

Phase 1 Progress Report

- By Group ASOUL

Members:

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

Specification summary: (William)

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.

In regards to Phase 1, there were several implementation details changed.

We've merged the two user types (customer/seller) into a single "User" type. Each user starts off with an "unactivated" FoodTruck under its name. This is to simplify the Scene implementations in the controller layer. This means now, to use the app, a user won't need to specify the account type when registering the account, and can both sell food from its food truck, or buy from other's food truck. The functionalities of customer and seller are still preserved in the User class and similar to what is being described in Phase 0.

Order status now takes only two forms: in progress or completed. The seller is responsible to end the order once it is completed.

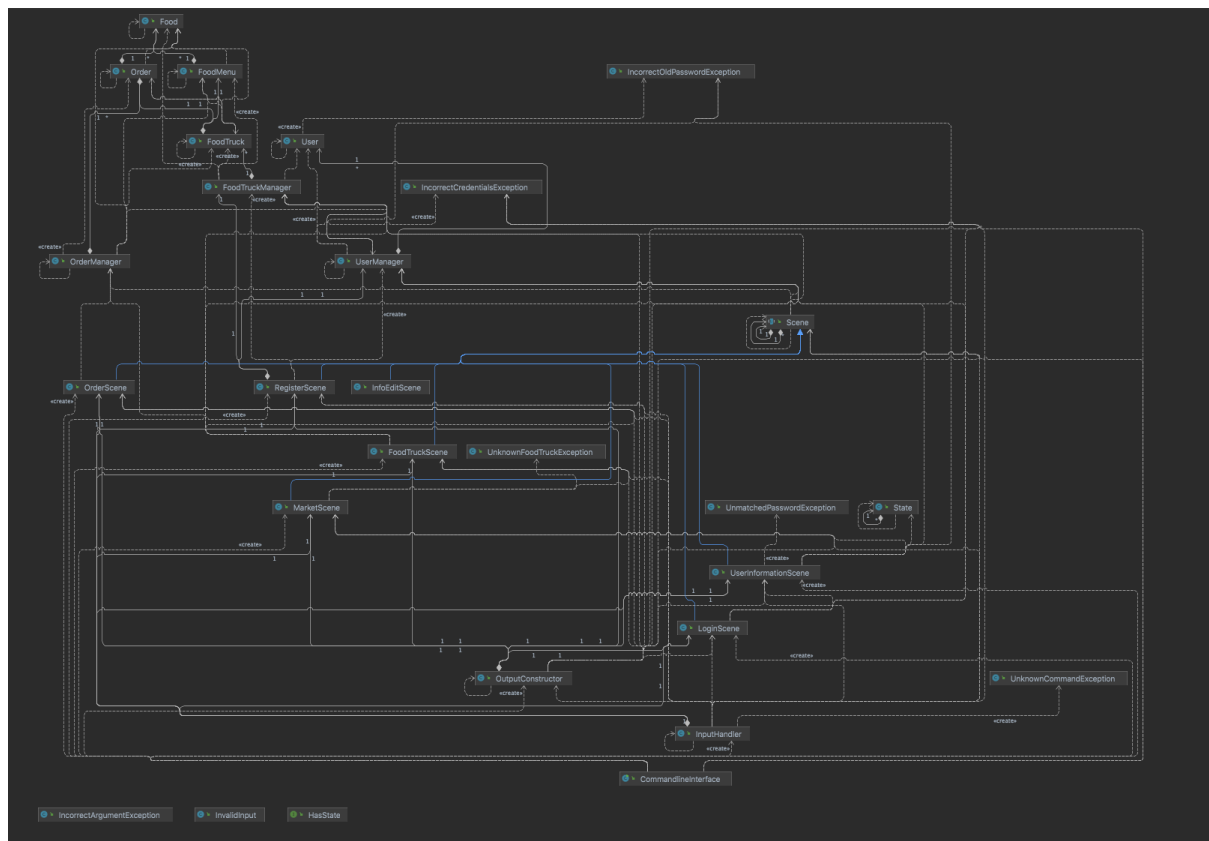
Market Scene is something we further implemented. It only displays active Foodtrucks (so the default food truck created upon the creation of each user account won't be displayed until it is activated). Then, it sorts and displays food trucks by distance or name or ratings.

Major Design Decision Description: (Everyone)

1. Redesign of the controller layer.
2. Separate out the responsibility of parsing input and constructing output and make them into separate classes.
3. Merging the two user types Customer and Seller into a single User class.
4. Change the construction logic for order id.
5. (Future Plan) Add support to the android interface to make it easier for users to interact with the program
6. Packaging the code by layers.
7. Each user has a food truck, which starts with unactivated and empty.
8. Every method in use case classes has been made static to allow easier communication between these methods.

Clean Architecture: (William)

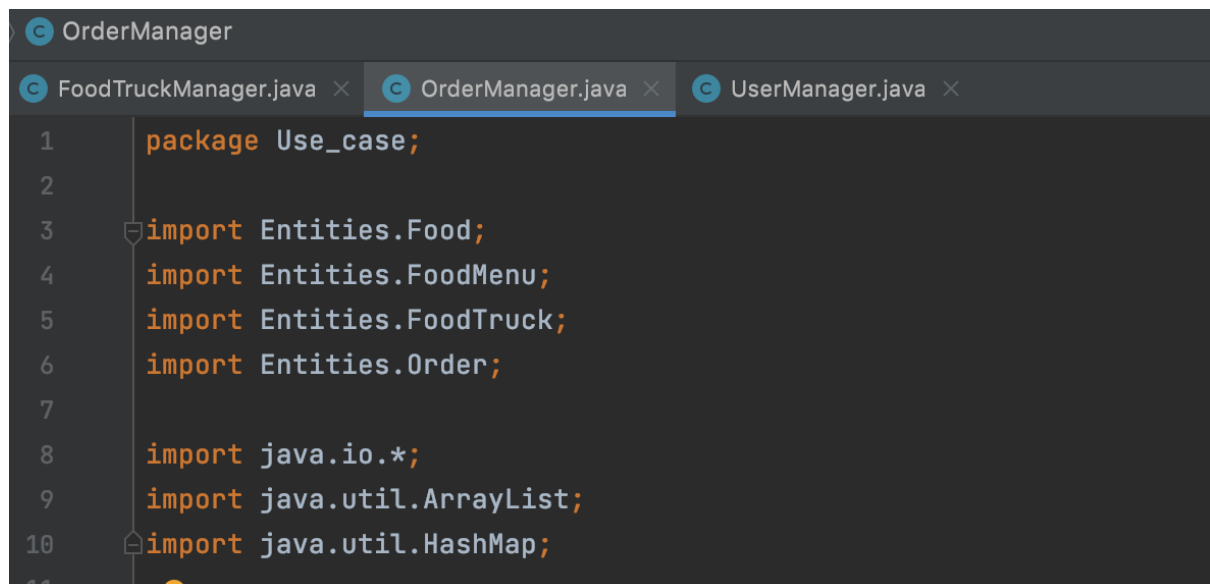
UML Class Diagram:



Just like how we've divided our program into four layers on the CRC Model, the implementation follows the same pattern and the dependencies have not changed much since Phase 0.

For all the classes, if you would randomly check the import statements, none of them would not follow the dependency rules. For example, looking at classes in Entities, all of them do not import any classes of the outer layers (use cases or controllers). Similarly, all classes in the use cases layer would only be importing classes in the Entities layer; all classes in the controller layer would be only importing the use cases classes and none of the entities.

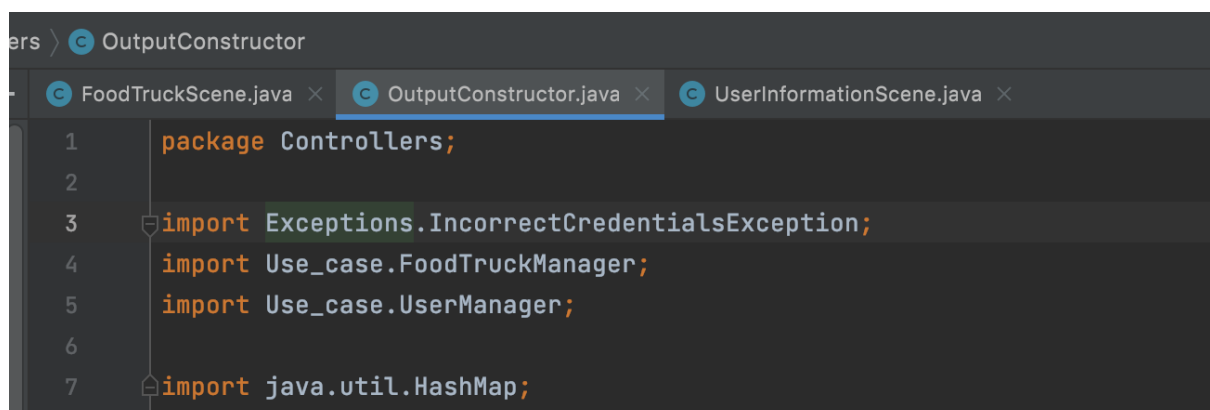
Example 1: OrderManager, which is a use case class, imports only Entities. (and additional outside packages)



The screenshot shows an IDE with three tabs: FoodTruckManager.java, OrderManager.java (selected), and UserManager.java. The code in OrderManager.java is as follows:

```
1 package Use_case;
2
3 import Entities.Food;
4 import Entities.FoodMenu;
5 import Entities.FoodTruck;
6 import Entities.Order;
7
8 import java.io.*;
9 import java.util.ArrayList;
10 import java.util.HashMap;
```

Example 2: OutputConstructor, which is a controller, imports only use cases.



The screenshot shows an IDE with three tabs: FoodTruckScene.java, OutputConstructor.java (selected), and UserInformationScene.java. The code in OutputConstructor.java is as follows:

```
1 package Controllers;
2
3 import Exceptions.IncorrectCredentialsException;
4 import Use_case.FoodTruckManager;
5 import Use_case.UserManager;
6
7 import java.util.HashMap;
```

In terms of serialization and database, we see that first, the entities are implementing the Serializable interface. Then, methods in the use case classes like UserManager define how to construct the database which the former are later called by the Scene class in the controller layer upon initiating and exiting the program and sent outside as files.

Example: User class implements Serializable

```
public class User implements Serializable {
```

Then, UserManager defines a method that constructs the user database

```
public static void constructUserDataBase() throws IOException, ClassNotFoundException {
    try {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("name: ./data/user info"));
        userMap = (HashMap<String, User>) ois.readObject();
        ois.close();
    } catch (EOFException e) {
        // Do nothing, no seller has been registered
    }
}
```

Lastly, Scene, one of the controllers, calls the constructUserDataBase from UserManager and outputs the file.

```
public static void init() throws ClassNotFoundException, IOException {
    FoodTruckManager.constructFoodTruckDataBase();
    UserManager.constructUserDataBase();
    OrderManager.constructOrderDataBase();
}
```

SOLID design principle: (Steven):

Moving from Phase 0 to Phase 1, our group mainly focuses on cleaning the codes to make them follow the SOLID principle.

Single responsibility principle:

Previously in Phase 0, our codes mostly follow the SRP. We have entity classes which each of them is responsible for one single entity object (seller, order, foodtruck etc.). We also have use-case classes (manager classes) that each of them is responsible for editing one single entity object. Our main change in Phase 1 is we have redesigned our controller classes so that it also follows the SRP. Before our change, the controller classes did many functions, for example, one single controller is responsible for calling many different managers to edit their entity, as well as

responsible for changing scenes. But now, our one single controller is just responsible for one functionality.

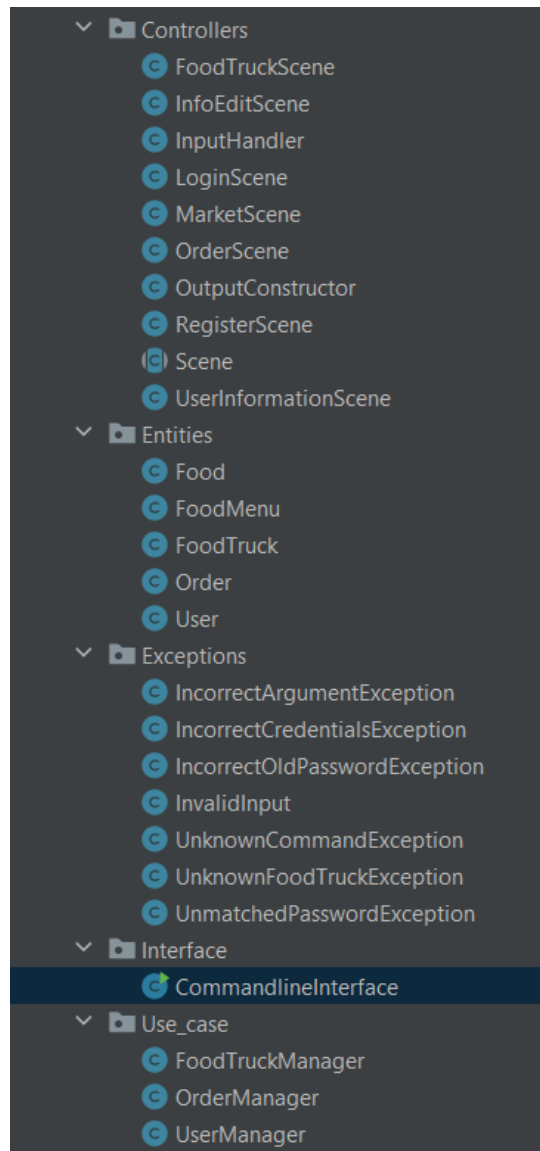


Figure S1 - Our files follow the SRP

Open/Close Principle:

As an extension to the SRP, our code has also followed the OCP. Since each class in our program has a single responsibility, we have created sub-methods that are only related to the responsibility of this class. Thus, while moving from Phase 0 to Phase 1, we have never modified methods in the deepest layers (entity, manager etc.) because all the existence methods are useful and have been used. Instead, to meet the need of new functionalities, we have extended our classes by creating new methods in the corresponding class.

Liskov substitution principle:

As an extension to the OCP, all subclasses we created in our code have followed the LSP. For example, we have created a parent class “Scene”, the “Scene” class has some common functions such as switchScene. All children classes of “Scene” such

as “OrderScene” do not modify these common functions, instead, they have extended their class for more methods to meet their special behaviours.

Interface Segregation Principle:

Our program does not write an interface by ourselves, but we have used some built-in interfaces such as Comparable. These interfaces all follow the ISP that is small so classes that use these do not need to override some useless methods. Our parent classes also follow this principle. In our parent class, we have only defined methods, which are useful and share commons with all its children classes.

Dependency Inversion Principle:

Refer to Figure S1, it shows how we divided our code into different layers including entities, use-cases, controllers, interfaces. Our layers have followed the DIP, where the entities are the lowest-level classes that are created first by us. Use-cases are the second-level classes that manage the entity classes but are not dependent on them. Controllers are classes that send requests to the Use-cases classes only, while the Interfaces are classes that send requests to the Controllers only. Under the DIP, all lower-level layers' classes have nothing to do with their higher-level layer's classes.

Packaging Strategies (Vector):

In our project, we choose to package our code mainly by layers. As Figure S1, we package the entity classes in the “Entities” folder, the use case classes in the “Use_case” folder, the controller classes in the “Controller” folder, and the interface classes in the “Interface” folder, whose package strategy is packaging by layers. Besides, we package the exception classes in the “Exception” folder. These classes help us to throw the correct type of exception when the program runs into an error, thus helping us fix the bugs.

The reason why we choose this packaging strategy is that by packaging by layers, our classes will be more organized and our design will be clearer. In addition, this packaging strategy makes it easy to follow the clean architecture, since the package strategy shows our project structure.

Summary of design patterns (Vector):

We implement a facade pattern in our controller classes. In the controller layer, we construct a facade class called “InputHandler”, and the scene classes in the controller layer are the complex sub-system under the facade class. When the client's input is passed from the interface layer to the controller layer, the program will call the facade class “InputHandler”. And the facade class directs the request to the correct object in the complex sub-system. The facade pattern provides a simplified interface of controller layers, thus making it much easier for interface layers to pass the request to the controller layer. Besides, the facade pattern separates the interface and complex sub-system in the controller layer, making our code clearer and more readable and easier to comprehend and maintain.

We also plan to implement a series of design patterns in our project later. For example, we plan to implement the adapter design pattern to our use case layer later. Since many methods in the use case classes need input from entities classes, and the controller layer can only pass the string input, we need an adapter pattern to adapt the interface of use case class and the controller class. Currently, we solve this problem by overloading each method in our use case class, but this makes our code mass and unreadable. Thus, we are going to use an adapter design pattern to eliminate mass overloading methods in our use case classes to make our code clearer and readable.

Progress Report:

Open Questions:

What are some problems with input-driven output constructors?

How to change current controllers and interfaces to android?

How to implement a finite state machine for the controller layer?

When in a situation where a design pattern may apply but we have to change a lot of code and class, what should we do?

Group Member Contribution (Everyone):

Tony, Mike, Tom: Controller Layer refactoring, implementation of facade design pattern, data serialization

Vector: Refactor the code in the use case layer and entity layer. Add the methods in these two layers which are needed to implement the extended function. Write the design document.

Steven: Add some new methods in entities and use cases layer, implementation of the progress report.

William: Order/FoodTruck Rating system implementation, progress report writing.

Successful Design: (Steven)

Until now, we believe our design is successful. First, our design has a specific goal that helps our school to manage the food trucks. Second, our design has provided many useful functionalities. We have established a system that can store all information of the food truck, users and their associated order history. It can help the food truck to improve their services (food quality, after-purchase service etc.). Also, we have created a platform that can possibly extend the food truck services such as online ordering, food delivery in the future. Finally and most importantly, the code in our design is clean (following the SOLID,

clean architecture principle etc.), so it is easy to understand what each part of our design does. Besides, we have provided brief explanations in the user interface so that users can easily follow our instructions to do certain behaviours, which makes our design easy to use.

Next step:

1. Continue working on constructing the order system and the rating system.
2. Refactor code to eliminate the code smell.
3. Design an android interface and adapt the program to the android system.
4. Add more tests and improve the docstring.