

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.

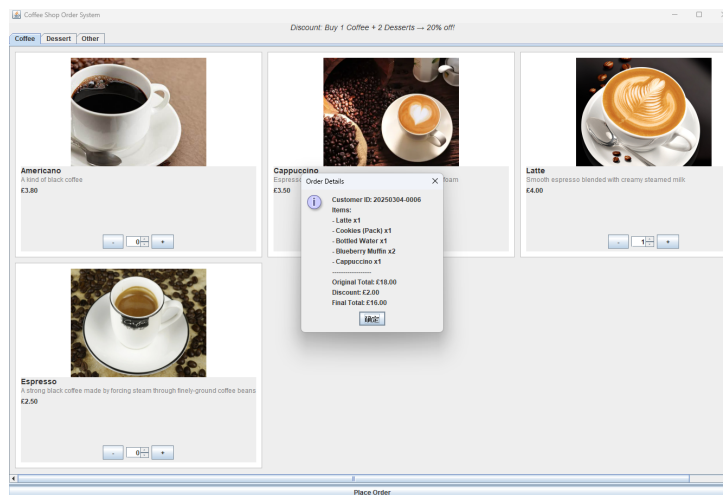


Figure 1: GUI

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

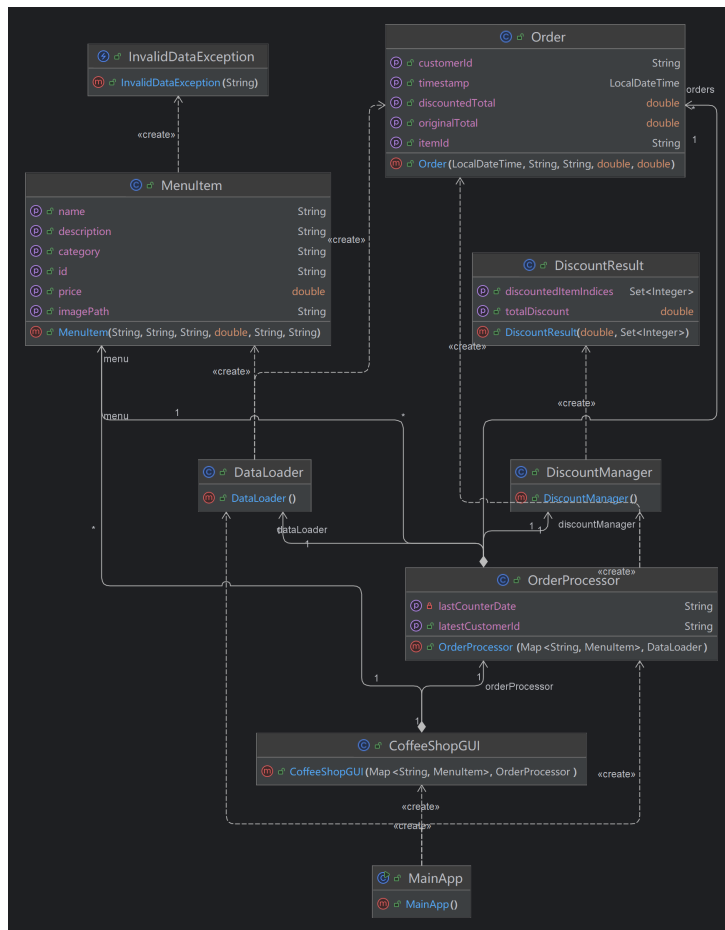


Figure 3: UML Diagram

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.

```

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.

```

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.

Group report - stage 2

Team Member Contributions

Erzhizhi Hu	Implemented the online order processing functionality, designed the customer generation and queue management workflow, and tested the overall simulation runtime behavior.
Haotian Jiang	Developed the control panel functionality, implemented the EventLogger for event logging, exported the project as a JAR executable, and contributed to the writing of the final report.
Jiawen Zhang	Developed the data loading and report generation modules, documented the agile development process, and contributed to the writing of the project report and user guide.
Xiangjin Kong	Implemented the server and barista threads, designed inter-thread communication mechanisms, solved synchronization issues, and was responsible for code integration across the project.
Yusheng He	Designed and implemented the GUI interface, extended the MenuItem and Order classes, created the UML class diagrams for the project, and contributed to the writing of the report.

1. Overview and Implementation of System Functions

This project implements a multi-threaded system that simulates the operation of a coffee shop, successfully fulfilling all core requirements outlined in Stage 2, along with several extended features. The system is capable of simulating customer generation, queuing, order processing by servers, beverage preparation by baristas, and real-time status updates through a graphical user interface (GUI), forming a complete simulation process.

Core Functionality:

The system supports multiple server threads working concurrently, all drawing customers from a shared queue. Customers (along with their orders) are retrieved by a dedicated thread and added to the end of the waiting queue. When a customer reaches the front of the queue and a server becomes available, the customer is assigned to that server for processing. Upon startup, the system loads pre-existing order data from the *orders.csv* file generated in Stage 1 and enqueues these orders in bulk for processing by the servers in sequence. Once all queues are empty and all server and barista threads are idle, the simulation terminates automatically, generates a sales report, and exits the program safely.

Throughout the simulation, the GUI displays the list of waiting customers and their order details, as well as the current status of each server and barista (e.g., idle, busy, processing an order). Users can adjust the simulation speed via the GUI to clearly observe changes in

system states. Additionally, the system features an event logging mechanism: all significant events—such as customer queuing, order assignment, and order completion—are logged and output to a file upon the conclusion of the simulation.

Extended Features:

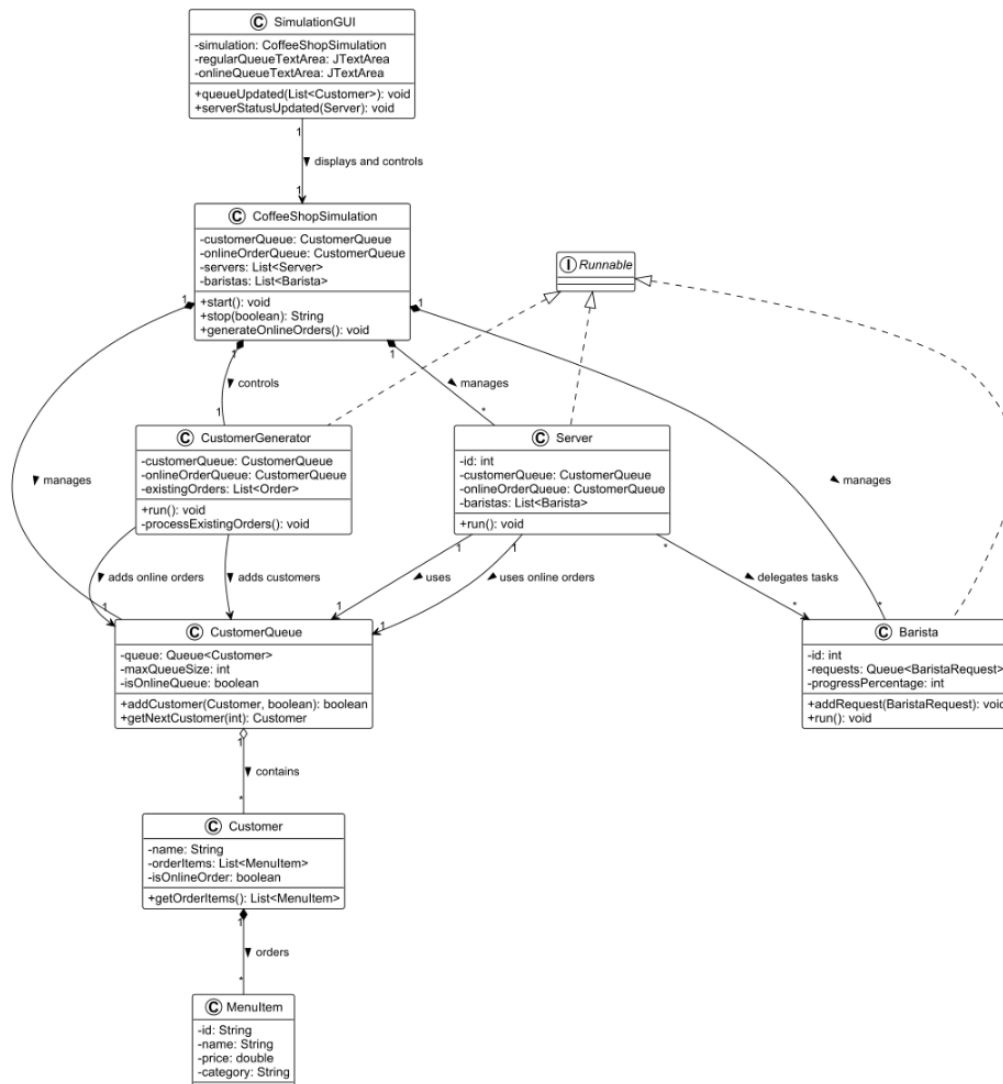
Beyond the fundamental requirements, the system includes a series of enhanced features to improve realism and flexibility. For instance, a simulation speed control slider allows users to dynamically accelerate or decelerate the simulation during runtime. The system also supports the dynamic addition and removal of server and barista threads: users can click GUI buttons to add new threads or remove existing ones, simulating varying staffing configurations.

To prevent resource issues caused by uncontrolled queue growth, a maximum queue size (e.g., 10 customers) is enforced. The system strictly adheres to the producer-consumer model in managing the queue: when the queue is full, the customer generator thread pauses and waits until a server removes a customer and makes space available. Furthermore, the system introduces an independent channel for online orders, utilizing a separate priority queue. When users click the "Generate Online Orders" button in the GUI, a batch of online orders is created; these are prioritized over regular queued customers and are picked up first by available servers. The system also includes an auto-stop option that determines whether it automatically shuts down once all orders have been processed, and a `GeneratorCustomer` option that allows random new customers to be generated after the system finishes reading orders from the *orders.csv* file.

Finally, the system introduces barista threads to simulate beverage preparation. While processing an order, a server delegates drink-making tasks to an independent barista thread. Upon completion, the barista notifies the server via a callback mechanism. All these processes are clearly visualized in the GUI, allowing users to observe the entire workflow in real time—from customer arrival and order placement to beverage preparation and order completion.

Overall, the system features comprehensive modules with tightly integrated interactions, fully satisfying the functional requirements for the coffee shop simulation project.

2.UML Diagram



3. Detailed Information on Agile Development Process

In Stage 2 of this project, we fully adopted agile development methods. The team followed an iterative approach to development, setting clear objectives for each iteration and continuously refining the system's functionality. We conducted three major iterations, each resulting in a well-defined, working software product. This methodology enabled the team to effectively manage project progress and quality, and to respond quickly to changing requirements.

(1) First Iteration (Version v0) – Implementation of Basic Functionality

Objectives:

Establish a basic thread framework to verify the producer-consumer relationship between servers and the customer generator.

Implement an initial GUI interface to provide basic status display.

Implementation Process:

In this iteration, we constructed a multi-threaded architecture and completed the initial design and implementation of the customer generator thread and server threads. We implemented a simple, thread-safe customer queue from which server threads could retrieve customers and process their orders. Simultaneously, a basic GUI was developed to display the number of customers in the queue and their basic statuses in real time.

Outcomes:

Completed the initial thread architecture and the implementation of a safe customer queue.

The GUI was able to display the customer queue status initially.

Retrospective and Improvements:

Through daily stand-up meetings and code reviews, we discovered issues such as unclear display of thread states and an unfair task distribution strategy among server threads. Some threads were overloaded while others were underutilized. This prompted us to introduce more comprehensive status monitoring and thread load balancing mechanisms in the next iteration.

(2) Second Iteration (Version v1) – Design Pattern Integration and GUI Enhancements

Objectives:

Introduce the Observer pattern to improve real-time GUI status updates.

Refine the Singleton pattern for the event logger (EventLogger).

Improve fairness in task distribution among server threads.

Implementation Process:

In this iteration, the team applied the Observer pattern to clearly separate state update responsibilities from the model to the view, allowing the GUI to update automatically in response to model changes. We also completed the Singleton implementation of the EventLogger, ensuring unified logging of system events, which facilitated debugging and analysis. Furthermore, we redesigned the task acquisition strategy of server threads, introducing polling and randomized selection to significantly improve load balance across threads.

Outcomes:

GUI state updates became significantly more stable and responsive.

Unified management of event logging simplified error tracing.

Thread workload was more evenly distributed.

Retrospective and Improvements:

In the iteration retrospective, the team recognized two key shortcomings: the lack of a priority mechanism for handling online orders, and the inability to dynamically add or remove servers and baristas during runtime. As a result, we decided to focus on implementing online order priority handling and dynamic thread management in the next iteration, along with further system performance optimization.

(3) Third Iteration (Version v2) – Advanced Feature Extensions and Performance Optimization

Objectives:

Implement a priority handling mechanism for online orders.

Introduce dynamic thread management for servers and baristas.

Add simulation speed adjustment functionality to enhance user experience.

Conduct comprehensive thread synchronization and performance optimization to prevent deadlocks and blocking.

Implementation Process:

In this iteration, the team added a separate priority queue for online orders and implemented a mechanism allowing server threads to prioritize and preemptively handle online orders, significantly improving their processing efficiency. We also implemented dynamic thread management, enabling users to add or remove server and barista threads in real time via the GUI, thereby simulating different staffing configurations. Additionally, a simulation speed control slider was added, allowing users to adjust the simulation speed as needed.

For performance optimization, we reinforced thread synchronization mechanisms using wait/notify, timeout mechanisms, atomic variables, and callback strategies. These improvements significantly enhanced system stability under high concurrency and effectively eliminated potential deadlock risks.

Outcomes:

Online order prioritization significantly improved user responsiveness.

Dynamic thread management enabled flexible runtime control.

Simulation speed adjustment enhanced the interactivity and observability of the system.

Overall system stability and concurrency performance were greatly improved.

Retrospective and Summary:

During the retrospective, the team reviewed the development experience and concluded that agile development through iterative delivery significantly improved our ability to adapt. Daily stand-up meetings enhanced team collaboration in solving complex problems, while continuous integration helped avoid integration conflicts and risks during code merging.

4. Thread Model and Synchronization Mechanism Analysis

(1) Thread Role Allocation

This system adopts a multi-threaded concurrent architecture, where different threads assume distinct responsibilities and collaboratively simulate the concurrent business processes of a coffee shop:

Main Thread (Event Dispatch Thread):

Responsible for starting the simulation, creating the GUI, and maintaining the event loop. It handles user interactions (such as button clicks to adjust parameters) and invokes corresponding control logic. At the same time, it ensures all GUI updates are executed on the event dispatch thread to avoid unsafe direct interactions between background threads and UI components.

Customer Generator Thread:

An independent producer thread that continuously generates new customers and their orders at fixed intervals. At startup, it reads the initial order list provided from Stage 1 and enqueues those customers in bulk. During the simulation, it randomly creates customers at intervals, simulating the real-world arrival of customers at a coffee shop.

Server Threads:

Each instance of Server corresponds to one server thread. Multiple servers concurrently process customer orders from the queue. Each server thread repeatedly retrieves the next customer: it first checks the online order queue and serves online customers with priority; if that queue is empty, it fetches from the regular queue. If both queues are empty, the thread waits briefly and then retries. The order processing procedure includes recording the order, computing price and discounts, delegating drink-making tasks to barista threads, and waiting for beverage preparation to finish. After serving one customer, the server thread loops back to fetch the next one, continuing until a stop signal is received.

Barista Threads:

Each instance of Barista represents a barista thread responsible for asynchronously handling beverage-making tasks assigned by servers. Each barista thread maintains a task queue to which servers submit requests via the `addRequest()` method. While running, the thread checks the request queue: if there are tasks, it processes them; otherwise, it enters a waiting state until new requests arrive. Upon completing a beverage, the barista triggers the callback function included in the request to notify the corresponding server, enabling it to proceed with further steps.

(2) Thread Synchronization Mechanism

To ensure safe and efficient collaboration among threads, the system implements various synchronization strategies to coordinate access to shared resources and inter-thread communication:

Monitor Locks (Synchronized Methods):

Critical shared data structures are protected with built-in Java locks to ensure thread safety. For instance, the methods `addCustomer()` and `getNextCustomer()` in `CustomerQueue` are declared public synchronized, ensuring that only one thread can access or modify the internal queue at any given time, thus preventing race conditions.

Conditional Wait and Notify (wait/notify):

Used alongside locks to coordinate producer-consumer behavior. When the customer queue is full, the customer generator thread calls `wait()` inside `addCustomer()` to suspend itself. Once a server removes a customer, `notify()` is called within `getNextCustomer()` to awaken waiting producers. This guarantees that the queue will not overflow, and producers will resume only when space becomes available. Similarly, for barista threads, if their task queue is empty, they enter a `wait()` state; when a new task is submitted, `notify()` is used to awaken any waiting barista. This wait/notify mechanism enables smooth coordination between threads and avoids wasteful busy-waiting.

Atomic Variables (AtomicBoolean):

Each major thread (server, barista, customer generator) uses an `AtomicBoolean` flag named `running` to control whether the thread should continue running. When a thread needs to be stopped, its running flag is set to false, and its loop checks this flag to safely exit. The atomic nature ensures visibility and correctness when accessed by multiple threads.

Callbacks and CountdownLatch:

After a server submits a drink-making task to a barista, it must wait for the barista to finish. To avoid long lock-holding or active polling, the system employs callback mechanisms combined with Java's synchronization utilities like `CountDownLatch` for asynchronous waiting. Specifically, the server includes a callback `Runnable` or a `CountDownLatch` in the

BaristaRequest. When the barista finishes the task, the callback is executed or the latch is decremented, allowing the server thread to resume. This approach avoids blocking and polling, ensures the server is properly notified, and maintains logical correctness.

Timeouts and Interrupt Handling:

To enhance robustness, all `wait()` calls specify timeouts or are checked for interruption. For instance, barista threads use `wait(1000)` to limit waiting to 1 second and periodically check their running flag, ensuring responsiveness to termination signals. Similarly, server threads use a custom `waitFor(500)` method to sleep briefly (0.5 seconds) when no customers are available, instead of continuous busy-waiting. These measures effectively prevent deadlocks and indefinite blocking.

GUI Update Synchronization:

Due to Swing's single-thread rule, background threads are not allowed to directly update GUI components. Therefore, all status change notifications from server and barista threads are handled via the Observer pattern in the GUI thread or explicitly passed to the event dispatch thread using `SwingUtilities.invokeLater()`. For example, when a server starts serving or a barista is making a drink, the GUI is notified to refresh its display. This asynchronous update method ensures thread safety and avoids cross-thread UI manipulation issues.

Through this multi-layered synchronization strategy, the system ensures thread safety while enabling efficient communication and collaboration between concurrent components. It avoids resource contention and deadlocks and maintains high responsiveness across all active modules.

5. Explanation of Design Pattern Usage

To enhance the maintainability and scalability of the system, we consciously adopted several classic software design patterns in our system architecture. The primary patterns used are explained as follows:

1. Singleton Pattern:

The event logger (`EventLogger`) in this system is implemented using the Singleton pattern. There exists only one instance of `EventLogger` throughout the application, which is accessed globally via the static factory method `getInstance()`. This pattern ensures unified management of event logging, avoiding inconsistencies caused by multiple logger objects. Additionally, the Singleton provides a simple global access point, making it convenient for components across the system to record events. The `EventLogger` is lazily initialized upon first use, thereby conserving system resources.

2. Observer Pattern:

The Observer pattern is widely used in the system to enable loosely coupled communication between components. When a model object changes state, it notifies its observers (typically the GUI) to update the interface. For example, the CustomerQueue maintains a list of QueueObserver objects. When a customer joins or leaves the queue, the notifyObservers() method is called, triggering each observer's queueUpdated() method and passing a snapshot of the current queue to the GUI. Similarly, the Server and Barista classes define ServerObserver and BaristaObserver interfaces respectively, which are implemented by the GUI. Whenever a server or barista's status changes, it notifies the GUI to update the display accordingly.

Through the Observer pattern, the model and view are decoupled: the model does not need to know any details of the interface, and it simply notifies observers based on predefined conventions. The view, on the other hand, actively subscribes to the model by implementing observer interfaces and receives updates automatically when the model changes. This pattern supports one-to-many notification, allowing the GUI to listen to the status of multiple servers and baristas simultaneously. Observers can also be added or removed dynamically, enabling the system to register or unregister interface components flexibly at runtime.

3. Model-View-Controller (MVC) Architecture:

The overall design adheres to the MVC architecture, separating business logic, interface rendering, and control flow. The Model layer contains core business logic and data structures, such as the simulation main class CoffeeShopSimulation, the CustomerQueue, and entities like Customer, MenuItem, and Order, as well as the background processing threads Server and Barista. The View layer is represented by SimulationGUI and its subcomponents, which are responsible for rendering the graphical interface and managing user interactions. The Controller layer includes various components working together, such as the CustomerGenerator for generating input (customers), the OnlineOrderQueue and CustomerQueue for dispatching tasks, GUI event handlers for translating user actions into model updates, and DataLoader for loading data during initialization.

The benefit of MVC lies in separation of concerns: the view and model interact through observer interfaces without needing to know each other's internal implementation, allowing GUI layout and business logic to evolve independently. Moreover, MVC facilitates team development, enabling one developer to focus on model logic while another designs the interface without conflict. It also improves testability—for example, core logic in the model can be independently tested without launching the GUI.

4. Command Pattern:

In the beverage preparation process handled by baristas, we used the Command pattern to encapsulate drink-making requests into objects for asynchronous execution.

Specifically, we defined an inner class `BaristaRequest`, which includes the `MenuItem` to be prepared and a callback `Runnable`. When a server needs a drink prepared, it creates a `BaristaRequest` object, passing in the menu item and callback. The barista thread executes the request and invokes `request.complete()` upon finishing, which in turn executes the previously provided callback (notifying the server that the drink is ready).

With the Command pattern, the system encapsulates the description of the task (the beverage and the follow-up action) into an object that can be queued and executed parametrically. The server does not need to know the implementation details, only waiting for the callback. This design makes the task handling process more flexible and extensible. For example, it is easy to add support for different types of drink requests in the future. The pattern also facilitates request queuing and logging. The Barista's request queue can hold multiple `BaristaRequest` objects in sequence, each carrying all necessary information for execution—making debugging and tracking more efficient.

5. **Factory Method Pattern:**

The Factory Method pattern is used to encapsulate object creation logic and avoid scattered use of new operations throughout the business code. In this project, we applied this pattern in the `CustomerGenerator` class to create different types of customer objects. For instance, based on the input parameters, the `createCustomer()` method determines whether to generate a regular customer or an online customer. This centralizes customer object creation and simplifies adding special creation logic in the future (e.g., tagging online customers).

Furthermore, the factory method allows subclasses to override the object creation logic, following the open-closed principle. When introducing new types of customers, developers can simply modify the factory method without changing the calling code, thus enhancing extensibility and flexibility.

6. **Producer-Consumer Pattern:**

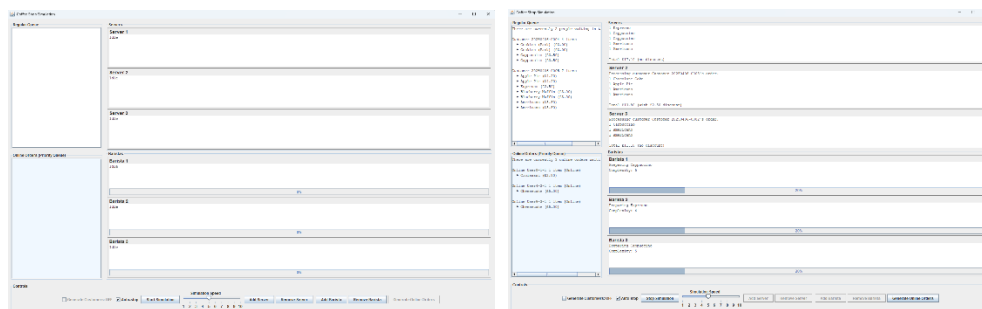
This pattern is extensively applied in the system's queue management. The `CustomerGenerator` (producer) and the `Server` (consumer) form a classic producer-consumer pair: the former continuously produces customers and adds them to a queue, while the latter retrieves them for order processing. They are decoupled through a shared buffer—the `CustomerQueue`—so the producer does not directly interact with the consumer. If the production rate exceeds the consumption rate, the queue builds up; if the queue is full, the producer waits; if the queue is empty, the consumer waits. Our implementation follows this pattern strictly: `CustomerQueue` has a fixed capacity, and when `addCustomer()` detects the queue is full, the producer thread waits. Once a customer is removed, `getNextCustomer()` calls `notify()` to wake

the producer.

This buffering mechanism helps the system automatically balance production and consumption rates, preventing overload. Additionally, the Producer-Consumer pattern decouples customer generation from order processing, allowing them to evolve independently. Apart from the customer queue, the same pattern is applied between server and barista threads: servers produce beverage requests, and baristas consume them. Overall, this pattern is effectively used in the project to ensure coordinated and orderly operation among threads under concurrency.

In summary, the thoughtful application of design patterns significantly improved the modularity, code reusability, and extensibility of the system. By considering these patterns during the design phase, we were able to meet complex requirements at a relatively low cost and maintain code clarity even as functionality expanded.

6. Sample screen shots



7. Comparison of Development Experiences in Stage 1 and Stage 2

Throughout the entire project, our team experienced two distinct development methodologies. Stage 1 adopted a plan-driven development approach, while Stage 2 was conducted using agile development practices. Each approach has its own advantages and disadvantages and brought different experiences and challenges in practice.

(1) Stage 1: Characteristics and Experience of Plan-Driven Development

Stage 1 emphasized complete upfront design and unified planning. Before writing any code, we carried out detailed class structure design, task responsibility allocation, and data structure determination. All functionalities were implemented strictly based on the static requirement specifications provided in the coursework, with minimal adjustments made during development.

Advantages:

A clear architectural blueprint was established early, ensuring unified direction at the beginning of development.

Tasks could be allocated in advance, allowing team members to work relatively independently on their respective modules.

For systems with relatively stable and clearly defined functionality, plan-driven development can lead to higher efficiency.

Issues and Limitations:

It was difficult to respond quickly to changes. Once a design flaw was identified, modifying the system incurred high costs.

There was a lack of iterative feedback. Problems such as incompatible interfaces or poor module integration were often not discovered until the final stages.

Low communication frequency among team members led to incomplete understanding of the overall system architecture for some individuals.

(2) Stage 2: Agile Development Practice and Reflection

In Stage 2, we adopted an agile methodology and proceeded with development in iterations. Each iteration had clearly defined goals and resulted in a functional and testable version of the system. During this phase, we also implemented agile practices such as daily stand-up meetings, pair programming, and continuous integration.

Advantages:

Problems were discovered and resolved more quickly; the system was gradually improved, and directions could be adjusted flexibly.

After each iteration, the system was runnable and testable, which facilitated test-driven development.

Team members developed a better understanding of the overall system structure, and the high communication frequency led to closer collaboration.

Challenges and Responses:

Changes in requirements led to partial redesigns, but we managed to maintain system maintainability through refactoring.

We encountered technical difficulties in thread synchronization and GUI interaction, which were gradually resolved through pair programming and team discussions.

Effective time management was essential. To prevent over-scoping in iterations, we later

adopted time-boxing techniques, setting fixed time limits for each task to improve delivery efficiency.

Summary:

Overall, while plan-driven development is stable and suitable in scenarios where requirements are well-defined and functionality is fixed, agile development proved to be more efficient and adaptable for a project like ours, which involved multithreaded concurrency, graphical interface interactions, and evolving requirements.

We believe that the agile methodology is particularly well-suited for exploratory and frequently changing development tasks, allowing for continuous delivery and ongoing improvement in short cycles. In the future, we intend to continue using agile development methods in similar projects, supplemented by phased planning to achieve a balance between flexibility and stability.

8. Problems Encountered During Development and Their Solutions

During the development and integration testing in Stage 2, we encountered several practical issues and challenges. Below is a summary of typical problems and the corresponding solutions we adopted:

Thread Termination and Restart Issues:

When implementing dynamic removal of server and barista threads, we discovered that restarting new threads after stopping old ones could lead to conflicts. Initially, we simply interrupted the threads, but they could become stuck in a `wait()` state and fail to exit in time. Our solution involved introducing a thread-running flag (`AtomicBoolean running`) combined with timed waiting: when the user requested to remove a server, we set `running = false` and called `interrupt()` to wake up the thread, allowing it to exit the loop and terminate safely. We then started a new server thread either from a thread pool or by instantiating a new object. Additionally, we ensured that all thread addition/removal operations were executed serially on the GUI event thread to avoid concurrent modifications of the server list. These measures enabled smooth thread restarts: terminated threads released resources properly, and new threads joined the system without interference.

Uneven Thread Workload:

In our initial implementation, we observed that some threads were heavily loaded while others remained idle. For example, servers initially assigned drink-making tasks to baristas in a fixed order, causing the barista with the smallest ID to handle nearly all drink preparation tasks, while others were underutilized. To resolve this imbalance, we modified the task

distribution strategy: drink preparation requests were allocated using round-robin or randomized selection, ensuring a more balanced workload among barista threads. As for servers retrieving customers from the queue, our use of a single shared queue with a fair locking mechanism naturally ensured that the longest-waiting server received the next customer, helping balance the server workload. In cases where one server happened to take multiple online orders consecutively, we introduced a short delay (`waitFor(500ms)`) after each retrieval, giving idle servers a better chance to compete for the next customer. After these adjustments, the workload across server and barista threads became much more balanced, and no individual thread was overloaded.

Queue Capacity and Blocking Issues:

Before we implemented blocking mechanisms, if the customer generation rate exceeded the order processing speed, the queue could grow uncontrollably, consuming excessive memory. To address this, we imposed a maximum queue capacity (e.g., 10 customers). Initially, we considered discarding new customers when the queue was full, but this resulted in order loss, which was unacceptable. Eventually, we adopted a strict producer-consumer model: when the queue was full, the customer generator thread would block and wait in `addCustomer(..., wait=true)` until a server removed a customer and triggered a `notify()`, allowing the producer to resume. This approach both prevented queue overflow and ensured that no orders were lost. Additionally, we added a GUI warning when the queue was full, alerting users to the congestion. Through tight capacity control and coordinated blocking/waking mechanisms, the system effectively resolved overflow issues under high concurrency.

Thread Synchronization and Deadlock Risks:

During the development of interactions between servers and baristas, we were initially concerned about potential deadlocks—for instance, servers waiting for baristas while baristas waited for servers to release locks. Upon analysis, we found that a potential deadlock could occur if a server thread held a customer lock while waiting for the barista callback. To avoid this, we introduced additional lock objects specifically for waiting on drink preparation, separate from the main lock, and used counters or `CountDownLatch` to track the number of unfinished drinks. The server thread would only proceed after all drinks were completed. We also set timeouts during waiting to ensure that unexpected issues (e.g., barista thread failure) would not cause indefinite blocking. Furthermore, we carefully designed the nesting order of synchronized blocks to ensure that no thread would attempt to hold multiple resource locks simultaneously. In the final implementation, collaboration between servers and baristas was deadlock-free, and threads were able to acquire and release locks in an orderly fashion.

GUI Refresh and Thread Concurrency Issues:

Initially, we updated GUI components directly from background threads, which occasionally caused interface lag and runtime exceptions due to violations of Swing's single-thread rule. We promptly corrected this by ensuring that all GUI updates were handled either by observers on the event dispatch thread or wrapped in `SwingUtilities.invokeLater()`. For example, when a server's status changed, it would no longer directly manipulate UI labels; instead, it would call `notifyObservers()` to let the GUI thread handle the updates. In some specific cases, we explicitly used `SwingUtilities.invokeLater()` to encapsulate UI update logic. These measures completely resolved the instability caused by cross-thread UI updates. As a result, both queue changes and thread status updates are now communicated to the GUI in a thread-safe manner, and the interface remains responsive and stable without freezing or flickering.

The process of identifying and solving these problems greatly deepened our understanding of the intricacies of multi-threaded applications. It also reinforced the importance of sound concurrent design—such as the appropriate use of synchronization mechanisms and design patterns—in preventing real-world development pitfalls.