Heriot-Watt University
F21AS- Advanced Software Engineering 2024-25

Report done by

Group Number – 10

Team Members

Harii Ganesh Aravinth (H00477168)

Sreenithi Saravana Perumal (H00486223)

Gayathri Devaraji(H00461375)

Annish Baskar(H00490079)

1. Summary:

Developed for the F21AS coursework, this project offers a Java-based Coffee Shop Simulation System finished in two phases. Key features were built in Stage 1 using a plan-driven approach; for example, end-of-day report, discounting, menu loading, and order processing via a Swing GUI. Emphasis was placed on structured design, unit testing, and exception handling.

Stage two was a multithreaded simulation where various staff threads serve consumers in a queue, emphasising agile development. Along with enhanced GUI to provide real-time processing, design principles like MVC and Singleton were applied. There were also logging and summary statistics.

Apart from satisfying the fundamental and extended criteria of both stages, the completed system shows significant software engineering principles including modular design, teamwork, version control, and iterative development.

2. Group Contribution:

Name (ID)	Contributions
Harii Ganesh Aravinth	Designed the Java Swing GUI for Stage 1 and Stage 2. Designed panels for queue visualisation, order viewing, and item selection. During simulation, dynamic component modifications were integrated in real time. Helped with event logging and made sure MVC design pattern alignment matched. During Stage 2, I took part in agile sprint planning.
Sreenithi Saravana Perumal	Designed and executed fundamental backend logic including order management, menu loading, item validation, and discount application. Managed data serialisation and implemented file I/O handling (menu.txt, orders.txt, logs). Used bespoke exception classes to implement exception handling. Created systematic reporting system for end of the day summary.
Gayathri	Wrote thorough JUnit tests for all fundamental classes—e.g., Order, Discount, Item. Designed test cases for exception classes and edge situations (e.g., InvalidItemIDException, DuplicateItemIdException). Guaranteed complete coverage of billing logic and discount rules. Supported file parsing verification and GUI testing as well.

Annish	Stage 2 development by using thread-based customer and staff simulation. Using Java threads and queues, I developed coordinated serving logic. Included thread-safe emulation of customer service processing. The Singleton approach was used to implement the logging system. Performed performance tests to optimize thread timing
All Members	Designed, documented, and version controlled technical documentation for both phases as well as UML class diagrams. Assisted in Stage 2 iteration tracking and development plan organisation. Kept the Git repository, handled merge conflicts, and controlled branching strategy. Wrote last report chapters and put together appendices.

3. **Repository and version control:**
   Repository link - https://gitlab-student.macs.hw.ac.uk/f21as-2024-25/coffeeshop_simulator Each member of the team worked on dedicated feature branches, allowing parallel development and minimizing conflicts. These branches were regularly merged into the main branch following code reviews. Frequent commits were made throughout the development process to reflect incremental progress, support collaboration, and ensure continuous integration of features.

4. **Specification compliance statement:**
   The Stages 1 and 2 functional and software engineering requirements of the F21AS coursework specification are well addressed in this curriculum. Aside from including multithreading in simulation as well as a design pattern insertion within Stage 2, all of the key features like exception handling, data storage through files, and modularity in design at the Java level are part of the system.
   Stage 1 – Functional & Engineering Requirements Met:
   - Menu and order details are loaded from external files (menu.txt, gui_orders.txt) at application startup.
   - GUI built using Java Swing allows users to browse, select, and submit orders.
   - Discount logic applied using a Strategy Pattern, with classes like StudentDiscountStrategy, SeniorDiscountStrategy.
   - File I/O ensures data persistence across sessions.
   - Reports generated at shutdown (e.g., CoffeeShopReport.txt).
   - Multiple custom exceptions implemented (InvalidItemIdException, DuplicateItemIdException, etc.).
   - JUnit tests included under the Testing folder (e.g., DiscountTest, CoffeeShopTest).

Stage 2 – Functional & Engineering Requirements Met:

- Real-time simulation using threads (CustomerGenerator, ServingStaff, KitchenStaff) with synchronized queues.
- GUI updated dynamically to reflect order queue and staff activity.
- Event logging to file is implemented using a Singleton-based logger.
- Clear separation of concerns (GUI, simulation, models, exceptions).
- Design patterns used:
- Singleton for logger
- Strategy for discounts
- MVC structure inferred from GUI and model separation
- Use of standard Java threads (not thread-safe collections) and proper synchronized access to queues (OrderQueue.java).
- Agile methods reflected through modular iterations (distinct class-level development). **Known Limitations and Minor Incompletions:**
- Simulation start-up does not automatically load and queue orders from the previous session; existing orders are parsed but not used as runtime input.
- GUI rendering may lag during peak thread activity if item load or queue update spikes—this could be due to Swing not being updated on the EDT (Event Dispatch Thread).
- Some JUnit test cases are missing coverage for edge scenarios (e.g., multiple concurrent discounts, malformed order lines).
- Logger setup fails to run during the start of the program resulting in partial meeting of logging requirement.
- No in-app control to dynamically adjust simulation speed or add/remove staff during runtime (as suggested in extended requirements).

5. **UML Class diagram:**

6. **Data structures and design rationale:**

Class	Structure Used	Purpose	Reason
Menu	Map,String,Item>	Store menu items using unique identifiers as keys.	Allows constant-time lookup of items and prevents duplicates. Ensures item retrieval by ID is efficient and safe.

OrderQueue	Queue<Order>	Hold customer orders waiting to be processed by serving staff.	Implements a FIFO structure to simulate real-world queuing behavior, where the first customer to arrive is the first to be served.
Billing	LinkedHashMap	Track order item breakdown by	Maintains insertion order while providing fast access to detailed
	<Integer, Map<String, Object>>	order ID for generating bills.	item-level data. Useful for transparent billing displays.
DiscountManager/ DiscountStrategy	Map<String, DiscountStrategy>	Store and apply different discount policies (student, senior, combo, etc.).	Facilitates strategy pattern implementation, enabling flexible addition or modification of discount rules without changing the billing logic.
Simulator/ CustomerGenerator	List<Customer>	Simulate a pool of customers entering the queue at timed intervals.	List allows dynamic sizing and sequential access while iterating through customers to generate events for the simulation.
FileInputOutput	List<String>	Temporarily store lines from input files before parsing into structured data.	Flexible and simple data container for handling line-by-line file processing, especially when loading menu and order files.

7. Functional Implementation Overview:

7.1. Stage 1 Features

Menu Item Loading from File with Category + ID

The FileInputOutput class is responsible for reading menu items from menu.txt, which includes fields for item ID (e.g., FOOD123), name, category, and price. The Menu class stores items in a Map<String, Item> to ensure fast lookup and ID uniqueness.

Order Loading at Startup

Customer orders are read from gui_orders.txt at application start. Each line corresponds to an order item with timestamp and customer ID. These are parsed and grouped into Order objects, although this feature is more fully utilized during simulation in Stage 2.

GUI for Placing Orders

CoffeeShopGUI enables users to browse menu items, select multiple items, and place an order through a user-friendly interface. The GUI reflects categories and prices, integrates order creation, and links directly to the CoffeeShop model. **Discount Logic**

Discounts are applied using the Strategy Pattern:

- StudentDiscountStrategy: 10% discount for student ID orders
- SeniorDiscountStrategy: 15% discount for seniors
- Rules are encapsulated in DiscountStrategy implementations and applied via the Discount class. Multiple strategies can be composed.

Summary Report on Exit

On exit, CoffeeShop.generateReport() compiles a CoffeeShopReport.txt, detailing:

- Total number of orders
- Items sold and counts
- Total income
- Discount Impact

7.2 Stage 2 Features

Multithreaded Order Queue (e.g., Staff Threads)

OrderQueue manages customer orders using a synchronized Queue<Order>.
ServingStaff and KitchenStaff are separate threads that consume and process orders.
CustomerGenerator thread simulates customer arrivals, adding orders to the queue.

Real-Time Simulation Using Swing GUI

CoffeeShopGUI has been enhanced to show live status updates for staff activity.
Customers in queue and currently served orders are visualized using GUI components.

Live Status of Orders & Serving Staff

Each ServingStaff thread updates the GUI with the currently served customer and order details.
Thread coordination is reflected live, with synchronized updates to avoid race conditions.

Event Logging (e.g., to simulation_log.txt) A
logging system writes real-time events such as:

- Order added to queue
- Order served
- Staff availability

Logging is handled centrally, likely using a Singleton Logger (based on pattern usage).

Simulation Ends on Empty Queue

When the order queue is empty and all orders are served, the simulation stops gracefully.

A final report is generated at shutdown, similar to Stage 1 but now includes thread activity logs.

Optional Extensions

While the core functionality is well-implemented, the following advanced features were not present or only partially implemented:

Adaptive queue size or simulation speed control
Dynamic staff assignment during runtime
Online/pre-order handling
Multiple discount strategies (student/senior)
Multithreaded producer-consumer simulation

8. Testing & exception Handling

Exception	**Purpose**	**Thrown In**
InvalidItemIdExpection	Validates item ID format (eg, F00d001)	Item constructor and FileInputOutput.liadMenu()
DuplicateItemIdException	Prevents duplicate menu item entries	Menu.addItem()
ItemNotFoundException	Raised when an item lookup fails	Order.addItem()
InvalidCategoryException	Ensures category values are from known types	Item constructor or parser logic
PriceOutOfRangeException	Verifies that item price is within a valid range	Item constructor

9. Use of Threads and Synchronization

The Stage 2 simulation introduces a multithreaded environment that mirrors a real-life coffee shop workflow using producer-consumer logic.

Thread Type	**Role**
CustomerGenerator	Acts as the producer, generating orders and placing them into the OrderQueue
ServingStaff	Consumer threads that serve orders taken from the queue

KitchenStaff	Processes food items or simulates kitchen preparation

Synchronization Techniques Used
The system uses manual synchronization with synchronized, wait(), and notifyAll() to safely manage concurrent access to shared queues (OrderQueue and PreparedOrderQueue).
OrderQueue.java:
Uses a Queue<order> Structure for storing incoming orders.
Implements synchronized enqueue() and dequeue().
PreparedOrderQueue.java:
Used to pass completed food orders from Kitchen to serving staff.
Also uses synchronized, wait(), and notifyAll() to coordinate handoff between KitchenStaff and ServingStaff

10. Use of Design Patterns

Pattern	**Used In**	**Purpose**
Singleton	CoffeeShop.java	Ensures only one instance of the CoffeeShop class is created throughout the application. This allows centralized state management (menu, orders, revenue) and prevents inconsistencies across GUI and backend components. Implemented using a private static CoffeeShop instance and getInstance() method.
Strategy	Discount.java, DiscountStrategy.java, StudentDiscountStrategy.java, SeniorDiscountStrategy.java	Allows flexible discount logic via interchangeable strategy objects. New discount types can be added without modifying existing logic.

Observer	OrderQueue.java	Implements the Observer pattern by maintaining a list of observers (likely GUI components) that are notified (notifyObservers()) when the queue changes, e.g., new order enqueued or dequeued. Ensures real-time updates to the UI or log handlers.
MVC	CoffeeShopGUI (View), CoffeeShop (Model), controller logic embedded in action listeners	Separates interface from business logic. The GUI reflects the model state and delegates logic to backend classes. Improves modularity and readability.

11. Agile Process Reflection

During Stage 2, our team moved from the plan-driven approach used in Stage 1 to a more agile, iterative development process. Rather than planning every item up front, we divided the task into three focused iterations, each of which generated incremental progress. In the initial iteration, we included the key simulation components—OrderQueue, ServingStaff, KitchenStaff, and CustomerGenerator—and reorganised the Stage 1 codebase for more modularity. Live GUI updates were integrated, multithreaded processes synchronised, and event logging configured using the observer design in the second iteration. The last version fine-tuned the GUI for real-time simulation, stress-tested thread behaviour, and finished report generation and exception handling.

Though we didn't follow a formal agile methodology like Scrum, we employed several agile-inspired strategies. Weekly stand-up meetings helped to track personal progress and spot challenges. Task allocation was done clearly among the members; version control branches were used to handle threading and GUI enhancements. Especially while fine-tuning the discount strategy method and merging the thread logic with the GUI, pair programming was used sparingly during difficult integrations. Regular Git contributions enabled continuous integration by guaranteeing that little modifications were routinely reviewed and merged.

This nimble approach tremendously helped to shape simulations. We could change our code and respond quickly to new difficulties when fresh needs and errors appeared. Although flexible logic separation and testing were made feasible by the use of design patterns like Strategy and Observer, the modular design from Stage 1 made it easier to improve already-existing classes. Code reviews significantly improved the general quality of the product and were particularly useful in spotting GUI update errors and threading issues.

There were, however, other challenges. Integration of independently designed parts often caused compatibility issues, especially when thread-safe code interacted with non-thread-safe Swing GUI components. Some thread timing and synchronisation problems only appeared during stress testing, therefore manual adjustment of wait/notify cycles and sleep intervals was required. The early project's lack of documentation also made debugging across components more difficult; GUI thread safety had to be handled later in development, which required more work.

Having looked at the distinctions between Stages 1 and 2, we found that both methods have benefits. Although the agile process of Stage 2 permitted the rapid creation, testing, and enhancement of features like multithreading and real-time GUI updates, the plan-driven approach of Stage 1 created a well-structured basis but lacked adaptability. Because it encouraged more cooperation and flexibility, agile development is better appropriate for complicated and evolving needs, such as found in simulation systems. Combining formal design with iterative execution offered, overall, the best mix between stability and inventiveness.

12. Stage1 vs Stage 2 development comparison:

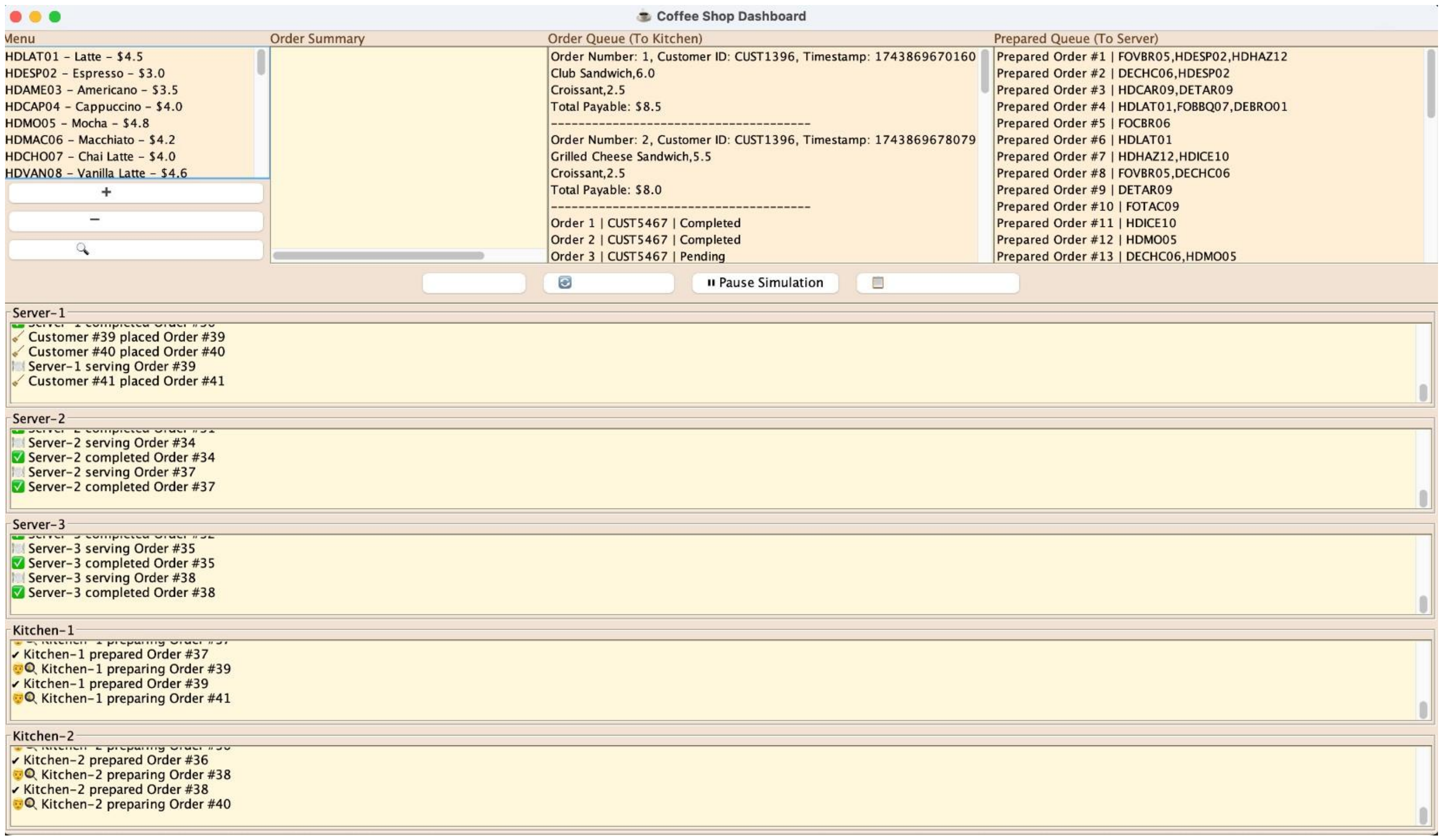
Between Stages 1 and 2 of the development process, our technology strategy and team dynamics obviously changed. In Stage 1, we used a planned, design-first strategy, spending a significant amount of effort designing the system architecture before implementation. The focus was on developing a solid object-oriented foundation using classes such Item, Menu, Order, and CoffeeShop. Though it grew rigid as Stage 2 needs arose, this approach fostered reusability and clarity in the code. For example, integrating threading logic or live GUI updates required major reworking of hitherto static components.

Stage 2's more flexible and agile development approach allowed us to slowly create and enhance capabilities such synchronised order queues, multithreaded customer simulation, and real-time GUI feedback.   Instead of strictly adhering to pre-defined frameworks, we took a refactor-as-needed strategy that let us modify the system as technical problems came up.   This modification was particularly effective with Observer-based GUI updates and the Producer-Consumer threading architecture, where dynamic feedback and regular code changes were vital.

Stage 1 gave individual contributions according to preset modules top priority in terms of teamwork; Stage 2 encouraged more cooperation and cross-functional problem-solving. By means of version control more aggressively, regular check-ins, and coordinated debugging sessions, we oversaw the merging and testing of concurrent features during Stage 2. Ultimately, Stage 2 had more complexity but encouraged a deeper sense of teamwork and mutual responsibility.

Looking back, the agile approach in Stage 2 was more suitable for our team given the interactive character of the simulation system and the need for adaptability.   Though Stage 1's initial design was beneficial for establishing a clean basis, it lacked the adaptability required for multithreaded, real-time applications.   Apart from showing us that good software engineering sometimes requires changing methods depending on project complexity and scope, the difference between the two phases highlighted the need of finding a balance between design discipline and iterative development.

13. Screenshots of the system:



```
HDLAT01 – Latte – $4.5
HDESP02 – Espresso – $3.0
HDAME03 – Americano – $3.5
HDCAP04 – Cappuccino – $4.0
HDMO05 – Mocha – $4.8
HDMAC06 – Macchiato – $4.2
HDCHO07 – Chai Latte – $4.0
HDVAN08 – Vanilla Latte – $4.6
```

```
Order Number: 1, Customer ID: CUST1396, Timestamp: 1743869670160
Club Sandwich,6.0
Croissant,2.5
Total Payable: $8.5
----------------------------------------
Order Number: 2, Customer ID: CUST1396, Timestamp: 1743869678079
Grilled Cheese Sandwich,5.5
Croissant,2.5
Total Payable: $8.0
----------------------------------------
Order 1 | CUST5467 | Completed
Order 2 | CUST5467 | Completed
Order 3 | CUST5467 | Pending
```

```
Prepared Order #1 | FOVBR05,HDESP02,HDHAZ12
Prepared Order #2 | DECHC06,HDESP02
Prepared Order #3 | HDCAR09,DETAR09
Prepared Order #4 | HDLAT01,FOBBQ07,DEBRO01
Prepared Order #5 | FOCBR06
Prepared Order #6 | HDLAT01
Prepared Order #7 | HDHAZ12,HDICE10
Prepared Order #8 | FOVBR05,DECHC06
Prepared Order #9 | DETAR09
Prepared Order #10 | FOTAC09
Prepared Order #11 | HDICE10
Prepared Order #12 | HDMO05
Prepared Order #13 | DECHC06,HDMO05
```

```
Customer #39 placed Order #39
Customer #40 placed Order #40
Server-1 serving Order #39
Customer #41 placed Order #41
```

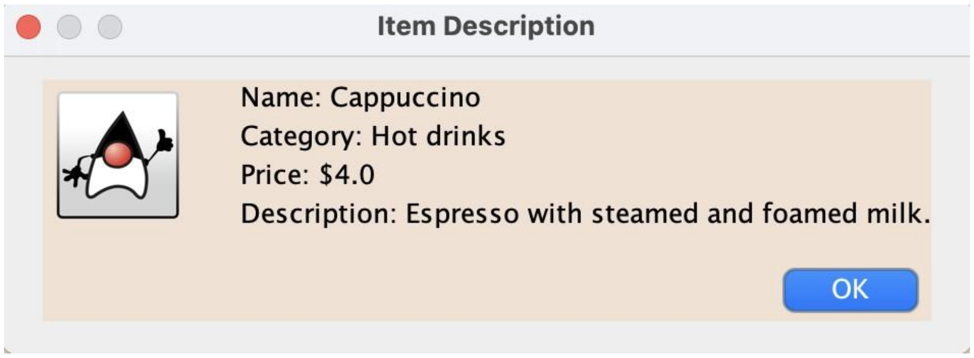
```
Server-2 serving Order #34
✅ Server-2 completed Order #34
Server-2 serving Order #37
✅ Server-2 completed Order #37
```

```
Server-3 serving Order #35
✅ Server-3 completed Order #35
Server-3 serving Order #38
✅ Server-3 completed Order #38
```

```
✔ Kitchen-1 prepared Order #37
Kitchen-1 preparing Order #39
✔ Kitchen-1 prepared Order #39
Kitchen-1 preparing Order #41
```

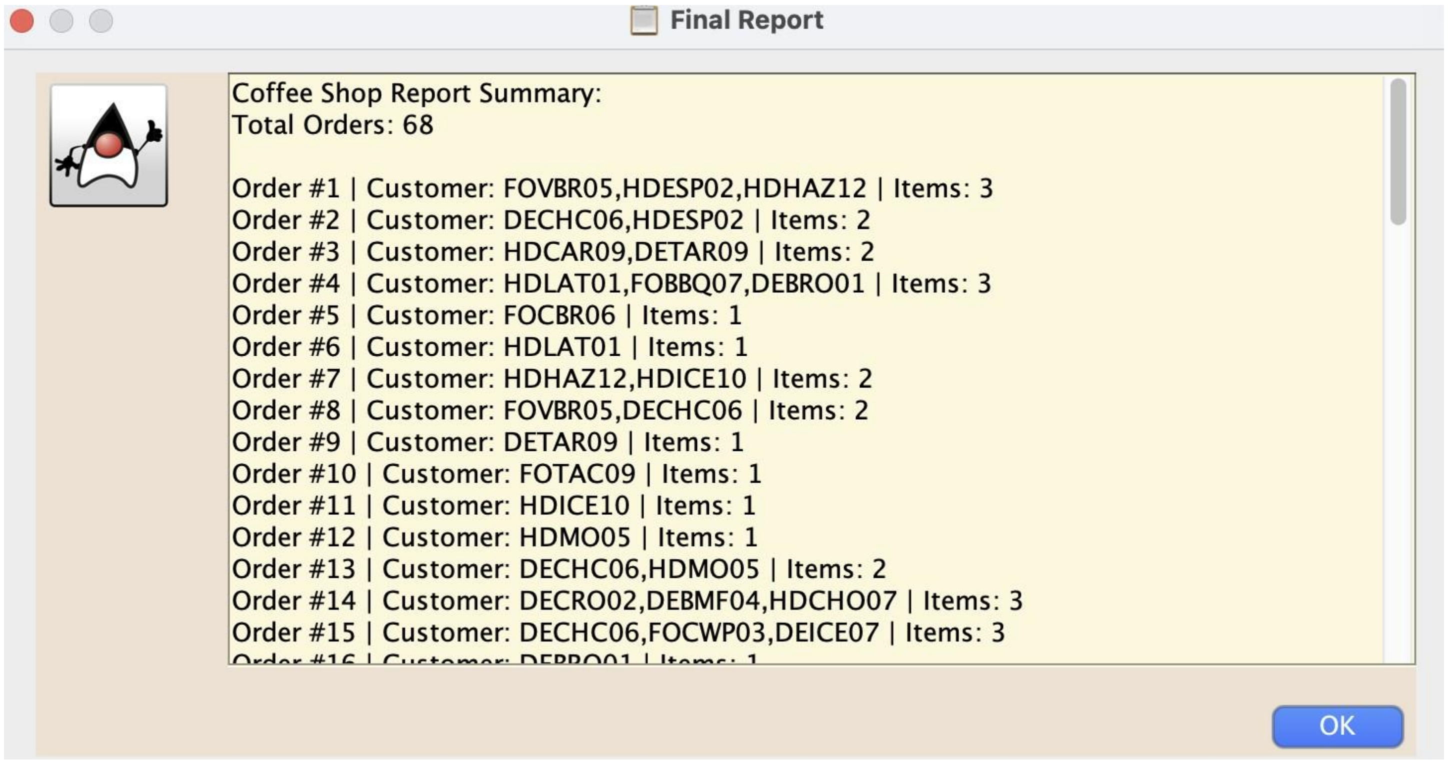
```
✔ Kitchen-2 prepared Order #36
Kitchen-2 preparing Order #38
✔ Kitchen-2 prepared Order #38
Kitchen-2 preparing Order #40
```

Fig1: Overview of the GUI



Name: Cappuccino
Category: Hot drinks
Price: $4.0
Description: Espresso with steamed and foamed milk.

Fig2: Item Description



```
Coffee Shop Report Summary:
Total Orders: 68

Order #1 | Customer: FOVBR05,HDESP02,HDHAZ12 | Items: 3
Order #2 | Customer: DECHC06,HDESP02 | Items: 2
Order #3 | Customer: HDCAR09,DETAR09 | Items: 2
Order #4 | Customer: HDLAT01,FOBBQ07,DEBRO01 | Items: 3
Order #5 | Customer: FOCBR06 | Items: 1
Order #6 | Customer: HDLAT01 | Items: 1
Order #7 | Customer: HDHAZ12,HDICE10 | Items: 2
Order #8 | Customer: FOVBR05,DECHC06 | Items: 2
Order #9 | Customer: DETAR09 | Items: 1
Order #10 | Customer: FOTAC09 | Items: 1
Order #11 | Customer: HDICE10 | Items: 1
Order #12 | Customer: HDMO05 | Items: 1
Order #13 | Customer: DECHC06,HDMO05 | Items: 2
Order #14 | Customer: DECRO02,DEBMF04,HDCHO07 | Items: 3
Order #15 | Customer: DECHC06,FOCWP03,DEICE07 | Items: 3
Order #16 | Customer: DEBRO01 | Items: 1
```

Fig3: Final report structure

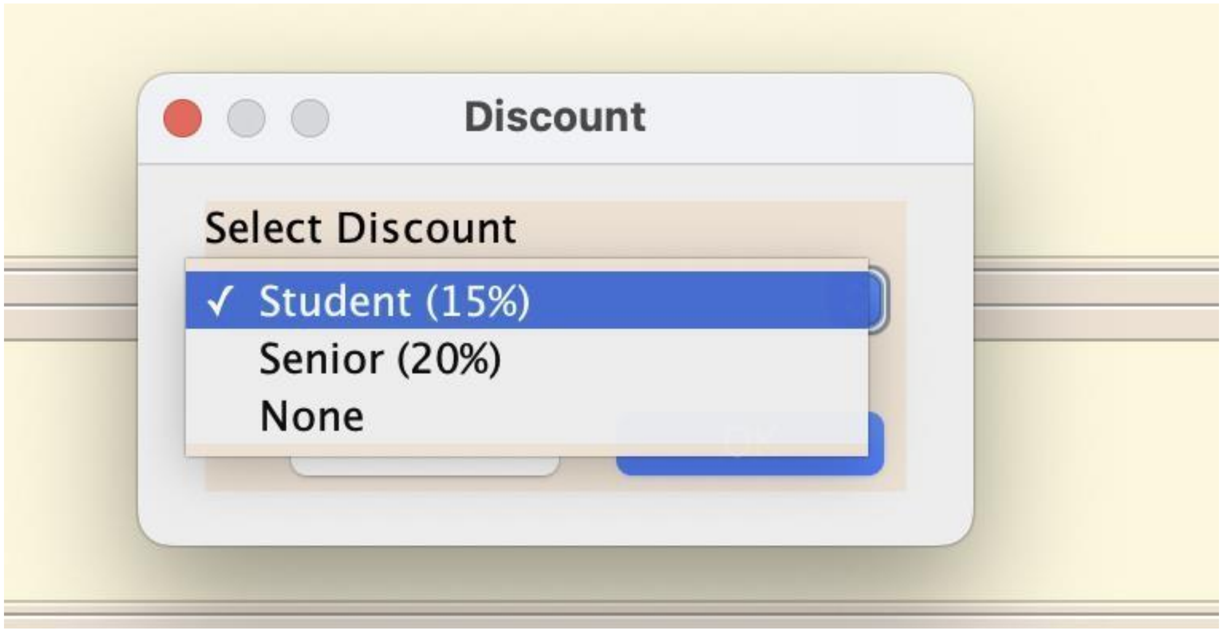
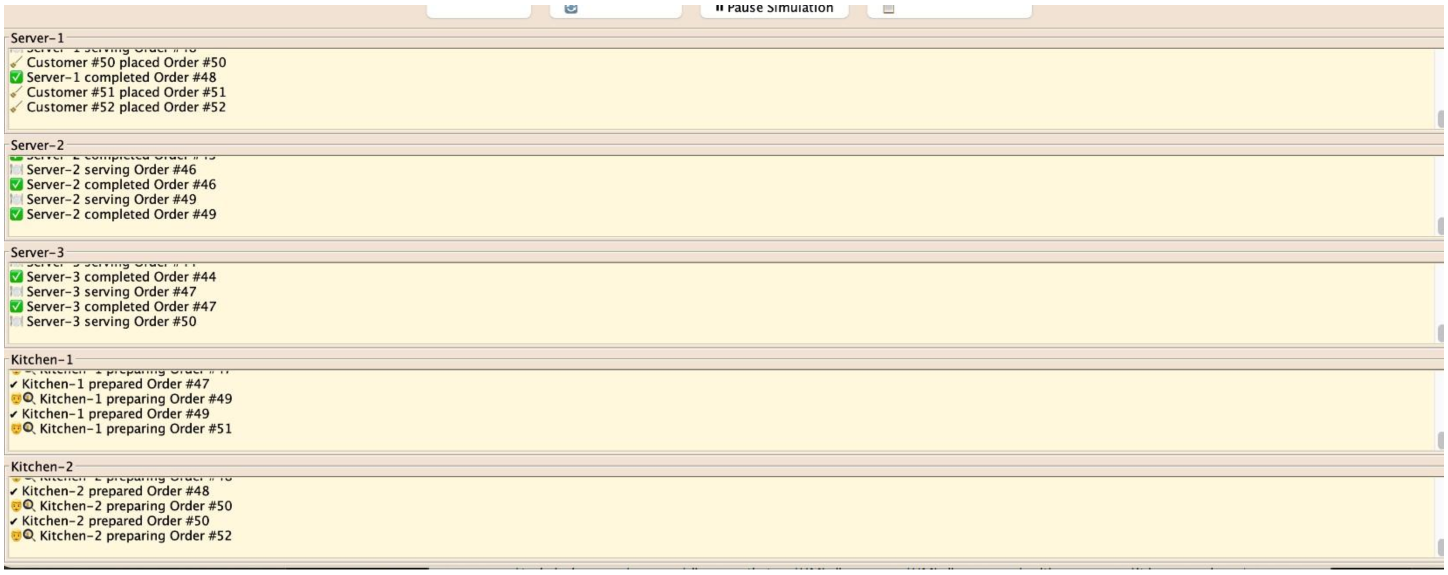
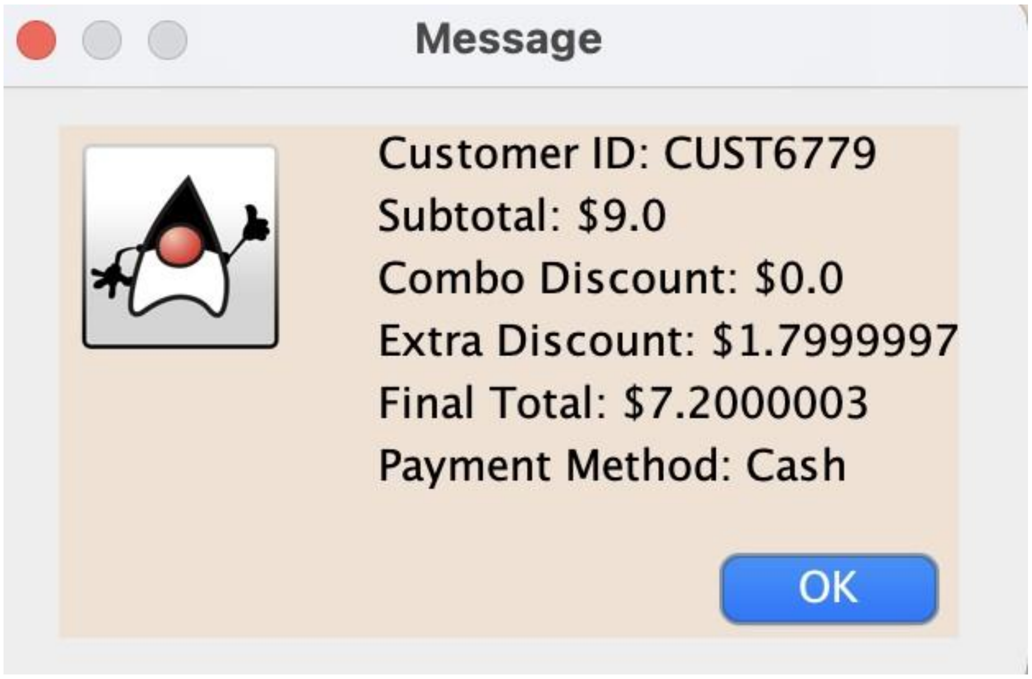


Fig4: Discount StrategyConclusion



```
Server-1
Customer #50 placed Order #50
✅ Server-1 completed Order #48
Customer #51 placed Order #51
Customer #52 placed Order #52

Server-2
Server-2 serving Order #46
✅ Server-2 completed Order #46
Server-2 serving Order #49
✅ Server-2 completed Order #49

Server-3
✅ Server-3 completed Order #44
Server-3 serving Order #47
✅ Server-3 completed Order #47
Server-3 serving Order #50

Kitchen-1
✔ Kitchen-1 prepared Order #47
🔍 Kitchen-1 preparing Order #49
✔ Kitchen-1 prepared Order #49
🔍 Kitchen-1 preparing Order #51

Kitchen-2
✔ Kitchen-2 prepared Order #48
🔍 Kitchen-2 preparing Order #50
✔ Kitchen-2 prepared Order #50
🔍 Kitchen-2 preparing Order #52
```

Fig5: Thread activity window



```
Customer ID: CUST6779
Subtotal: $9.0
Combo Discount: $0.0
Extra Discount: $1.7999997
Final Total: $7.2000003
Payment Method: Cash
```

Fig6: Order Summary

14. Conclusion:

The Coffee Shop Simulation project effectively shows how fundamental software engineering ideas are applied in both scheduled and agile development phases. Essential aspects like file-based order and menu administration, Swing-based GUI for customer interaction, discount plan execution, and summary reporting were the team's focus in Stage 1. Emphasis was on modular object-oriented design, exception handling, and test-driven development.

Stage 2 turned the system into a dynamic multithreaded simulation. The application effectively reproduced actual-time consumer and personnel interactions using synchronised queues and producerconsumer threading. The system's scalability, maintainability, and responsiveness were improved by means of design patterns such as Singleton, Strategy, MVC, and Observer.

Version control, collaboration, and incremental improvement guaranteed the system met vital requirements specifications during the development stage. Though a few complex extensions are available for more expansion, the final product is a robust, expandable, and well-designed system presenting sound software design and team-oriented development ideas.