Software Engineering Department

SOFTWARE CONSTRUCTION & DEVELOPMENT

Submitted by

NIRANJAN LAL | 63416

Lab Instructor

MISS LAILA BALOCH

Lab Engineer, Department of Software Engineering,

Faculty of Information & Communication Technology,

Winter Session – 2025

BALOCHISTAN UNIVERSITY OF INFORMATION TECHNOLOGY, ENGINEERING AND MANAGEMENT SCIENCES, QUETTA.

1. Introduction

1.1 Project Description This project is an **Online Food Ordering System (Backend Simulation)**, developed using **Python** as part of the Software Construction & Development (SCD) Lab. The main goal of this project is to demonstrate the practical usage of **Software Design Patterns** inside a real-world e-commerce scenario.

The system allows users to:

- Browse a structured menu (Categories and Items).
- Add items to a shopping cart.
- Place orders seamlessly.
- Automatically notify the Restaurant and Delivery Driver upon order placement.
- Generate a dynamic bill/receipt.

Design Patterns Used:

- **Composite Pattern** → For Menu structure (Main Menu → Categories → Food Items).
- **Observer Pattern** → For auto-triggering notifications to Restaurant and Rider.
- **Command Pattern** → For handling the "Place Order" action securely.
- **Facade Pattern** → For a simplified User Interface (hiding complex backend logic).

This project fully implements Object-Oriented concepts and SCD design principles.

---

2. Design Pattern Usage

2.1 Composite Pattern

Problem It Solves: The menu has a hierarchical structure (e.g., Main Menu contains Fast Food, Fast Food contains Burgers). Managing these nested levels individually is difficult and messy.

Implementation: We used the Composite pattern to treat individual Food Items and Menu Categories uniformly.

- `MenuCategory` can hold other items or categories.
- `FoodItem` is a leaf node (end item).

Class-Level Structure:

- `MenuComponent` (Interface)
- `MenuCategory` (Composite)
- `FoodItem` (Leaf)

UML (Text Format): `MenuCategory` → contains list of `[MenuComponent]`

2.2 Observer Pattern

Problem It Solves: When a user places an order, the Restaurant needs to start cooking, and the Delivery Driver needs to get ready. Hard-coding these calls makes the system rigid.

Implementation: We attached `Restaurant` and `DeliveryDriver` as subscribers (observers) to the Order. When an order is placed, they are automatically notified.

Class Structure:

- `Observer` (Interface)
- `Restaurant` (Concrete Observer)
- `DeliveryDriver` (Concrete Observer)
- `Order` (Subject - notifies observers)

2.3 Command Pattern

Problem It Solves: We needed to decouple the User (who clicks the button) from the Order Processing logic. This allows us to queue or log orders easily.

Implementation: We encapsulated the request as a `PlaceOrderCommand` object. The `Waiter` class takes this command and executes it.

Class Structure:

- `Command` (Interface)
- `PlaceOrderCommand` (Concrete Command)
- `Waiter` (Invoker)

2.4 Facade Pattern

Problem It Solves: The backend system has many complex parts (Menu, Observers, Commands, Waiter). The user (client) shouldn't see this complexity.

Implementation: We created a `FoodSystemFacade` class. It provides simple methods like `browse_menu()` and `place_order()` so the user doesn't have to worry about the internal code.

Class Structure:

- `FoodSystemFacade` (The simple interface for the user)

---

3. Implementation Details

3.1 Technologies Used

Language:

- Python 3.x (Core Logic)

Tools:

- VS Code (Editor)
- Git & GitHub (Version Control)

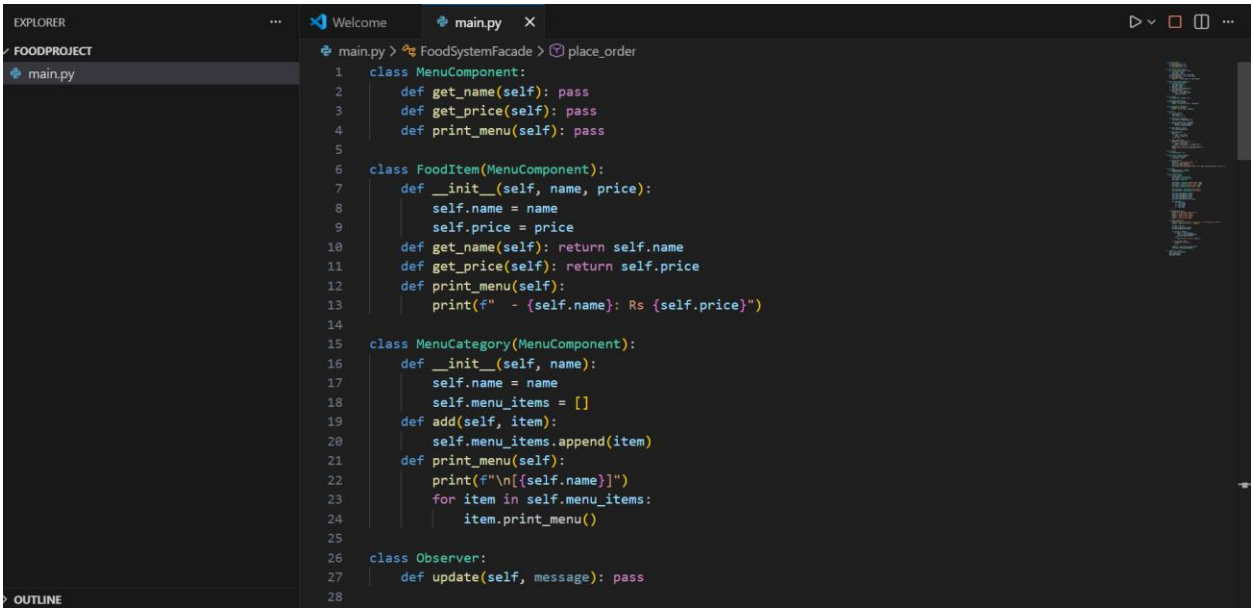
Environment:

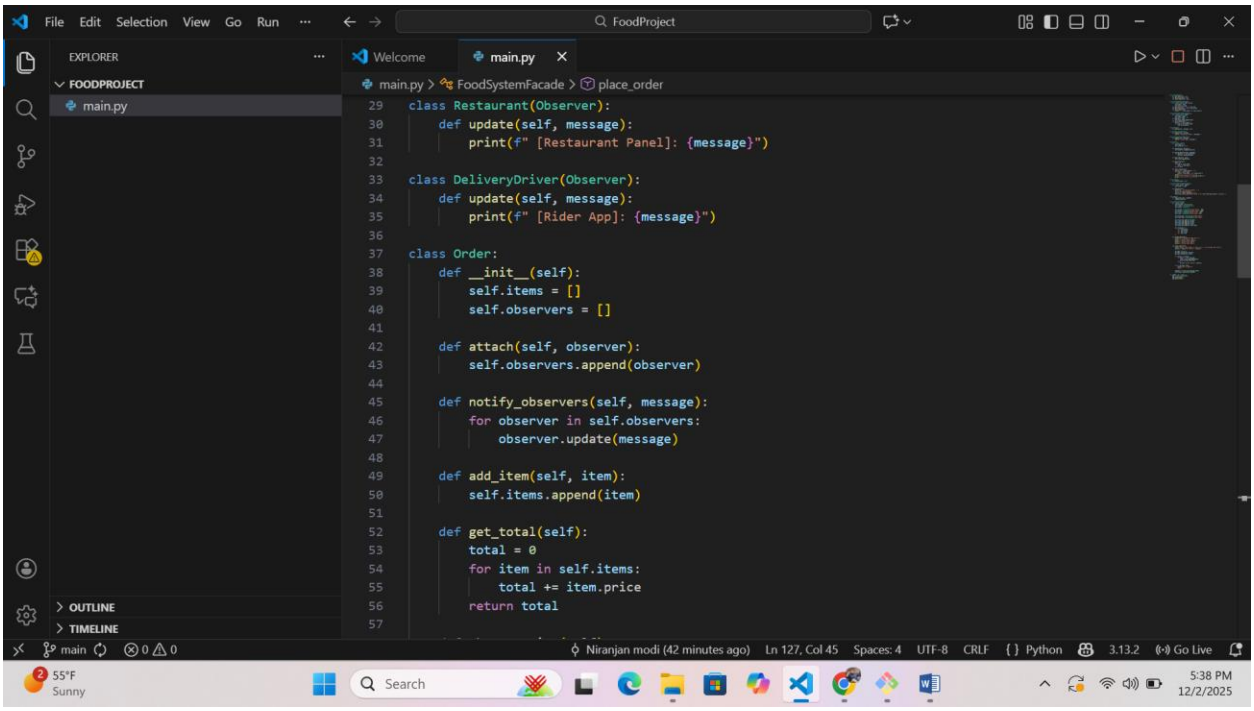
- Command Line Interface (CLI) / Terminal output for simulation.
- No external database was used (In-memory objects were used for simplicity and speed).

---

4. Screenshots / Output

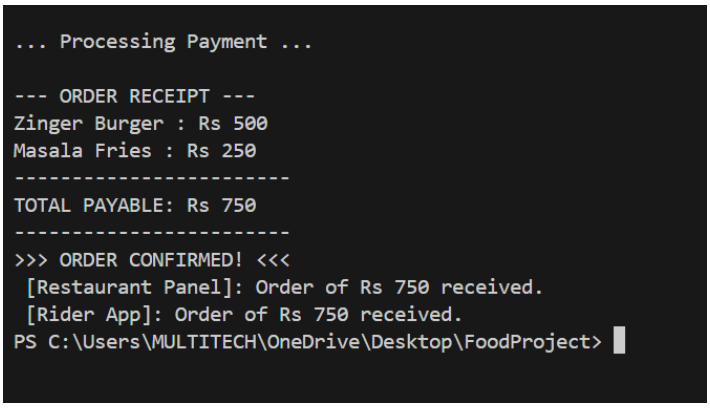
Screenshot 1: Main Menu Display





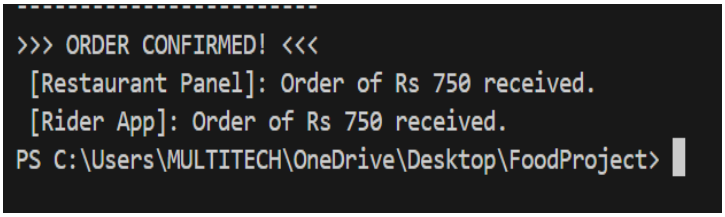

```
PS C:\Users\MULTITECH\OneDrive\Desktop\FoodProject> PYTHON MAIN.PY

=== WELCOME TO FOOD APP ===
1. Zinger Burger (500)
2. Chicken Pizza (1200)
3. Masala Fries (250)
4. Chilled Coke (100)

Enter food numbers to order (e.g., 1 3 for Burger and Fries):
Your Choice: S
```

Screenshot 2: Order Placement & Receipt



System generating the final Bill.

Screenshot 3: Notifications (Observer Pattern)



---

5. Problems Faced & Solutions

Problem 1: Handling Nested Menu Structures

- **Solution:** Implemented the **Composite Pattern** to allow categories to contain other categories or items seamlessly.

Problem 2: Notifying multiple parties (Kitchen, Driver) without tight coupling

- **Solution:** Used the **Observer Pattern**. The Order object just says "Notify", and all subscribers get the message automatically.

Problem 3: The `main` code was becoming too messy and complex

- **Solution:** Implemented the **Facade Pattern**. We moved all setup code into `FoodSystemFacade`, making the main execution file very clean (only 3 lines of code).

Problem 4: Decoupling the Order Action

- **Solution:** Used the **Command Pattern** (`PlaceOrderCommand`) to separate the action of ordering from the execution logic.

---

6. Conclusion

This project successfully demonstrates how **Software Design Patterns** improve system structure, maintainability, and flexibility.

By using **Composite, Command, Observer, and Facade Patterns**, the system became modular and easy to extend. The backend simulation effectively mimics a real-world food ordering scenario, fulfilling all the requirements of the SCD Lab Project.

Repository:

https://github.com/Niranjan780896/Food-order-system