1 Group Members

Baidy Hu, Jarvin Zhang, Haotian Jiang, Xiangjin Kong, Yusheng He.

Members	Tasks
Baidy Hu	Order calculation, Class and data structure design, Code testing
Jarvin Zhang	Menu design and data loading, Code testing, Report writing
Haotian Jiang	Discount manager, Class diagram drawing, Report writing
Xiangjin Kong	GUI design, Orders and reports generation, Code integration
Yusheng He	Discount calculation, Exceptions dealing, Report writing

Table 1: Task Allocation

2 Link to Repository

https://github.com/YushengHe-h/Advanced-Software-Engineering.git

3 Program Compliance Declaration

This program meets the functional specification requirements.

3.1 Function Coverage

1. Loads menu (menu.csv) and order (orders.csv) files, supporting product classification (coffee, dessert, other) and unique ID verification.

2. The GUI interface supports multiple product selection, discount calculation (20 percent discount for 1 coffee and 2 desserts), and order details display.

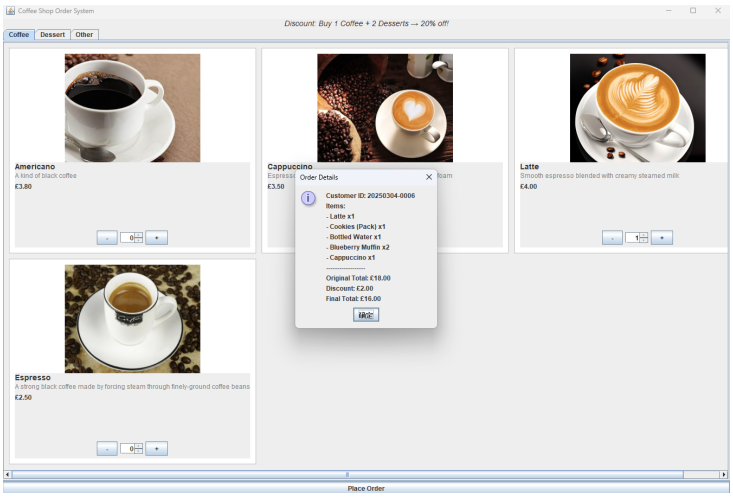

Figure 1: GUI

3. Generates a sales report (report.txt) upon exit to track product sales and total revenue.

```
=== Coffee Shop Sales Report ===
Item ID Item Name   Quantity Sold
COF001  A strong black coffee made by forcing steam through finely-ground coffee beans  4
COF002  Espresso with equal parts steamed milk and velvety milk foam    1
COF003  Smooth espresso blended with creamy steamed milk    2
COF004  A kind of black coffee  3
DES001  Moist chocolate layers with ganache frosting    5
DES002  Fluffy muffin bursting with fresh blueberries   2
DES003  Creamy New York-style cheesecake on a graham cracker crust  4
DES004  Classic pie with cinnamon-spiced apple filling  1
DES005  Buttery French-style flaky croissant    2
OTH001  Natural spring water in a 500ml bottle  1
OTH002  Crispy salted potato chips (100g pack)  4
OTH003  Assorted chocolate chip and oatmeal cookies (6 pieces)  1
OTH004  Fresh organic Fuji apple    2
---------------------------------
Total Revenue: £88.94
```

Figure 2: Sample Report

4 UML Diagram

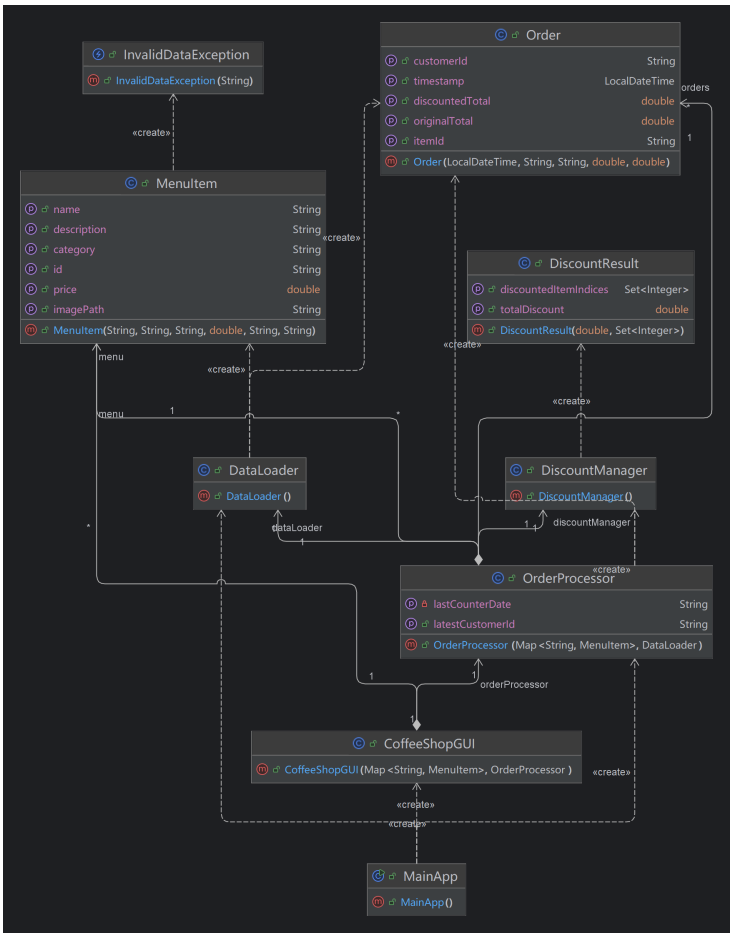

Figure 3: UML Diagram

5 Selection and Reasons for Data Structures

5.1 Map<String, MenuItem>

Used in `DataLoader` to store menu items loaded from CSV files. Key: Unique identifier of a menu item (e.g., COF001), of type String. Value: `MenuItem` object containing product details (name, description, price, category, etc.).

Advantages:

- Fast search – Retrieve product details in O(1) time complexity using menu item ID.
- Ensures uniqueness – The keys in HashMap are unique, preventing duplicate menu item IDs.
- Efficient lookup – Ideal for searching menu items based on unique IDs.

5.2 ArrayList<Order>

Used in `OrderProcessor` to store all order records. New `Order` objects are dynamically added to this list when processing new orders. When generating reports, the list is iterated to calculate total sales.

Advantages:

- Simple and logically clear – Provides intuitive order storage and retrieval.
- Supports dynamic expansion – Can handle an increasing number of orders efficiently.
- Fast index-based access – Accessing elements by index is O(1), making it efficient for order lookups.

5.3 Map<MenuItem, JSpinner>

Used in `CoffeeShopGUI` to link menu items with their corresponding quantities. Facilitates dynamic selection of product quantities in the GUI.

Advantages:

- Fast lookup – The time complexity is O(1), making it efficient for high-frequency operations.
- Simplifies order processing – Directly associates selected products with their quantities, reducing complexity.
- Enhances code readability – The map structure makes GUI integration straightforward.

6 Functional Decision Explanation

6.1 Identifier Rules

(Here you can describe the identifier rules if needed)

6.2 Discount Rules

The discount rule designed by our group is that when a customer orders a cup of coffee and two desserts, a discount of 20% will be triggered. The discount does not take effect until this combination is reached.

To achieve this goal, we designed a Java program called `DiscountManager` to implement discounts. The `calculateDiscount()` method is the core function that receives a list of products and returns the discount result.

```
List<Integer> coffeeIndices = new ArrayList<>();
List<Integer> dessertIndices = new ArrayList<>();
for (int i = 0; i < items.size(); i++) {
    MenuItem item = items.get(i);
    String category = item.getCategory().toLowerCase();
    if ("coffee".equals(category)) {
        coffeeIndices.add(i);
    } else if ("dessert".equals(category)) {
        dessertIndices.add(i);
    }
}
```

Figure 4: Discount Calculation Logic

The purpose of this code is to use a **for loop** to traverse all products and store coffee and desserts separately into two designed indexes by category.

Here, we use the **double pointer pairing method**, where:

- `coffeePointer` traverses the coffee index.
- `dessertPointer` traverses the dessert index.

```
Set<Integer> discountedIndices = new HashSet<>();
double discountAmount = 0.0;
int coffeePointer = 0;
int dessertPointer = 0;
```

Figure 5: Discount Calculation Logic

This code shows how our program processes a discount:

- When **one coffee and two desserts** are successfully paired, **20% of the total price** of the combination is stored in `discountAmount`.
- The index of discounted products is stored in `discountedIndicators`.
- For every coffee processed, `coffeePointer` increases by 1.
- For every two desserts processed, `dessertPointer` increases by 2.

The termination condition is also a loop condition in the `while` loop. When all the coffee has been used up or the remaining desserts cannot complete a valid combination, the pairing process stops, and the discount result is outputted to `DiscountResult`.

`DiscountResult` is a small class used to receive the data returned by `DiscountManager` and provide getter methods for external data access.

```
// Apply discount: for each coffee, pair with two desserts
while (coffeePointer < coffeeIndices.size() && dessertPointer + 1 < dessertIndices.size()) {
    int coffeeIndex = coffeeIndices.get(coffeePointer);
    int dessert1Index = dessertIndices.get(dessertPointer);
    int dessert2Index = dessertIndices.get(dessertPointer + 1);

    discountedIndices.add(coffeeIndex);
    discountedIndices.add(dessert1Index);
    discountedIndices.add(dessert2Index);

    MenuItem coffee = items.get(coffeeIndex);
    MenuItem dessert1 = items.get(dessert1Index);
    MenuItem dessert2 = items.get(dessert2Index);
    discountAmount += (coffee.getPrice() + dessert1.getPrice() + dessert2.getPrice()) * 0.2;

    coffeePointer++;
    dessertPointer += 2;
}
```

Figure 6: Discount Calculation Logic

Through this implementation, we can accurately apply the **"one coffee + two desserts"** discount without errors when purchasing additional products.

6.3 Order Processing

Order processing is the core function of our program, which is implemented in the `OrderProcessor` Java file. It integrates essential functions for coffee shop operations, including:

- Generating customer IDs
- Processing discounts

- Saving orders
- Calculating total amounts
- Generating sales reports

`loadOrderCounter()` and `saveOrderCounter(int counter)` manage the order counters:

- **loadOrderCounter()** reads the current date and counter value.
- **saveOrderCounter()** updates and saves the counter value to a file.

```
public class DiscountResult {
    private final double totalDiscount;
    private final Set<Integer> discountedItemIndices;

    public DiscountResult(double totalDiscount, Set<Integer> indices) {
        this.totalDiscount = totalDiscount;
        this.discountedItemIndices = indices;
    }

    public double getTotalDiscount() { return totalDiscount; }
    public Set<Integer> getDiscountedItemIndices() { return discountedItemIndices; }
}
```

Figure 7: Discount Calculation Logic

`generateCustomerId()` is responsible for generating a unique customer ID, starting from `0001` every day.

```
private String generateCustomerId() {
    String currentDate = LocalDate.now().format(DATE_FORMAT);
    if (!currentDate.equals(getLastCounterDate())) {
        orderCounter = 0;
    }
    String counterSuffix = String.format(format:"%04d", ++orderCounter);
    saveOrderCounter(orderCounter);
    return currentDate + "-" + counterSuffix;
}
```

Figure 8: Discount Calculation Logic

The core order processing logic:

- A unique **customer ID** is created.
- Discounts are applied based on eligible product combinations.
- If a product is in the discount list, the price is reduced to **80%** of its original price.
- An order object is created, saved to a file, and updates sales statistics.

```java
public void processOrder(List<MenuItem> items) {
    String customerId = generateCustomerId();
    DiscountResult discountResult = discountManager.calculateDiscount(items);
    Set<Integer> discountedIndices = discountResult.getDiscountedItemIndices();

    for (int i = 0; i < items.size(); i++) {
        MenuItem item = items.get(i);
        double originalTotal = item.getPrice();
        double discountedTotal = originalTotal;

        if (discountedIndices.contains(i)) {
            discountedTotal = originalTotal * 0.8;
        }

        Order order = new Order(
                LocalDateTime.now(),
                customerId,
                item.getId(),
                originalTotal,
                discountedTotal
        );
        try {
            dataLoader.saveOrder(order, ordersFile);
        } catch (IOException e) {
            System.err.println("Failed to save order: " + e.getMessage());
        }

        sales.put(item.getId(), sales.getOrDefault(item.getId(), defaultValue:0) + 1);
        orders.add(order);
    }
}
```

Figure 9: Discount Calculation Logic

To compute the total price of products:

- The original total price is calculated.
- Discounts are applied to calculate the final total price.

```java
public double calculateOriginalTotal(List<MenuItem> items) {
    return items.stream().mapToDouble(MenuItem::getPrice).sum();
}

/**
 * Calculates the total discount for a list of items.
 * @param items The items to be checked for discount.
 * @return The discount amount.
 */
public double calculateDiscount(List<MenuItem> items) {
    return discountManager.calculateDiscount(items).getTotalDiscount();
}
```

Figure 10: Discount Calculation Logic

Finally, the program generates a report containing:

- The total revenue
- The number of orders processed
- The total amount of discounts applied

```
public void generateReport() throws IOException {
    double totalRevenue = orders.stream()
            .mapToDouble(Order::getDiscountedTotal)
            .sum();
    dataLoader.saveReport(sales, menu, totalRevenue, reportFile);
}
```

Figure 11: Discount Calculation Logic

This implementation ensures an efficient and structured order processing system while maintaining accurate discount application and sales tracking.

7 Testing

7.1 Overview of Testing Approach

JUnit tests were conducted to ensure the correctness of discount application and order processing.

- Verifies correct discount calculation based on order combinations.
- Ensures order ID generation functions correctly across multiple days.
- Validates exception handling to prevent invalid data from crashing the program.

7.2 JUnit Testing for DiscountManager

`DiscountManagerTest` verifies various discount scenarios.

Test Method	**Scenario Tested**	**Expected Outcome**
testEmptyList()	Empty order list	No discount applied
testOnlyCoffee()	Order contains only coffee	No discount applied
testOnlyDesserts()	Order contains only desserts	No discount applied
testOneCoffeeTwoDesserts()	Order with **1 coffee + 2 desserts**	20% discount applied
testTwoCoffeesFourDesserts()	Order with **2 coffees + 4 desserts**	20% discount applied
testOneCoffeeThreeDesserts()	Order with **1 coffee + 3 desserts**	Extra dessert ignored, correct discount applied
testInsufficientDesserts()	Order has **not enough desserts** for a discount	No discount applied
testMixedCategories()	Order contains other categories (e.g., sandwiches)	Discounts apply only to coffee and desserts

Table 2: JUnit Testing for DiscountManager

7.3 JUnit Testing for OrderProcessor

Tests were conducted to verify order processing, customer ID generation, and discount application.

Test Method	**Scenario Tested**	**Expected Outcome**
generateCustomerId_ShouldResetCounterNextDay()	Customer ID resets daily	ID starts at `0001` every day
calculateOriginalTotal_ShouldSumAllItems()	Ensure total price calculation	Correct total price computed
processOrder_ShouldApplyDiscountCorrectly()	Order processed with discounts	Correct discount applied
testCustomerIdGeneration()	Unique ID for each order	IDs are unique and sequential

Table 3: JUnit Testing for OrderProcessor

7.4 Exception Handling

The program includes custom exceptions to handle invalid input cases.

Exceptions in DiscountManager

- Scenario: Order contains invalid menu items.
- Exception: `InvalidDataException`
- Purpose: Ensures only valid coffee and desserts contribute to discounts.

Exceptions in OrderProcessor

- Scenario 1: Order references a non-existent menu item.
- Scenario 2: Order data is empty or corrupt.
- Exception: `InvalidDataException`
- Purpose: Prevents system crashes due to invalid orders.

7.5 Key Development Issues and Solutions

During development, several critical issues were identified and resolved.

7.5.1 1. Issue: GUI Quantity Selector Allowed Negative Numbers

Symptom: Users could decrease item quantity below zero using the minus button.

Root Cause:

- The spinner component (`JSpinner`) did not have a minimum value restriction.
- The decrement button directly reduced the value without checking the lower bound.

Solution:

- **Set a minimum value:** Define the spinner model with `(0, 0, 10, 1)` to prevent negative values.
- **Validate button actions:** Ensure that the decrement button only decreases the value if greater than zero.
- **Filter invalid input:** Implement a numeric input validator to prevent non-numeric entries.

Simplified Pseudocode:

```
if (spinner_value > 0) then
    decrement spinner_value by 1
end if
```

7.5.2 2. Issue: Customer ID Reset Failure Across Days

Symptom: Customer ID counter did not reset at midnight, leading to incorrect numbering.

Root Cause:

- The order counter file only stored a number without tracking the date.
- The program had no way to detect whether a new day had started.

Solution:

- **Modify file format:** Store `"date-counter"` format (e.g., `20240301-0005`).
- **Check and reset counter:** At program startup:
  - Read the stored date from the file.
  - If the date does not match the current date, reset the counter to 1 and update the file.

Simplified Pseudocode:

```
read last_saved_date from file
if last_saved_date is different from today then
    reset customer_counter to 1
    update last_saved_date
end if
```

7.5.3 3. Issue: Duplicate Discount Calculation

Symptom: The same product received multiple discounts incorrectly.

Root Cause:

- Discount logic relied on product ID matching, causing discounts to be applied multiple times.

Solution:

- **Index-based discounting:** Use item indices in the order list rather than product IDs.
- **Pairing logic:** Ensure each coffee-dessert combination is counted only once.

Simplified Pseudocode:

```
group items into coffee_list and dessert_list
initialize empty set discounted_items
while (more coffee and at least two desserts exist) do
    select one coffee and two desserts
    add them to discounted_items set
end while
```

7.6 Summary of Testing Outcomes

- All JUnit tests passed successfully, confirming system reliability.
- Edge cases were handled efficiently, including mixed categories and insufficient discounts.
- Exception handling ensured invalid data did not disrupt order processing.

The tests confirmed that `DiscountManager` and `OrderProcessor` work correctly and robustly, ensuring accurate discount application and order tracking.