofT staff who regularly come to the campus during the daytime. Besides, we also expect people who live around UofT to be part of our target users because buying food at UofT Food Truck can be a convenient choice for them.

Less likely audience:

Since these Food Trucks that our program tracks are located on the UofT campus, people who live far away from campus are less likely to use our program due to distance. Furthermore, since our program (both frontend interface) rely heavily on electronics, we expect that only the portion of the population that are familiar with electronic use will be using our program to order, which means statistically speaking, the older generation might not be so knee-deep in using our program, and rather chooses to order physically.

Progress report:

*Note that all the android codes are in the “android” branch rather than the main (default branch), thus GitHub doesn’t include contributions committed to that branch.

- **MEMBER CONTRIBUTIONS**
  - William:
    - Android app (Login, Register, Userinfo, ChangeBalance, Globalvariable)
    - UML diagram
    - Design Document
    - Rating system
  - Tom:
    - Android app
    - Design Document
    -
  - Tony:
    - Sorters implementation
    - Command Line interface
    - Design patterns
    - Design Document
    - Add java documentation
    - Participate in the design structure of the controller layer.
  - Vector:
    - Android app
    - UML diagram
    - Design Document
    - The test for the command-line program
  - Mike:
    - Place order/order rating (Command Line Interface)
    - Market (Command Line Interface)
    - Use cases refactoring solidified login process and user access control
    - Entity Clean up
    - Singleton, Observer implementation (Command Line Interface)
    - Design Document
    - Serialization
  - Steven:
    - Android app
    - Design Document

        - Android Refactoring
        - Design Document

- **OPEN QUESTIONS**
    - How to implement dynamic buttons for android apps? Dynamic button refers to creating new sets of buttons in a specified order every time when switched to the market layout.
    - How to put our Android app to an app store, like Google Play.

- **SIGNIFICANT PULL REQUEST**
    - William
        - Pull Request #114 This pull request was significant to the group with its addition of the global variable to the android project and solved our long-troubled problem of not being able to pass variables across activities.
        - Pull Request #105 This pull request was significant to me as it marked my better understanding of git. Although difficult to tell from the content of the pull request, prior to that, our group was having trouble always committing unnecessary local files. It was then I figured out the interaction between git cache, .gitignore, and files already added to VCS and solved this problem.
    - Tom
        - https://github.com/CSC207-UofT/course-project-asoul/pull/101
        - This pull request was very significant for the group because we figured out how to put android app codes to the Github repository. So everyone can finally start working on android development.
        - It is significant for me too, as it consists of my first activity.java file and XML file for the android app.
    - Tony
        - https://github.com/CSC207-UofT/course-project-asoul/pull/72
        - In this pull request, we finally get to the final version of our design of controller layers. There were lots of flags in the first version of the controller and although we later refactored some code smell, it did quite fit the clean architecture and SOLID. In this final version, I discussed a lot with Mike and we can up this version, which is clean and easy to read. I also implemented the observer and facade design pattern. I think this is my most important pull in the term

because I combined a lot of design knowledge learned in this class in this single pull request.

- Vector
  - https://github.com/CSC207-UofT/course-project-asoul/pull/12/files
  - In phase 0 and phase 1, I mainly focused on the extension of the function of our program and refactoring. I was responsible for part of the use case classes of our project. And this pull request is the draft of my use case classes.
  - https://github.com/CSC207-UofT/course-project-asoul/pull/108/files
  - In phase 2, I mainly focused on the construction of the Android App. I was working on developing Android App and implementing the function from the command-line version to the Android version. The pull request is the draft of the Android App.
- Mike
  - https://github.com/CSC207-UofT/course-project-asoul/pull/99
  - This pull request was very significant for the group, as it implemented a vast amount of functionalities for the command-line user interface and sets the foundation for other controller implementations.
- Steven
  - Steven Pull Request #123 (github.com)
  - This pull request was very significant for our group because it is an important milestone while implementing our Android Interface since it connects the user information, market and order altogether.

- **GOOD DESIGNS**
  - Sorting features
    - Allows users to sort the trucks by rating and by name once they are in the market view. This helps users to find the desired truck based on their own needs.
    - Used template design methods, which benefits us from implementing more kinds of the sorting algorithm in the future. It also obeyed the open/close principle.
  - Rating system
    - Users are allowed to rate any orders in their order history. This will be a fair evaluation of each food truck seller, but also a recommendation to other users.
  - Plugging Scenes
    - Our implementation of controllers is separated from the controller package, allowing other developers to modify the scene flow

behaviour by implementing their own version of controllers without affecting the existing code base.

- Access Control
  - Controllers are restricted from modifying underlying models without the corresponding access key (i.e User a cannot modify the information of User b), such key is randomly generated and returned each time the user logging in with the correct credentials.

- Use of Design patterns
  - We used lots of design patterns which made our code clean and organized. Moreover, it helped us to program more efficiently and less error-prone during development.

Use of GitHub Features

- Issues: we used issues while implementing the Android part of our project, we mainly used it as a way to distribute workloads to members and make notes for each individual's tasks as a record.
- Pull requests: This is one of the GitHub features we extensively used. We have opened well over a hundred pull requests since the beginning of the project. We mainly used pull requests to resolve conflicts, track changes, and make sure that codes are added or deleted in our desired way.
- Branches: This feature is mainly used to separate the work environment and aids the process of pull requests. We each have a branch that we normally code on and then merge to main we have ensured everything we will be pushing are bug-free. Same with the “android” branch, we each have a “username-android” branch, and the “android” is serving as the “main” branch of our android app.

Code Style and Documentation

- All warnings were resolved.
- We have documented all necessary methods as we’re implementing them.
- Our code is structured and packaged strictly following the clean architecture and functionalities, it should be easy for one to navigate between them.

Testing

- We have tested all the methods in classes in the entities layer and use cases layer. We choose to test these classes because they are the core of our project and if they are correct, all the classes in the lower layers will be easy to implement with the correct implementation of entities and use cases. Also,

due to our design of controllers and interface, it is rather hard to test them, so we choose to leave them out.

Refactoring

- We refactor our controller layers during phase 1 and phase 2.
- There were no design patterns and we added 5 design patterns during phase 2.
- We eliminate all the warnings and add more documentation.
- We delete all the unused methods and attributes.

Code Organization

- The classes are packaged following the clean architecture and are divided by functionalities. The controller classes are made abstract and public inside the controller package to allow implementations of other controllers to inherit them.

Functionality

- Login & Register user
- Change user information (nickname, balance, phone number…)
- Activate Truck & Change truck information (add food to menu…)
- View market & Sort food truck view by different methods
- Select food from food truck & Place order
- Review order histories (both buyer and seller)
- Rate Order
- Help function for command-line

Current Status and Future Plan:

- Current Status: By the end of Phase 2, we have created a **complete** command-line interface for our food truck system, that supports all the functionalities including creating users, editing user information, creating/editing food trucks, view markets, food ordering, order rating, order history, order/user database etc.
    - **Current Status with the Android app and its reasons (important):** we have created a runnable Android app with some of the functionalities mentioned above (namely all the login/register, user information, edit user info/truck info, and also serialization). Within the Android app, we have designed the layout page for every single scene. However, due to our lack of experience with creating android app, along with many unexpected gradle and setting problems, we were unable to fully implement all the features that our full program supports.

- Future Plan: At this point, the functionalities that our Android app supports are behind the functionalities that our command-line interface supports. In other words, our Android

app now is not completed since we have encountered some technical problems about Android that are beyond the scope of this course. Therefore, as a future plan, we are going to learn some relative concepts about Android app designing, and afterwhile, we will definitely move back to this project and finish up some functionalities that our Android interface is not currently supported.