Phase 0 Progress Report

- By Group ASOUL

Members:

- Tom Li
- Steven Liu
- Yanke Mao
- Mike Yuan
- William Zhang
- Hongshuo Zhou

Specification summary:

We plan to develop a food truck ordering app in which users can interact with the app to place orders for the food truck. The project will consist of two parts, a front-end and a back-end. The front-end will be where all the information is displayed to the user and where the user can interact with the app and send commands. The back-end is where all information is being processed and stored, including how orders are placed, user account info and more.

To use the app, a user will register and have the option to choose between being a **Seller** or a **Customer**.

A Seller will have to register a corresponding (or more if needed) **Food Truck** under their account, in which each Food Truck will have its individual description and **Menu** (of **Foods**). A Seller will maintain the Food Truck service through various features, such as updating the menu, keeping track of Food Truck's order history and more.

A Customer can place an **Order** on the app by choosing Food from a list of Food Truck, but before that, the Customer needs to add balance to the app. The Customer can also choose the pickup time of the Order, and rate the Order once it is completed, which later will be reflected onto the Food Truck's rating. A Customer is also entitled to their own order history which they can review at any time.

CRC Model summary:

The Basic structure of our CRC Model is as follows:

- Entities: User (Abstract), Customer, Seller, Food, FoodTruck, FoodMenu, Order
- Use Cases: User Manager(Abstract), Customer Manager, Seller Manager, FoodTruck Manager, Order Manager
- Controller: Scene(Abstract), UserInformationScene, LoginScene, MarketScene, FoodTruckScene
- Interface: CommandLineInterface

All Entities classes are responsible for storing their corresponding attributes and perform basic operations like login for a User, add order to queue for FoodTruck, add Food for FoodMenu and such.

All Uses Cases are essentially “managers” for their corresponding entities, hence all named with “Manager” in the end. For example, a Customer Manager would be in charge of all the customers in the system, with functionalities such as login for a customer, create a customer account, as well as order food for the customer. Similarly, a FoodTruck Manager would be creating FoodTrucks, adding and removing Orders from order queue, updating the FoodTruck’s menu and such.

All Controllers are responsible for passing the commands from the interface to the user cases, getting information of the User, FoodTruck, and so on. Moreover, they send the information from the lower level classes and display the information to the user.

The only interface is the Command Line in which it serves the functionality of displaying information to the user and passing commands from the user to the controllers for them to handle.

Scenario Walk-through summary:

Paul runs the main method in the CommandlineInterface, and it then creates 4 scenes which are LoginScene, UserInformationScene, MarketScene and FoodTruckScene which add themselves to Scene.allScenes upon creation. The active scene is set to be LoginScene initially. Paul is then prompted to enter login information to login into the system. However, he does not have an account just yet. So he types in “register” command and enters “Seller” as the user type as well as all of the necessary information for this new account, then confirms it to create his new account. After that, he returns to the login menu and enters information to login. As he moves onto the next scene called UserInformationScene, he sees his account information listed on the console, where all this information is provided by user manager classes. Paul sees that he doesn’t have any funds in his wallet, so he types in “add fund” command to add some funds to his wallet.

He is now satisfied and used “view market” command to move onto the scene called MarketScene to see all food trucks available to him. He sees his food truck on the list and he selected it to see what it actually contains. The program then moves onto FoodTruckScene to display detailed information for Paul. From the food menu, he can see names, descriptions and prices of these food. After he finishes browsing through them, he decides to call it a day. So he uses “back”, “view user info” and “sign out” commands consecutively to sign out of the system, then types in “exit” to fully exit the program.

Skeleton Program summary:

Until now, our program has mostly covered the codes for the first three layers of the Clean Architecture Principle: Entities, Use Cases & Controller; as well as a rough User Interface.
Entities: User (Abstract), Customer, Seller, Food, FoodMenu, FoodTruck, Order

1. User abstract class: User abstract class is an abstract class representing an User instance. An User can be either a Seller or a Customer. They have some shared attributes including an accountName , an accountBalance, a nickname, a phoneNumber and a login status. For all User classes (User abstract, Seller, Customer), they have shared some public methods including: login, logout, addMoney, withdrawMoney, (Re)setPassword, (Re)setNickname. These are all associated with editing the class attributes. Besides, we have also provided all public getter methods for looking up the class attributes; and a toString abstract method that should be implemented in Seller, Customer that represents the User instance in String.
2. Seller class: Seller class is a Children class of User. Besides the inherited class attributes & methods from its parent class User, Seller class has added another class attribute named ownedFoodTruck which stored all FoodTruck (in an ArrayList -- will be changed to HashMap in next phase) owned by this seller. To edit this attribute, we have created two methods called addFoodTruck and removeFoodTruck, which each of them takes in an instance of FoodTruck and added/removed to our ownedFoodTruck. Besides, we have provided a getter method for the ownedFoodTruck; and have overridden the toString Method in form: “Account Name: xxx; Password: xxx; Nickname: xxx; PhoneNumber: xxx; Stored owned: TODO.”
3. Customer class: Customer class is another Children class of User. Besides the inherited class attributes & methods from User, Customer class has also added another class attribute named orderHistory which stores all Order(in an ArrayList -- will be changed to HashMap in next phase) that has been completed by this Customer. To edit this attribute, we have created two methods called storeOrder and removeOrder, which each of them takes in an instance of Order and added/removed to our orderHistory. Besides, we have also provided a getter method for the orderHistory; and have overridden the toString Method in form: “Account Name: xxx; Password: xxx; Nickname: xxx; PhoneNumber: xxx; Order HIstory: TODO.”

4. Food class: Food class is a class representing a Food instance. For a Food, it has attributes of: a foodName, price, id, label and a description (of the food). Label here represents the type of this food. For example, labels include Appetizer, Beverage, Meal, Dessert, Italian Food, Fast Food etc. Besides the getter methods, we have also included a changePrice method that allows us to change the price of this food; and a changeDescription method that allows us to re-describe this food. Moreover, it is noted here that Food class has used the interface CompareTo, meaning two Food instances can compare with each other following the rule that return 0 if two Food has the same name, and return 1 otherwise.
5. FoodMenu class: Food menu is an instance mainly recording a series of food that a certain FoodTruck has. It is a simple class that only has one attribute named foodList. We can add a food to this FoodMenu by calling addFood and remove one by removeFood (removeFood is not implemented yet). Besides, we can get a Food instance that is in the foodList by inputting the food name through createCopy. The toString method for a FoodMenu takes the form of: "Food name (1): xxx$ + food description; Food name (2): xxx$ + food description...."
6. FoodTruck class: FoodTruck class is a class representing a FoodTruck instance. It has attributes of truckName, location, serviceTimeStart, serviceTimeEnd, status (close or open), seller, orderHistory, order, orderQuene and a menu. FoodTruck has two overloading constructors that can optionally take in a seller or not. Within FoodTruck class, we can modifying the menu using updateMenu; change the status calling chageStatus; add order to history while calculating the rate in order history through updateOrderHistory; Add/remove order queues with addOrderToQueue/removeOrderWithID. In addition, we have created getter methods for all instance attributes, and a toString method taking the format of
   FoodTruck Name is located at xxx.
   The service time for this food truck is: xxx-xxx.
   The food truck is currently (not) operating.
   The rating of the food truck is xxx.
7. Order class: An Order instance has following attributes: a (unique) ID, foodtruck (which food truck to buy), foodList, totalPrice, customerName, customer(Phone)Number, sellerName, seller(Phone)Number, (customer's)rating and status (order received, in progress, order completed). The associated methods that modify these attributes include: changeOrderStatus, rateOrder (from 0-10). We have provided getter methods for all class attributes, and a toString method that follows:
   (ID)
   customerName: customerPhoneNumber
   FoodTruckName: FoodTruckPhoneNumber
   Food name (1): xxx$
   Food name (2): xxx$...

Total $: xxx

Use Case: UserManager(Abstract), CustomerManager, SellerManager, FoodTruck Manager, OrderManager

1. UserManager: UserManager is an Abstract class for SellerManager and CustomerManager. User Manager has three HashMaps mapping the user name to the User instance, including: customermap, sellermap and usermap. These three maps are created as a database for all user information. Whenever we have called createUser method, it requires us to specify the user type (Customer or Seller). For Customers, we are going to store the new customer information in both customermap and usermap; and store the new seller information in both sellermap and usermap (Similar way if we want to delete a User using deleteUser). UserManger has also implemented addMoney, withdrawMoney, checkBalance, setPassword, setNickname, getter methods for User instance which make use of some methods in User abstract class. We have done so because the Manager, as an intermediate layer between the controller and entities, must have ways to transfer commands from the controller to entities. In addition, Since the controller will only send an add/withdraw money etc. request based on the username, it is Manager's role to find its corresponding User instance to call the entity method. Last, UserManager has created some support methods for the controller, such as getUserType which returns the type of User; getUserByAccountName which returns all important information of the User with input username; CheckUserExist to see whether an account name has been created.
2. Customer/Seller Manager: Since most functions have been defined in the User Manager, Customer/Seller Manager, as Children classes of the User Manager, only implement some abstract methods in the User Manager class. Both classes have overridden the login method since they rely on their own database. Besides, they have created their own getCustomer/getSeller method that returns all Customer/Seller in their database.
3. FoodTruck Manager: FoodTruck Manager is another Manager Class that focuses on managing the operations of FoodTruck. A FoodTruck instance can be created through createFoodTruck in this class. Simultaneously, a FoodTruck has a similar HashMap as User Manager named food_truck that maps ID to a FoodTruck instance after the FoodTruck has been created. By default, we have provided some FoodTruck that we known through method createDefautlFoodTruck (The main purpose for this method is for testing the controller and as a reference for the scenery walk-through). FoodManager has also contained some methods associated with its instance variable, food_truck. Basically, all methods are checking or calling the FoodTruck instance from this ID to FoodTuck Map. For example, getFoodTruckDetail will find the FoodTruck based on its ID and return a summary information of this FoodTuck using a HashMap; getAllFoodTruckDescription return as a HashMap maps the FoodTruck's ID to its representation in String; getFoodTruckById return us the FoodTruck instance with given ID; and getExistFoodTruckName gives us all ID that has been used. As the manager for

FoodTruck, FoodTruck Manager also takes the role of transferring commands between controller and FoodTruck entity class. For example, we have created changeStatus, getOrderHistory, getMenu, getOrderQueue, getRating that directly pull information from the existing FoodTruck instance.

4. OrderManager: Similarly, OrderManager focuses on managing Orders. It has a similar HashMap called orders that maps the order's ID to the Order instance. Some related methods for orders include: getOrder based on ID and getOrderDetail which returns a mapping between order ID and that order details. OrderManager system is not completed yet, but we have still provide some useful functions including: createOrder which creates a new Order instance and stored it in orders database; getFoodMenu which takes in an Array of the name of the desired food and a target foodTruck to get an Array of Food instance in this given foodTruck that matches the desired food. getTotalPrice which calculates the total costs of all foods in the given array with the price under the given FoodTruck.

Controller: Scene (Abstract), LoginScene, UserInformationScene, Market Scene, FoodTruck Scene.

1. Scene (Abstract) : The Scene is an abstract class that is created to process all the commands from the toppest and make them applible by calling Use Cases Methods. To do so, we have called all Manager Class as local static variables within the Scene class. Since Scene needs to process all the commands, we need to create a static method field that keeps updating commands from the command interface through the method fillInField(ClearField is provided here to Clear all stored commands). As we want to use different Scenes to process different requests all the time, we have created a static map named allScene which stores all existing Scenes in the format of SceneClassName maps to the Scene instance; and creates an instance Scene variable activeScene that indicates which Scene we are using now (call getActiveScene.method() allow us to use this specific Scene). When we want to change a Scene to another, we can simply call the overloading switchScene method to change the acticeScene.
   The principle of Scene is to handle the input from the command interface, and construct a proper output based on the input. To achieve different goals, we have created four different Scenes that handle the input in different ways.
2. LoginScene: Login Scene is always initialized first at the outest layer. Login Scene has two main functions; register an account or Log In to an existing account. When receiving input commands starting with "register", the LoginScene will open the register mode that allows the outest layer input the register information including name, password, user type, nickname and phone number. We have a strict role on how these commands should be requested on the outest layer. For example, input "P[space] +password" for password, "U[space] +username" for username" etc. (These commands will be changed to a button in the real app so just provided a brief illustration here) Once we have filled out all the information, we can input "confirm" to create this new User account. When "confirm" is

pressed, the handleInput method actually calls the method to handle the output (Create a new User account). Here, we can achieve this by calling the UserManager where all information that has been input from input command (stored in field as introduced in Scene abstract class) is passed to the UserManager for creating a new instance. After registering an account, we can login to it by inputting “login” - “U [space]+Username”-“P[space]+password”- “confirm”. Then, we can play with our User account that changes the activeScene to UserInformationScene.

3. UserInformationScene: First of all, when we have logged in to the User Account, the UserInformationScene will automatically call the UserManager to load and display the information of our account. At UserInformationScene, We can receive requests from the command that ask the program to changeNickName (“change_nickname[space] + xxx”), addFund“add_fund[space] + xxx”, and changePassword(“change password”) etc. Our UserInformationScene does not cover all functions of a seller. But until now, we are able to view the MarketScene by calling “view market” or signing out to the account by calling “sign out”. For viewing the market, we are switching the activeScene to MarketScene; and switching the activeScene back to LoginScene for log out.
4. MarketScene/FoodTruck Scene: These two parts are not completed in Phase 0. But basically, in MarketScene, the user who uses this application will see all valid FoodTruck. The user could select one FoodTruck and direct to a FoodTruck Scene, or request to return back to the UserInformationScene. When directing to a FoodTruck Scene, this user can see the list of all valid food items that he/she can order. An order will be created after the user pays the money (remove money from account using the method from userManger) and the seller receives money. We may also add some other scenes for rating in the next coming phase.

User Interface: CommandLineInterface

1. CommandLineInterface: This is where the user can input their command. We process their commands by creating a BufferReader that reads each line they have input. Initially, we have sent these commands to a LoginScene, but also initialized other Scene in order to store them in allScene so that we can switch the scene while running. We have included a while loop that will never break unless we quit the application. Within the while loop, we constantly call the activeScene to process the user’s request, which realizes our goal to execute the user’s request through all layers of the code.

Individual Responsibilities:

Codes Implementation:

- Steven: User / Seller / Customer
- William: Food / FoodTruck
- Tom: Order / FoodMenu
- Hongshuo: (User / Seller / Customer) Manager
- Yanke: (FoodTruck / Order) Manager

- Mike: Scene / Command Line

Specification: written and edited by Hongshuo, and William

CRC Model: The CRC Model was created in a combined effort of the team through discussions.

Scenario Walk-through Summary: written and edited by Mike

Skeleton Program Summary: written and edited by Steven

Open Questions:

- How to design a good user interface, and what framework would be most suitable to use?
- Where should we put the database for the information?
- How to connect our program to remote sources such that it can perform online features? (Connect to server and such)
- How to resolve circular references for classes in the same layer?
- How can we check and confirm that the principles of clean architecture are being followed and implemented in our design of codes?
- What happens when we need to access classes that are two or more layers higher than the current layer, for example if we need the controller to interact with an entity class, what would be the optimal solution to that, or should we just avoid this situation at all cost?

Design Spotlights:

- We strictly followed the four layers of clean architecture, which are entity classes, use cases, controllers, and interface.
- The classes in the higher level never access the lower level classes.
- We had create a good outline of CRC cards for the entities and uses cases that made implementing codes of those two layers easier
- The single responsibility idea is followed thoroughly in the Controller layer (as well as other layers but best displayed here). This is shown by the different subclasses of Scene. For instance, login scene is purely for managing the login part of the program, MarketScene is purely for managing browsing the food Trucks, and so on.

Future Plans:

1. Set up the database system for back-end
2. Begin building the front-end, which is mainly the user interface.
3. Continue to add more features to the program, such as but not limited to the rating system, order status control, pickup time feature and more.
4. Connect the program to real-world attributes, for example, synchronize it with real-time and provide actual locations for Food Truck to choose from.
5. Continue to add more javadoc to make the codes more accessible and easily understandable.
6. Keep on fixing bugs / improve things like more descriptive method/variable names
7. Most important: Focus on implementing the connection between the Scene and Manager. (Example, method that deducts money when purchasing, method that sends food orders to Order Manager etc.) Revise Order Manager if needed.