Project Report
myFoodora - Food Delivery System

Object oriented software engineering 2.6 - CS - PARIS - SACLAY
(2023-2024)
GITHUB PROJECT

31. May 2024

Avery Atchley
Arvid Ehlert

.

Contents

1 Introduction

The goal of this project is to develop a software solution, called myFoodora, whose functionality is similar to that of nowadays Food Delivery Systems such as, for example, Foodora (www.foodora.fr) or Deliveroo (www.deliveroo.fr).

1.1 Goal and Scope

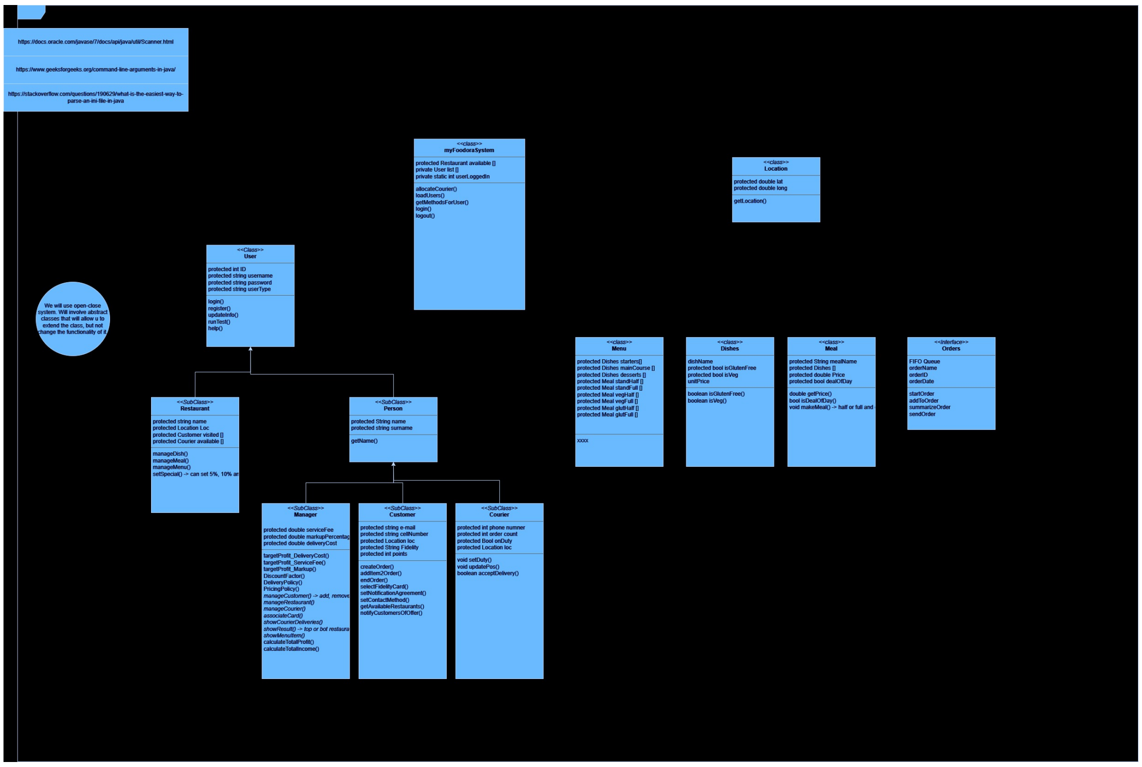
1. **myFoodora core:** design and development of the core Java infrastructure for the myFoodora system
2. **myFoodora user interface:** design and development of a user interface for the myFoodora system

1.2 Structure of this report

In this report, we provide an overview of our myFoodora system, beginning with UML diagrams that illustrate the system's structure and key components. We delve into the main characteristics of our system, outlining its core functionalities and how they contribute to enhancing user experience. Additionally, we discuss the design decisions that influenced the architecture and development of the system, highlighting key considerations such as scalability, flexibility, and user accessibility. Furthermore, we analyze the advantages and limitations of our system, addressing areas of strength and areas for potential improvement. Test scenarios are presented to demonstrate the functionality of the myFoodora system and guide users on how to effectively utilize its features. Finally, we provide insights into the workload split between team members, detailing each member's contributions to the development and implementation of the system. Through this comprehensive report, readers gain a thorough understanding of the myFoodora system, its design principles, functionality, and the collaborative effort involved in its creation.

2 Main Characteristics and Design Choices

In the following chapter, we introduce the UML diagram developed at the beginning of this project (Fig. 2) as well as our finalized UML diagram after completing the project (Fig. 3).
The first diagram outlined the necessary characteristics of our system, and was referenced as the system was realized. Throughout development, however, the structure of our system also changed. This resulted in an updated version of our UML diagram. This updated version shown in Fig. 3 shows all variables and functions for all classes except the test classes. Inheritance is shown as a solid line with an arrow on the end of the parent class. A dotted line is used to signal implementation of interfaces. The CLUI classes are shown in a different color than the normal system classes. For improved readability we decided not to show aggregation and composition arrows in the UML diagram.
This chapter details these design choices that led to the changes in the UML throughout the project in detail and how they affect the functionality of the system.



https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html
https://www.geeksforgeeks.org/command-line-arguments-in-java/
https://stackoverflow.com/questions/190629/what-is-the-easiest-way-to-parse-an-ini-file-in-java
<<class>> myFoodoraSystem
protected Restaurant available []
private User list []
private static int userLoggedIn
allocateCourier()
loadUsers()
getMethodsForUser()
login()
logout()
<<class>> Location
protected double lat
protected double long
getLocation()
We will use open-close system. Will involve abstract classes that will allow u to extend the class, but not change the functionality of it.
<<Class>> User
protected int ID
protected string username
protected string password
protected string userType
login()
register()
updateInfo()
runTest()
help()
<<SubClass>> Restaurant
protected string name
protected Location Loc
protected Customer visited []
protected Courier available []
manageDish()
manageMeal()
manageMenu()
setSpecial() -> can set 5%, 10% an
<<SubClass>> Person
protected String name
protected string surname
getName()
<<SubClass>> Manager
protected double serviceFee
protected double markupPercentag
protected double deliveryCost
targetProfit_DeliveryCost()
targetProfit_ServiceFee()
targetProfit_Markup()
DiscountFactor()
DeliveryPolicy()
PricingPolicy()
manageCustomer() -> add, remove
manageRestaurant()
manageCourier()
associateCard()
showCourierDeliveries()
showResult() -> top or bot restaura
showMenuItem()
calculateTotalProfit()
calculateTotalIncome()
<<SubClass>> Customer
protected string e-mail
protected string cellNumber
protected Location loc
protected String Fidelity
protected int points
createOrder()
addItem2Order()
endOrder()
selectFidelityCard()
setNotificationAgreement()
setContactMethod()
getAvailableRestaurants()
notifyCustomersOfOffer()
<<SubClass>> Courier
protected int phone numner
protected int order count
protected Bool onDuty
protected Location loc
void setDuty()
void updatePos()
boolean acceptDelivery()
<<class>> Menu
protected Dishes starters[]
protected Dishes mainCourse []
protected Dishes desserts []
protected Meal standHalf []
protected Meal standFull []
protected Meal vegHalf []
protected Meal vegFull []
protected Meal glutHalf []
protected Meal glutFull []
xxxx
<<class>> Dishes
dishName
protected bool isGlutenFree
protected bool isVeg
unitPrice
boolean isGlutenFree()
boolean isVeg()
<<class>> Meal
protected String mealName
protected Dishes []
protected double Price
protected bool dealOfDay
double getPrice()
bool isDealOfDay()
void makeMeal() -> half or full and
<<Interface>> Orders
FIFO Queue
orderName
orderID
orderDate
startOrder
addToOrder
summarizeOrder
sendOrder

Figure 2: Beginning UML Diagram

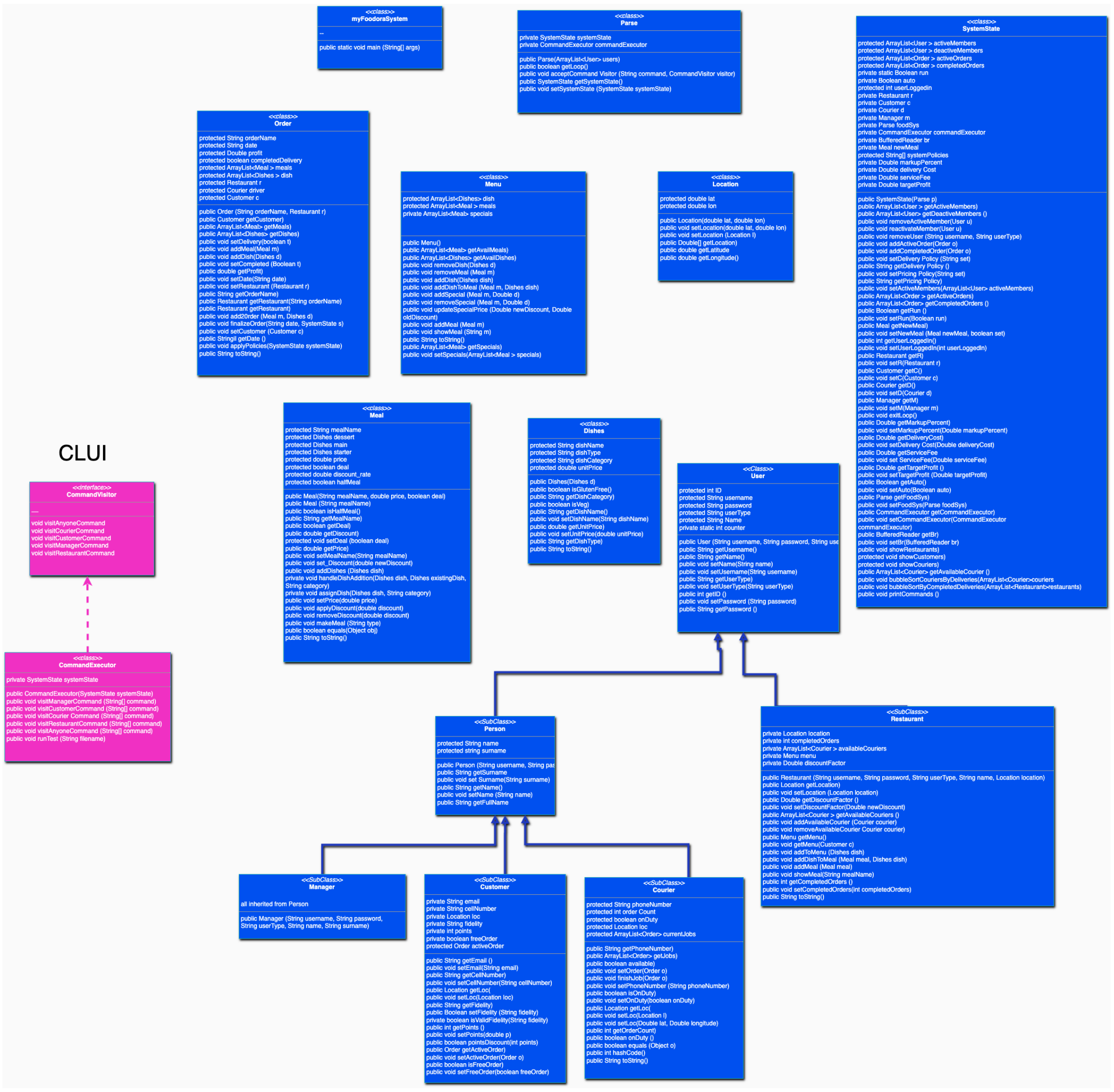


Figure 3: Final UML Diagram

2.1 Main Characteristics

Our development began with creating a comprehensive UML diagram outlining the system architecture. We followed this with test-driven development, employing both the Command Line User Interface (CLUI) and JUnit tests to ensure robustness and adherence to the Open-Closed Principle. By incorporating both forms of tests, we determined if each class functioned properly. Moreover, this structure of development and testing showed how the different classes worked together through incremental progress. By choosing an incremental design and test pattern, we ensured that each segment of the system worked before incorporating more items. This enabled us to alter the classes to achieve the desired outcome before reaching a point where changing one segment of code might destroy the entire system. Moreover, this testing system guided our design choices, emphasizing extensibility and maintainability. The overall structure of our system is modular, with clearly defined responsibilities for each class, promoting code reuse and ease of understanding.

2.2 Design Decisions

2.2.1 Class Composition

Class composition involves using objects of a given class as attributes of a new class. For example, the `Location` class is used by various user classes (e.g., `Restaurant`, `Courier`, etc.). This design pattern promotes reusability and modularity. Our UML diagrams show this separation, with `Location` being an isolated class used by different user classes to handle location-related attributes. Class composition is also used extensively in different classes in the `Food` section of our code (e.g., `Restaurant`, `Menu`, etc.).

2.2.2 Inheritance

Inheritance is used extensively in the user hierarchy. The `User` class serves as a base class with sub-classes such as `Restaurant`, and `Person`, with its own sub-classes `Manager`, `Courier`, and `Customer`. This allows common methods like `login`, `register`, and `updateInfo` to be defined once in the `User` class and inherited by all sub-classes. Moreover, the inheritance utilized by the `User` class enables a structured system for how users of the myFoodora system are stored. Users have essential attributes (e.g., `name`, `username`, `password`, etc.), but are open to more specific attributes in their sub-classes.

2.2.3 Shadowing and Overriding

`Shadowing` was not utilized in our project, however, `Overriding` was extensively used throughout the different classes. `Overriding` in each class largely occurred through the toString and equal functions for different class objects (e.g., `Dishes`, `Meal`, etc.). By using `Overriding`, we make it easier to print various objects and to determine if two objects of the same type are equal. Thus, this design pattern simplified the overall complexity of our system.

2.2.4 Error and Exception Handling

Error and exception handling are crucial for system robustness. Our design includes try-catch blocks for sections of code that might encounter errors in the CLUI or Scanner interface, as well as validation checks to ensure that methods accept the correct visitor, handle unexpected inputs gracefully, and provide meaningful error messages, helping the user effectively steer the system.

Below is a portion of our `visitManagerCommand` method to show how we handle errors using both try-catch blocks and if-else conditions. Here's how each mechanism works:

Using If-Else Conditions

If-else conditions are used to validate input and ensure that only valid commands are processed. Once the `parse` class confirms that a valid command has been called, the command is sent to our `commandExecutor` class. This class accepts five types of visitors: `Manager`, `Restaurant`, `Customer`, `Courier`, and `Anyone`. For each visitor type, we verify that the current user has permission to use that command. Specifically, in the provided code, we check if the user logged into the system has the necessary permissions to execute a manager command. Then, we validate the format of the command and either proceed with processing the command or provide a specific error message.

```

@Override
public void visitManagerCommand(String[] command) {
    if (command[0].equalsIgnoreCase("registerrestaurant")) {
        if (systemState.getUserLoggedIn() != 1) {
            System.out.println("User cannot access this command");
            return;
        }
        if (command.length != 6) {
            System.out.println("Invalid Command. Use:
    registerRestaurant <name> <Latitude> <Longitude> <username> <
    password>");
            return;
        }
        // Additional processing...
    }
}
```

Using Try-Catch Blocks

Try-catch blocks are used to handle potential exceptions that might occur during the execution of a method. After validating the input, we attempt to parse the command and create a new restaurant. If a parsing error occurs (e.g., invalid latitude or longitude format), the catch block captures the specific `NumberFormatException` and provides an appropriate error message. Another catch block handles any other unexpected exceptions, providing a general error message and outputting the stack trace for debugging purposes:

```
@Override
public void visitManagerCommand(String[] command) {
    if (command[0].equalsIgnoreCase("registerrestaurant")) {
        if (systemState.getUserLoggedIn() != 1) {
            System.out.println("User cannot access this command");
            return;
        }
        if (command.length != 6) {
            System.out.println("Invalid Command. Use: registerRestaurant <name> <Latitude> <Longitude> <username> <password>");
            return;
        }
        try {
            Location loc = new Location(Double.parseDouble(command[2]), Double.parseDouble(command[3]));
            Restaurant r = new Restaurant(command[4], command[5], "Restaurant", command[1], loc);
            System.out.println("Successfully added");
            systemState.getActiveMembers().add(r);
        } catch (NumberFormatException e) {
            System.out.println("Invalid latitude or longitude format.");
        } catch (Exception e) {
            System.out.println("An error occurred while registering the restaurant.");
            e.printStackTrace();
        }
    }
}
```

2.2.5 Abstract Classes and Interfaces

Although we do not use abstract classes or interfaces extensively, they could have been used for the `User` class to prevent the instantiation of a generic user. However, since we control the exact input options and ensure that only specific subclasses (like `Restaurant`, `Manager`, `Courier`, and `Customer`) are instantiated, the need for abstract classes is minimized.

On the other hand, we do utilize `Interfaces` within our `myFoodora` system through the visitor pattern. This design pattern defines an interface for visitors, which includes visit methods for each concrete element type. This ensures that any visitor class must implement these methods. The elements that can be visited also implement an interface, typically with an accept method. This method accepts a visitor and calls the appropriate visit method on the visitor. By using these interfaces, the Visitor pattern promotes decoupling between the objects and the operations performed on them. Each concrete element class implements the `ItemElement` interface, and any concrete visitor class implements the `Visitor` interface. Thus, the overall complexity of our system is reduced. Below is our `commandVisitor` interface, which shows the different visit methods for each concrete element type.

```
package Commands;

public interface CommandVisitor {
    void visitManagerCommand(String [] command);
    void visitCustomerCommand(String [] command);
    void visitCourierCommand(String [] command);
    void visitRestaurantCommand(String [] command);
    void visitAnyoneCommand(String [] command);
}
```

2.2.6 Open-Closed Principle

The open-closed principle states that software entities should be open for extension but closed for modification. In our system, this principle is achieved through several strategies. Some examples:

- **Using Inheritance:** The user classes are structured using inheritance. We have a base `User` class with subclasses and sub-subclasses for specific user types. For example, the login method relies on the variables defined in the `User` class and inherited by all user types (see UML diagram in Figure **??**). This allows for the login method to be defined for all users, no matter how many extra defining features the different users have.
- **Command Executor Class:** All commands are bundled within a single `CommandExecutor` class, simplifying command management.
- **Location Class:** A separate `Location` class is used, and any class requiring a location instance simply calls it. This modular approach allows easy addition of different user types without modifying parent or sibling classes.

2.2.7 Factory Pattern

The factory pattern is implemented to ensure that clients interact with the system through a predefined interface:

- **Controlled Object Creation:** Clients use the Command Line User Interface (CLUI) and a set of predefined methods from the `CommandExecutor` class, preventing arbitrary object creation.
- **Configuration Files:** When adding new dishes or similar objects, the client creates entries in a text file that can be read later. The source code remains unchanged with the `CommandExecutor` acting as a factory class.
- **No Abstract Factory Pattern:** Although we use a factory pattern, we do not implement an abstract factory pattern. Clients cannot create new dish types or similar objects independently.

2.2.8 Model-View-Controller (MVC) Pattern

Our system follows the MVC pattern:

- **Model:** Data is stored in external files, e.g. the StartUp.txt file which loads an initial system state- not a .ini file.
- **View:** The MVC pattern separates the user interface from the data handling. The commands typed into the CLUI are taken from the command line and sent to our `Parser` class, which dissects the input to determine if the provided command is valid. Valid commands are then transformed into specific command objects (e.g., `ManagerVisitor`). These command objects implement a common `Command` interface, which declares an `accept` method for the visitor. The command objects are then passed to the `CommandExecutor` class, which implements the `CommandVisitor` interface. The `CommandExecutor` defines the operations for each type of command by implementing the different `visit` methods available for each user type. This approach ensures that the user interface (View) is separated from the main data handling core of our system (Model).
- **Controller:** The Controller acts as an intermediary that uses the Visitor pattern to manage different types of commands in a clean and decoupled manner. This part of the MVC manipulates data and updates the view for the user interface.

2.2.9 Singleton Pattern

In our project, we opted not to implement the singleton pattern in a traditional sense. Instead, we utilized a static variable to manage the generation of unique user IDs. This approach facilitated the creation of distinct identifiers for each user by ensuring that the ID generation process remained centralized and consistent across the system. While we did not adhere strictly to the singleton pattern, our implementation achieved the desired outcome of ensuring uniqueness for user IDs.

2.3 Advantages and Limitations

Advantages:

- Open-close principle to a certain degree.
- Users can fulfill the actions they are supposed to fulfill.
- Added functionalities giving users more options within the system.
- Code can be utilized to implement more complex functionalities.

- Easy-to-understand error messages providing clear instructions on how to utilize the system correctly.

Limitations:

- Adding new meal types without changing the classes (array xxx in class yyy) prevents the abstract factory pattern from always being implemented.
- The current system takes into account the two required policies, however, the implementation can be improved with more time.

3 Test Scenarios and System Utilization

Test Scenarios:
We have four different scenarios, each of which can be executed using the CLUI (the .txt ending is optional). To run the program, run the main file. Test by entering one of the following tests:

```
runtest testscenario1.txt
runtest testscenario2.txt
runtest testscenario3.txt
runtest testscenario4.txt
```

Here follows a quick description of the different scenarios:

1. **Test Scenario 1:**
   Purpose: Demonstrate the overall functionality of the system and key features for managers and customers, while highlighting error-handling capabilities. This scenario is the most important and extensive one; the others serve to show a few detailed functionalities of the system for other users.

   Purpose of the Test Case: The main goal of this test case is to evaluate how well the system handles errors and invalid commands. Specifically, it aims to:

   - Confirm that the system can identify and respond to attempts to interact with non-existent entities.
   - Ensure that commands with incorrect formats or missing parameters are properly rejected.
   - Check the system's session management, such as preventing multiple logins without logging out.
   - Test the robustness of the system's error-handling mechanisms.

   Brief description of actions: In this scenario, we begin by logging in as a manager to set up the system, including registering a new restaurant and courier. After setting the stage, the manager logs out, and a customer logs in to place an order. Along the way, we deliberately introduce some incorrect commands and references to non-existent entities to see how the system handles errors.

   Details: This scenario showcases both the system's functionality and its error-handling prowess. For example, what happens when a customer tries to order from a restaurant that doesn't exist? Or when a command is entered with the wrong format? The system responds with helpful error messages, such as:

   - `"Unable to find Restaurant.  Try again using correct name."` for non-existent restaurants.

- `"Invalid Command. Use following format: registerCourier <firstName> <lastName> <username> <position> <password>"` for incorrect command formats.
- `"User already logged in. Logout first."` when a user tries to log in without logging out first.

These interactions ensure that the system provides clear, useful feedback to users, helping them correct their mistakes quickly.

2. **Test Scenario 2:**
   Purpose: Login as a restaurant user and perform the different actions attributed to a restaurant user. This includes managing menu items, processing orders, and handling customer interactions.

3. **Test Scenario 3:**
   Purpose: An extended version of Scenario 1, including the activation and deactivation of users. This scenario tests the system's user management capabilities more thoroughly.

4. **Test Scenario 4:**
   Purpose: Login as a courier and explore the functions related to courier activities. This includes viewing delivery assignments, updating delivery status, and managing courier profiles.

System Utilization:
To maximize the utility of our system, we encourage every user to utilize the 'help' command within the Command Line User Interface (CLUI). This command serves as a comprehensive guide, displaying all available commands tailored to each user type, even for users who have not signed in yet. By accessing the 'help' command, users can familiarize themselves with the system's capabilities and understand the correct syntax for executing commands.

Here's how the 'help' command enhances the user experience:

1. **Accessible Guide**: The 'help' command provides an easily accessible guide for users at any point during their interaction with the system. Whether they are new to the platform or need a quick reference, the 'help' command is there to assist them.

2. **Complete Command List**: Users can view a complete list of available commands specific to their user type. This includes commands for customers, managers, restaurants, and any other user roles defined within the system. Each command is accompanied by a brief description to explain its purpose and usage.

3. **Error Handling Guidance**: In cases where users input commands incorrectly, the system responds with an error message. This message not only notifies users of the error but also provides guidance on how to correct it. By referring back

to the ‘help‘ command, users can identify the correct syntax and parameters for each command.

4. **Encourages Exploration**: The ‘help‘ command encourages users to explore the system’s functionality and capabilities. By presenting a comprehensive list of commands, users may discover features they were previously unaware of, leading to a more enriched user experience.

5. **Promotes Efficiency**: Users can quickly find the commands they need without having to search through extensive documentation or manuals. This promotes efficiency in executing tasks and reduces the learning curve associated with using the system.

By incorporating the ‘help‘ command into their workflow, users can leverage the full potential of the system and streamline their interactions, ultimately enhancing their overall experience.

4 Workload Distribution

The following table shows the workload distribution and its realization, detailing who was responsible for design, coding, and JUnit testing for each class.

Class/Task	**Design**	**Code**	**JUnit Test**
`myFoodora.java`	both	Avery Atchley	Arvid Ehlert
`SystemState.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`CommandExecutor.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`CommandVisitor.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`Parse.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`Dishes.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`Meal.java`	both	Avery Atchley	Arvid Ehlert
`Menu.java`	both	Avery Atchley	Arvid Ehlert
`Order.java`	Avery Atchley	Avery Atchley	Arvid Ehlert
`Courier.java`	both	Avery Atchley	Arvid Ehlert
`Customer.java`	both	Avery Atchley	Arvid Ehlert
`Location.java`	both	Avery Atchley	Arvid Ehlert
`Manager.java`	both	Avery Atchley	Arvid Ehlert
`Person.java`	both	Avery Atchley	Arvid Ehlert
`Restaurant.java`	both	Avery Atchley	Arvid Ehlert
`User.java`	both	Avery Atchley	Arvid Ehlert

Table 1: Workload Distribution