

# **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:

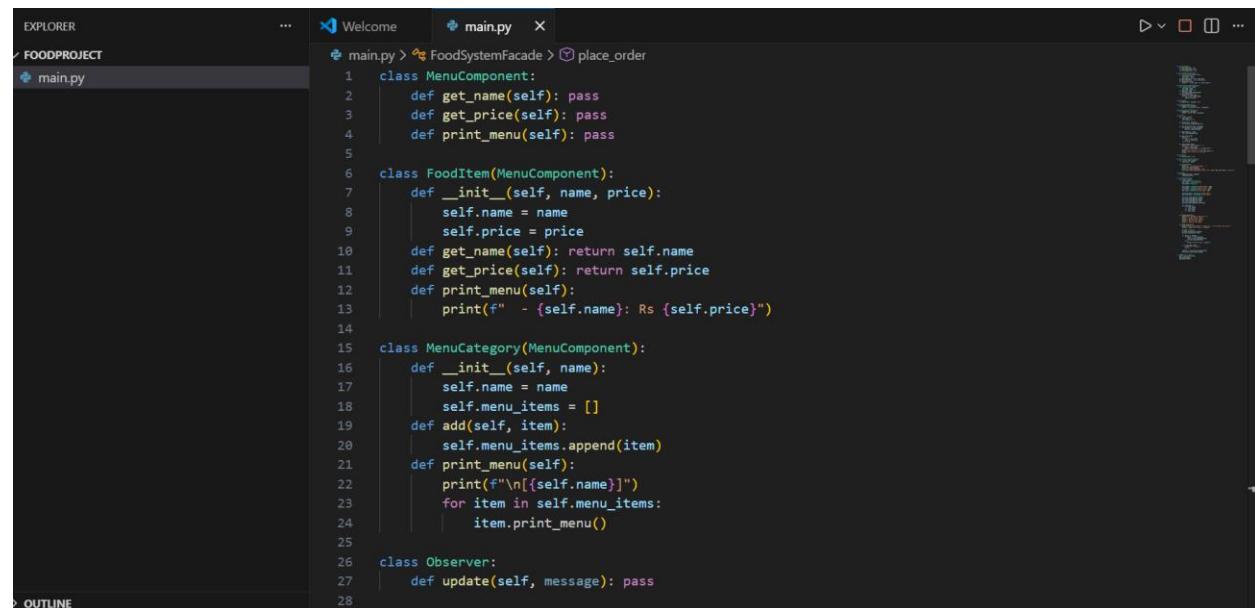
- VS Code (Editor)
- Git & GitHub (Version Control)

#### 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



The screenshot shows the VS Code interface with the main.py file open in the editor. The code implements the Visitor design pattern for a food system menu. It defines four classes: MenuComponent, MenuItem, MenuCategory, and Observer. The MenuComponent class has three abstract methods: get\_name, get\_price, and print\_menu. The MenuItem and MenuCategory classes inherit from MenuComponent and implement these methods. The MenuItem class has \_\_init\_\_ and print\_menu methods. The MenuCategory class has \_\_init\_\_, add, and print\_menu methods. The Observer class has update. The code uses f-strings for printing menu items.

```
EXPLORER          ...  Welcome  main.py  ...
FOODPROJECT
main.py

main.py > FoodSystemFacade > place_order

1  class MenuComponent:
2      def get_name(self): pass
3      def get_price(self): pass
4      def print_menu(self): pass
5
6  class MenuItem(MenuComponent):
7      def __init__(self, name, price):
8          self.name = name
9          self.price = price
10     def get_name(self): return self.name
11     def get_price(self): return self.price
12     def print_menu(self):
13         print(f" - {self.name}: Rs {self.price}")
14
15 class MenuCategory(MenuComponent):
16     def __init__(self, name):
17         self.name = name
18         self.menu_items = []
19     def add(self, item):
20         self.menu_items.append(item)
21     def print_menu(self):
22         print(f"\n[{self.name}]")
23         for item in self.menu_items:
24             item.print_menu()
25
26 class Observer:
27     def update(self, message): pass
```

The screenshot shows a code editor interface with a dark theme. The left sidebar has icons for file operations like Open, Save, and Run, along with sections for Explorer, Welcome, and Outline. The Explorer section shows a project named 'FOODPROJECT' with a file 'main.py'. The main editor area displays the following Python code:

```
29 class Restaurant(Observer):
30     def update(self, message):
31         print(f" [Restaurant Panel]: {message}")
32
33 class DeliveryDriver(Observer):
34     def update(self, message):
35         print(f" [Rider App]: {message}")
36
37 class Order:
38     def __init__(self):
39         self.items = []
40         self.observers = []
41
42     def attach(self, observer):
43         self.observers.append(observer)
44
45     def notify_observers(self, message):
46         for observer in self.observers:
47             observer.update(message)
48
49     def add_item(self, item):
50         self.items.append(item)
51
52     def get_total(self):
53         total = 0
54         for item in self.items:
55             total += item.price
56
57         return total
```

The status bar at the bottom shows the following information: Nirajan modi (42 minutes ago), Ln 127, Col 45, Spaces: 4, UTF-8, CRLF, Python 3.13.2, Go Live, 5:38 PM, 12/2/2025.

The screenshot shows a terminal window with a black background and white text. The command 'PYTHON MAIN.PY' is entered, followed by the output of the script:

```
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

```
... Processing Payment ...

--- ORDER RECEIPT ---
Zinger Burger : Rs 500
Masala Fries : Rs 250
-----
TOTAL PAYABLE: Rs 750
-----
>>> ORDER CONFIRMED! <<<
[Restaurant Panel]: Order of Rs 750 received.
[Rider App]: Order of Rs 750 received.
PS C:\Users\MULTITECH\OneDrive\Desktop\FoodProject>
```

*System generating the final Bill.*

## Screenshot 3: Notifications (Observer Pattern)

```
>>> ORDER CONFIRMED! <<<
[Restaurant Panel]: Order of Rs 750 received.
[Rider App]: Order of Rs 750 received.
PS C:\Users\MULTITECH\OneDrive\Desktop\FoodProject>
```

---

## 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>