Restaurant Management System

Problem Statement:
A mid-sized restaurant faces operational challenges due to manual order processing, table booking, and billing procedures. These manual processes lead to delays, errors, and poor customer experience. To modernize operations, you are tasked with designing and developing a Java console-based Restaurant Management System that supports real-time updates and multiple concurrent users.

Solution:
In this system, when a customer enters the restaurant, they first view the list of tables displaying table number, seating capacity, and availability status. The customer selects a single table to book, and since multiple customers may attempt booking the same table concurrently, the booking operation is synchronized using a shared persistent store (file/JSON) to ensure consistency. If two customers try to reserve Table 1 at the same time, only the first successful request locks the table, while the second customer receives a message that the table is already booked and must choose another one. After confirming the table booking, the customer can browse the menu, select multiple items with quantities, and place an order. Each order is treated as a task and submitted into an order queue, where waiter worker threads (managed using a fixed-size ExecutorService) pick up incoming orders and dispatch them to the kitchen queue. Chef worker threads then process these tasks concurrently, simulate preparation time, and upon completion update the order status to PREPARED, enabling smooth parallel execution of restaurant operations from booking to food preparation.

User Story 1: On-Arrival Table Booking

As a customer,
I want to view all tables with their seating capacity and availability status when I arrive,
so that I can choose and book a suitable table immediately.

Acceptance Criteria:
The system displays all tables with table number, seats, and status (Available/Booked).
Customers can select one or more tables for booking.
Booked tables are immediately marked unavailable.

User Story 2: Concurrent Table Booking Conflict Handling

As a customer,
I want the system to prevent double-booking when multiple customers try to book tables at the same time,
so that table allocation remains accurate and fair.

Acceptance Criteria:
Two customers cannot book the same table simultaneously.
If some requested tables are already booked, the customer receives partial allocation.
Customers are asked to confirm or cancel the booking based on available tables.

User Story 3: Menu Browsing and Item Selection

As a customer,
I want to view the restaurant menu with item details such as price and preparation time,
so that I can decide what to order.

Acceptance Criteria:
Menu shows item name, price, estimated cooking time, and available quantity.
Customers can select multiple items with required quantities.

User Story 4: Placing an Order from the Table

As a customer,
I want to place my selected food items as an order after booking a table,
so that the kitchen can start preparing my food.

Acceptance Criteria:
Order is created with table ID and selected menu items.
Order is submitted successfully into the restaurant's processing pipeline.
The customer receives confirmation that the order was placed.

User Story 5: Checkout Request After Order Completion

As a customer,
I want to request checkout only after all my orders are prepared,
so that I can complete my dining session correctly.

Acceptance Criteria:
Bill/checkout option is enabled only when all orders for that table are

completed.
Customers cannot checkout if orders are still pending.

User Story 6: Waiter Thread Order Forwarding

As the restaurant system,
I want waiter worker threads to pick up customer orders and forward them to the kitchen queue,
so that orders are delivered in real time without manual delays.

Acceptance Criteria:
Orders placed by customers are assigned to waiter threads.
Waiter threads push orders into the kitchen queue immediately.

User Story 7: Kitchen Queue Management

As the restaurant system,
I want incoming orders to be stored in a synchronized kitchen queue,
so that chefs can process them efficiently in the correct order.

Acceptance Criteria:
Orders are queued safely using a thread-safe structure (e.g., BlockingQueue).
Multiple chefs can consume orders without conflict.

User Story 8: Concurrent Order Preparation by Chefs

As the restaurant system,
I want chef worker threads to prepare multiple orders simultaneously,
so that service remains efficient during peak hours.

Acceptance Criteria:
Chef threads process orders concurrently using a thread pool.
Order status updates from RECEIVED → IN_PROGRESS → COMPLETED.

User Story 9: Real-Time Order Status Updates

As the restaurant system,
I want order status changes to be reflected immediately across customer and kitchen modules,
so that all users see live updates.

Acceptance Criteria:
When chefs complete an order, customers can see the updated status.
Kitchen view always shows active and completed orders correctly.

User Story 10: Automatic Table Release After Checkout

As the restaurant system,
I want tables to be automatically marked available after checkout,
so that they can be booked again by new customers.

Acceptance Criteria:
Once checkout is complete, table status resets to Available.
Released tables instantly appear free in the shared table view.

User Story 11: Shared Table Availability Across Multiple Terminals

As the restaurant system,
I want table booking data to remain consistent across multiple running instances of the application,
so that customers booking from different terminals see the same live availability.

Acceptance Criteria:
Table status is stored in a shared file-based persistence layer.
File locking prevents race conditions during concurrent bookings.
Updates made in one terminal are visible immediately in another terminal.

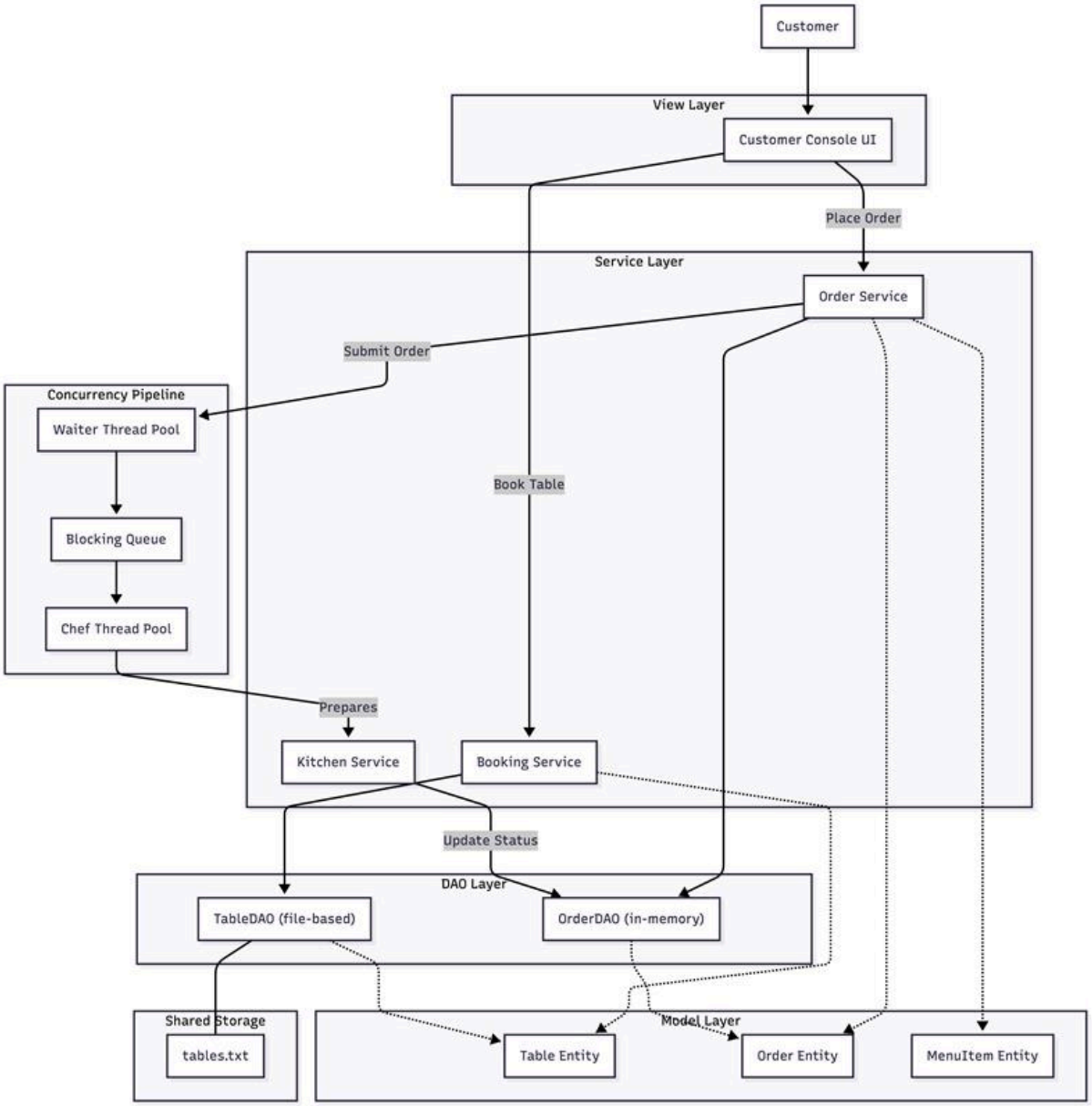

Fig-1:High Level Design

Fig-1 represents a **layered architecture** designed for a restaurant management system. The system follows clear separation of concerns across View, Service, DAO, and Model layers, with asynchronous order processing.

Flow of the project

- Customers interact with the system.

- Requests go to the View Layer (Customer Console).
- The View delegates actions to the Service Layer.
- Services interact with:
    - DAO layer (data access)
    - Concurrency pipeline (for order processing)
- Data is stored in:
    - File-based storage (tables)
    - In-memory storage (orders)

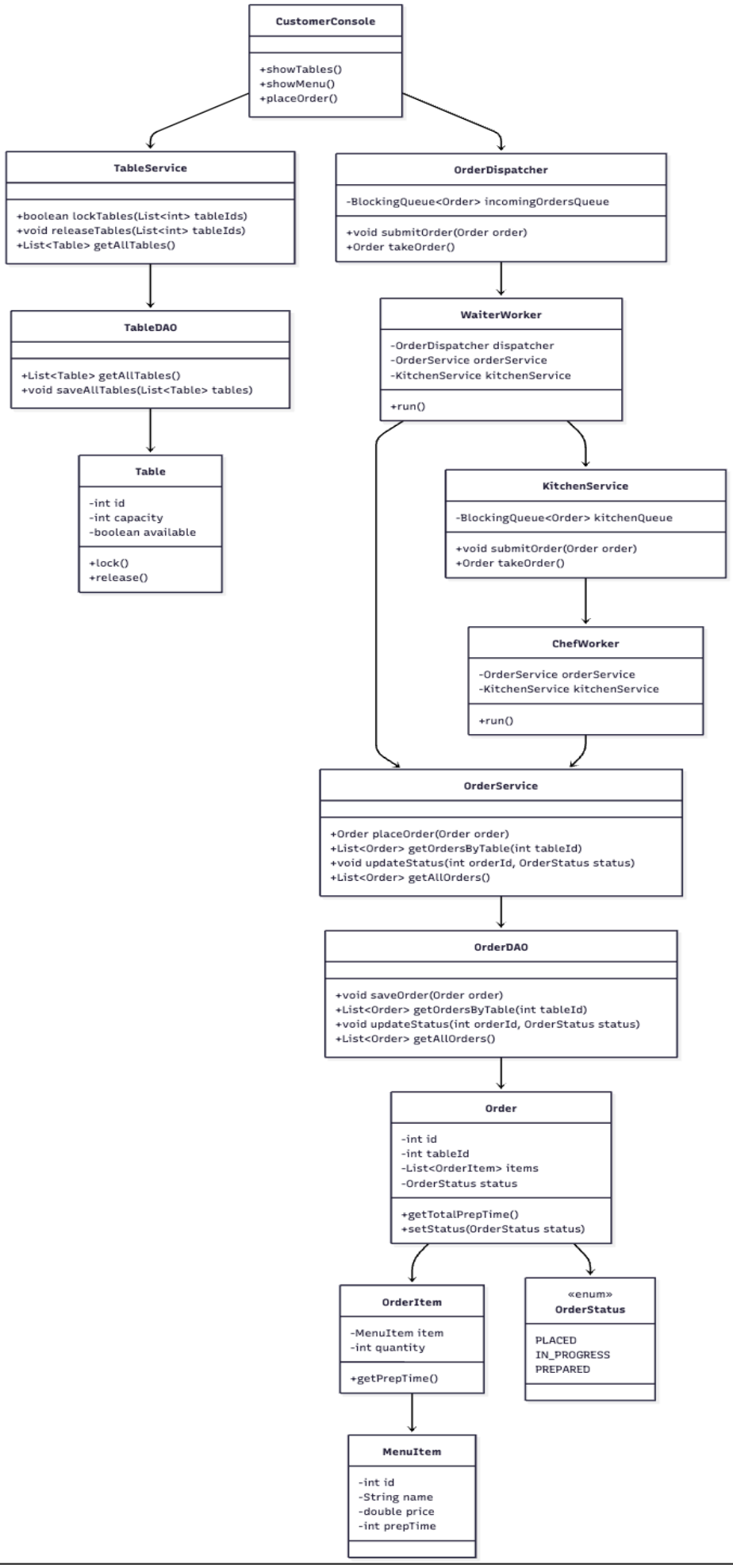
CustomerConsole
+showTables()
+showMenu()
+placeOrder()
TableService
+boolean lockTables(List<int> tableIds)
+void releaseTables(List<int> tableIds)
+List<Table> getAllTables()
OrderDispatcher
-BlockingQueue<Order> incomingOrdersQueue
+void submitOrder(Order order)
+Order takeOrder()
TableDAO
+List<Table> getAllTables()
+void saveAllTables(List<Table> tables)
WaiterWorker
-OrderDispatcher dispatcher
-OrderService orderService
-KitchenService kitchenService
+run()
Table
-int id
-int capacity
-boolean available
+lock()
+release()
KitchenService
-BlockingQueue<Order> kitchenQueue
+void submitOrder(Order order)
+Order takeOrder()
ChefWorker
-OrderService orderService
-KitchenService kitchenService
+run()
OrderService
+Order placeOrder(Order order)
+List<Order> getOrdersByTable(int tableId)
+void updateStatus(int orderId, OrderStatus status)
+List<Order> getAllOrders()
OrderDAO
+void saveOrder(Order order)
+List<Order> getOrdersByTable(int tableId)
+void updateStatus(int orderId, OrderStatus status)
+List<Order> getAllOrders()
Order
-int id
-int tableId
-List<OrderItem> items
-OrderStatus status
+getTotalPrepTime()
+setStatus(OrderStatus status)
OrderItem
-MenuItem item
-int quantity
+getPrepTime()
«enum»
OrderStatus
PLACED
IN_PROGRESS
PREPARED
MenuItem
-int id
-String name
-double price
-int prepTime

Fig-2: Low Level design

1. Model Layer (Domain Layer)

The Model Layer contains the core business entities of the restaurant system. These classes represent real-world objects such as tables, menu items, and orders. This layer contains pure domain logic and remains independent of upper layers like services or views.

The MenuItem class represents a dish available in the restaurant menu. It stores static information such as a unique identifier (id), the name of the dish (name), its price (price), and the preparation time required (prepTime). Its main responsibility is to describe a menu dish and provide basic details for ordering.

The OrderItem class represents a single line item inside a customer order. It consists of a MenuItem along with the quantity ordered. This class helps calculate preparation time based on the selected dish and quantity, making it an essential building block for complete orders.

The OrderStatus enum defines the lifecycle of an order in a type-safe manner. It includes the states PLACED, IN_PROGRESS, and PREPARED. Using an enum prevents invalid string-based status values and ensures consistent order state transitions.

The Order class represents a complete customer order. It contains a unique order ID, the associated table ID, a list of OrderItem objects, and the current status of the order. It provides methods such as getTotalPrepTime() to compute the overall preparation time and lifecycle-related methods like markPrepared() to update the order state. This class encapsulates order behavior and manages the order lifecycle.

The Table class represents a restaurant table. It stores information such as table ID, seating capacity, and availability status. It provides methods like lock() to mark a table as unavailable and release() to make it available again. This class encapsulates state changes related to table booking.

2. DAO Layer (Data Access Layer)

The DAO Layer is responsible for data storage and retrieval. It isolates persistence logic from business logic, ensuring that the service layer does not directly handle file operations or storage details.

The TableDAO component handles persistent storage of table data using a file-based approach. It provides methods such as getAllTables() to retrieve all tables from storage, lockTables() to reserve specified tables, releaseTables() to free them, and saveToFile() to persist updated table information. Its responsibility is to manage table availability consistently and ensure correct booking behavior even under concurrent access.

The OrderDAO component manages order storage using an in-memory approach. It supports operations like saving a new order, retrieving orders for a specific table, and updating the status of an order. This DAO maintains active order data and supports lifecycle state updates during order preparation.

3. Service Layer (Business Logic Layer)

The Service Layer contains the core business logic and orchestration of the system. It coordinates DAO operations, concurrency workers, and domain-level rules.

The BookingService is responsible for table reservation operations. It provides methods such as bookTables() which internally calls TableDAO.lockTables() and returns whether the booking was successful. It also supports releasing tables through TableDAO.releaseTables(). This service ensures that reservation logic is handled cleanly outside the persistence layer.

The OrderService manages the complete lifecycle of an order. When a customer places an order, this service creates a new Order object, saves it through OrderDAO, submits it into the waiter workflow, and returns the created order. It acts as the central coordinator between storage and concurrency components.

The KitchenService manages kitchen processing using a producer–consumer mechanism. It maintains a blocking queue of incoming orders and provides methods to start chef worker threads, submit orders into the kitchen queue, and mark orders as prepared. It updates order status by calling DAO methods and ensures asynchronous kitchen workflow execution.

4. Concurrency Workers

The concurrency layer introduces multithreaded workers to simulate real restaurant operations where multiple orders are processed simultaneously.

The OrderDispatcher acts as the central incoming order queue. Customers submit their orders into this dispatcher, and waiter threads consume orders from

it. It works as a buffer between customer input and waiter processing, ensuring that orders are safely handled in a thread-safe queue.

The WaiterWorker represents waiter threads in the system. Waiters continuously take orders from the OrderDispatcher, save them through OrderService, update their status to IN_PROGRESS, and forward them into the kitchen queue managed by KitchenService. This worker acts as a producer for the kitchen workflow.

The ChefWorker represents chef threads. Chefs continuously consume orders from the kitchen queue, simulate preparation using preparation time, and once completed, update the order status to PREPARED. This worker acts as the final consumer in the producer–consumer pipeline.

5. View Layer (Presentation Layer)

The View Layer provides the user interaction entry point of the system. It does not contain business rules but delegates actions to the service layer.

The CustomerConsole class handles customer interaction through a console interface. It provides methods such as showTables() to display table availability, showMenu() to list menu items, and placeOrder() to collect customer order input. Once input is collected, it calls OrderService.placeOrder() to trigger the complete ordering workflow.

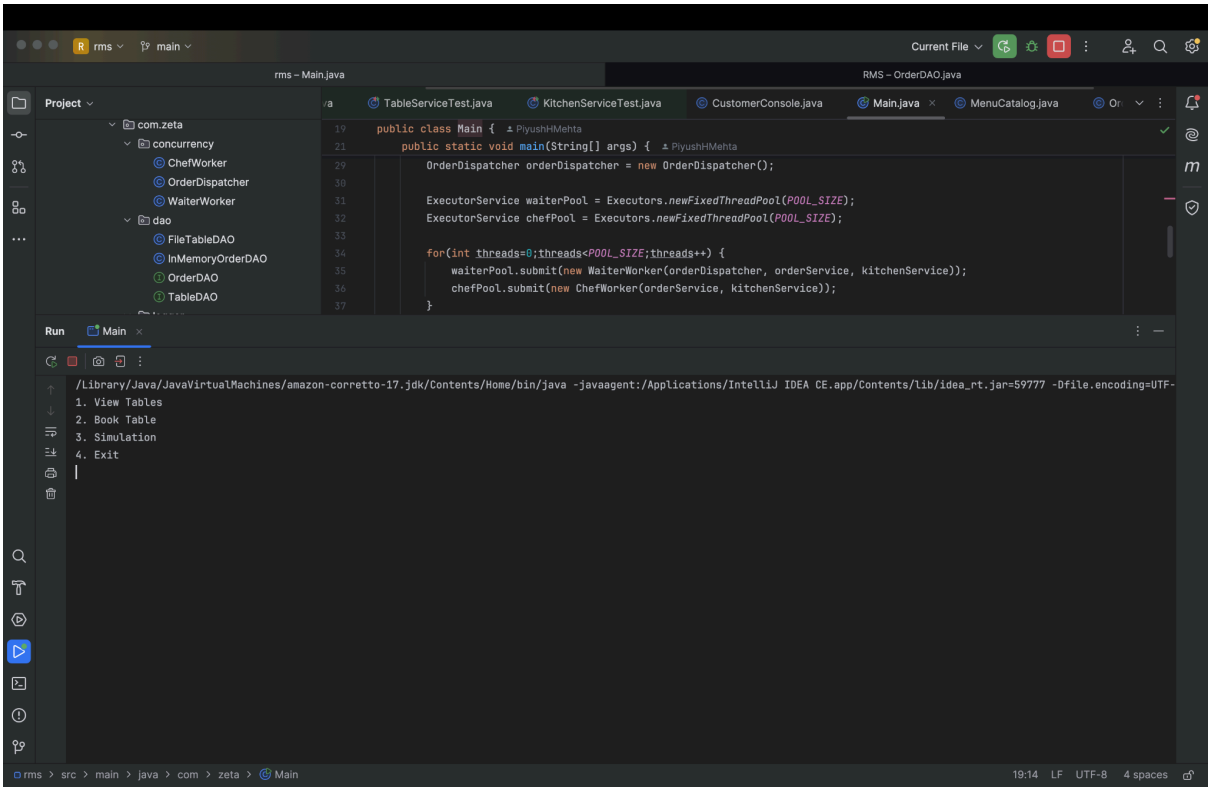

Fig 3 : Customer Console for booking and to view tables

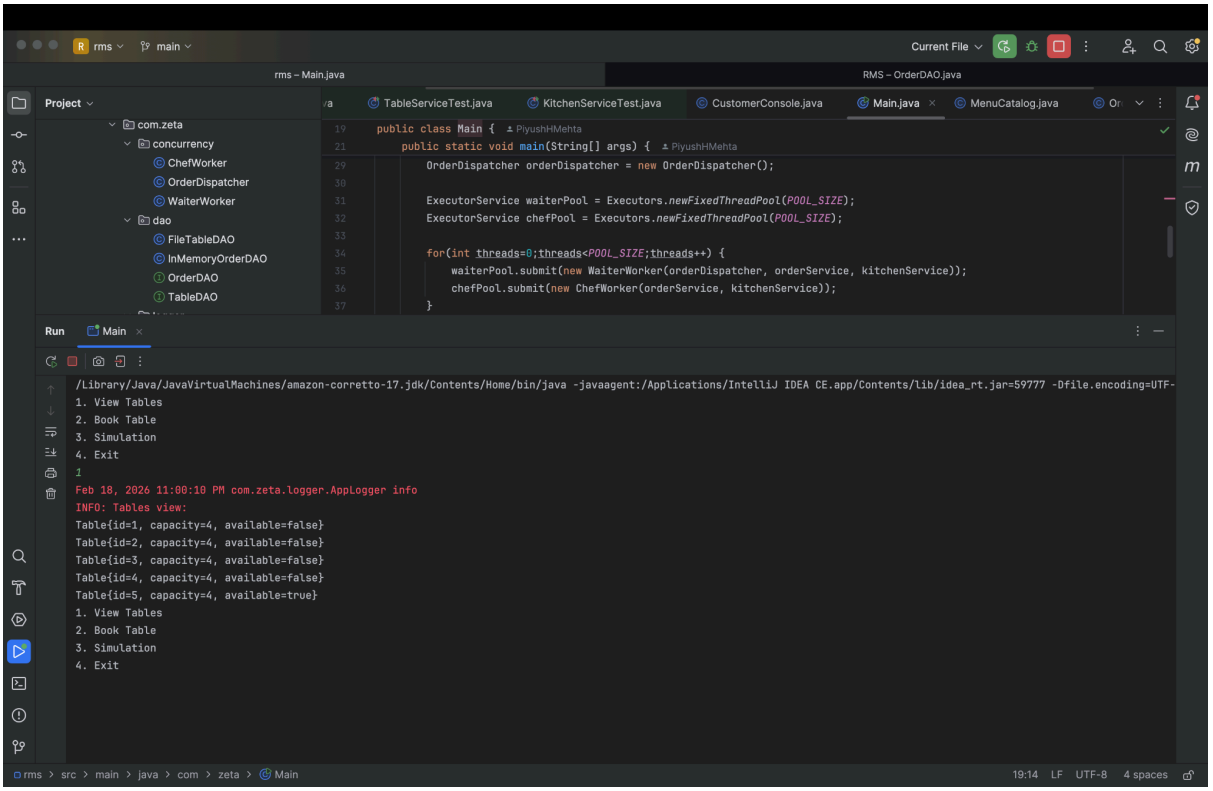

Fig 4 : view of tables with availability and capacity of table

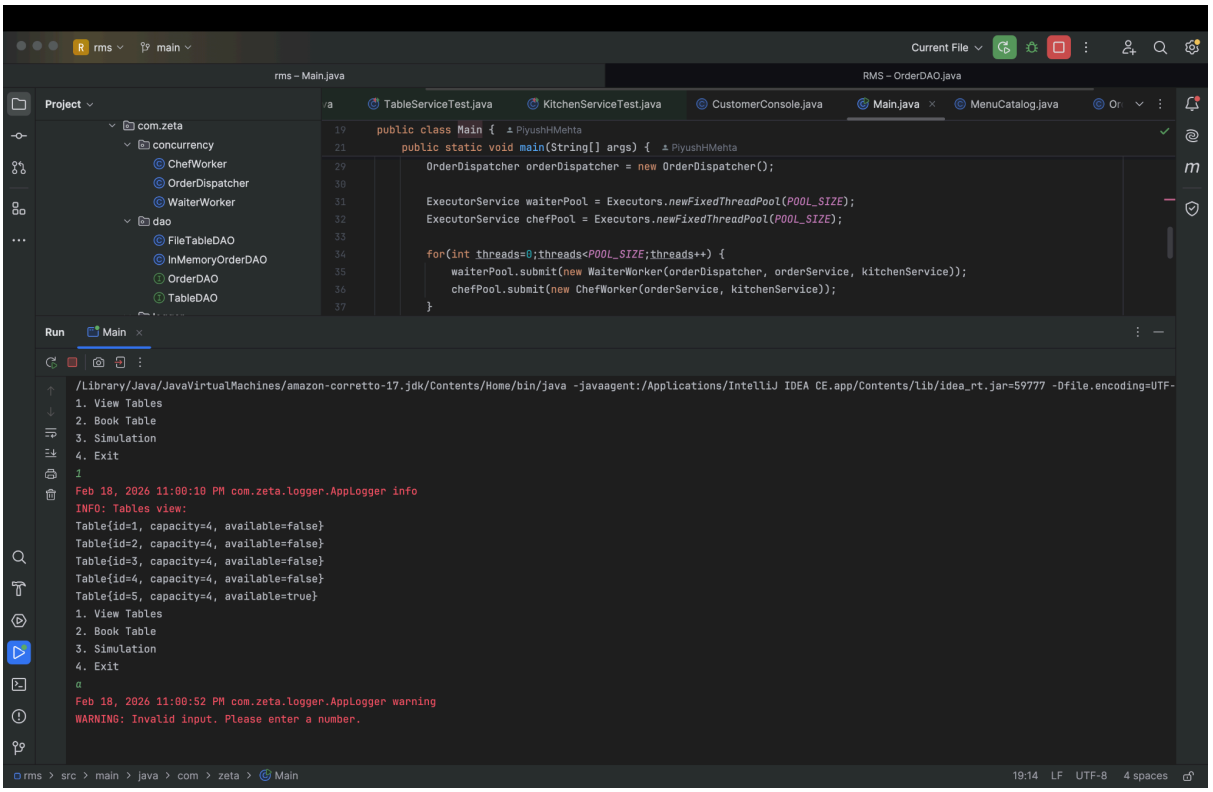

Fig 5 : Validation for Integer Input

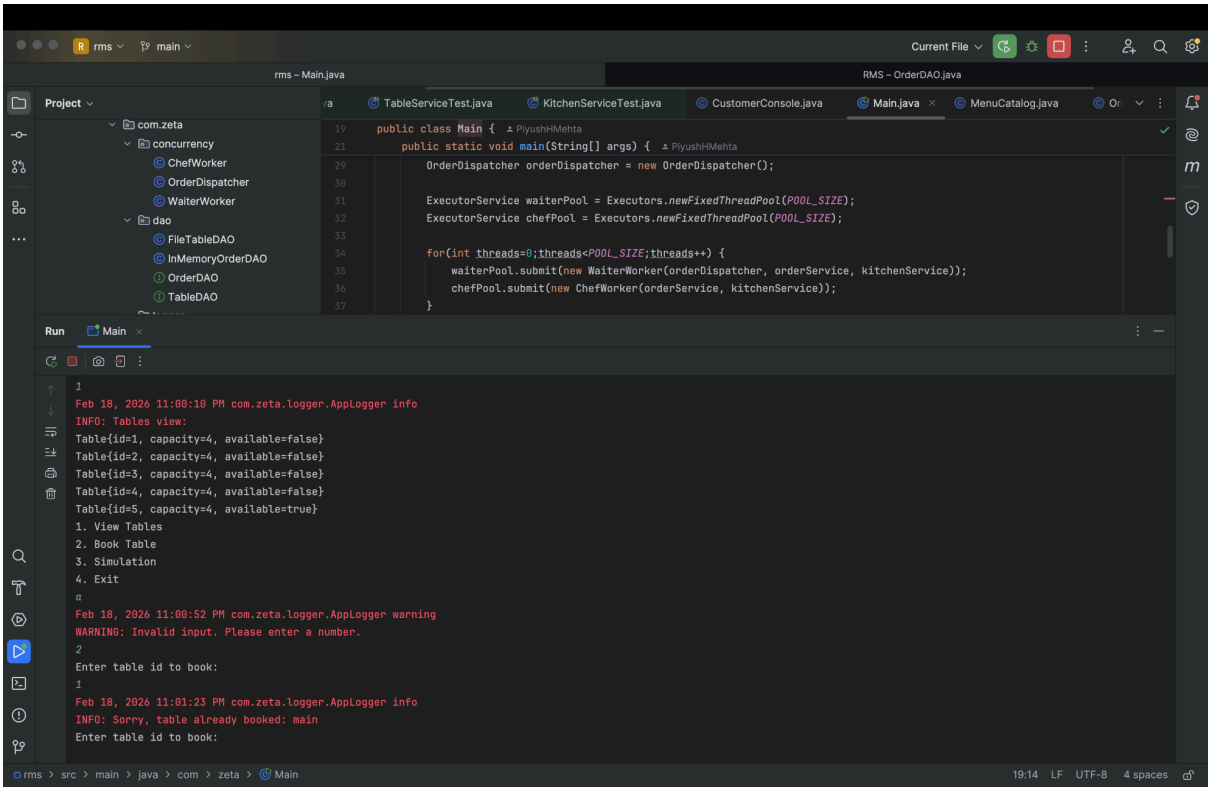

Fig 6 : Booking is not allowed for already booked table

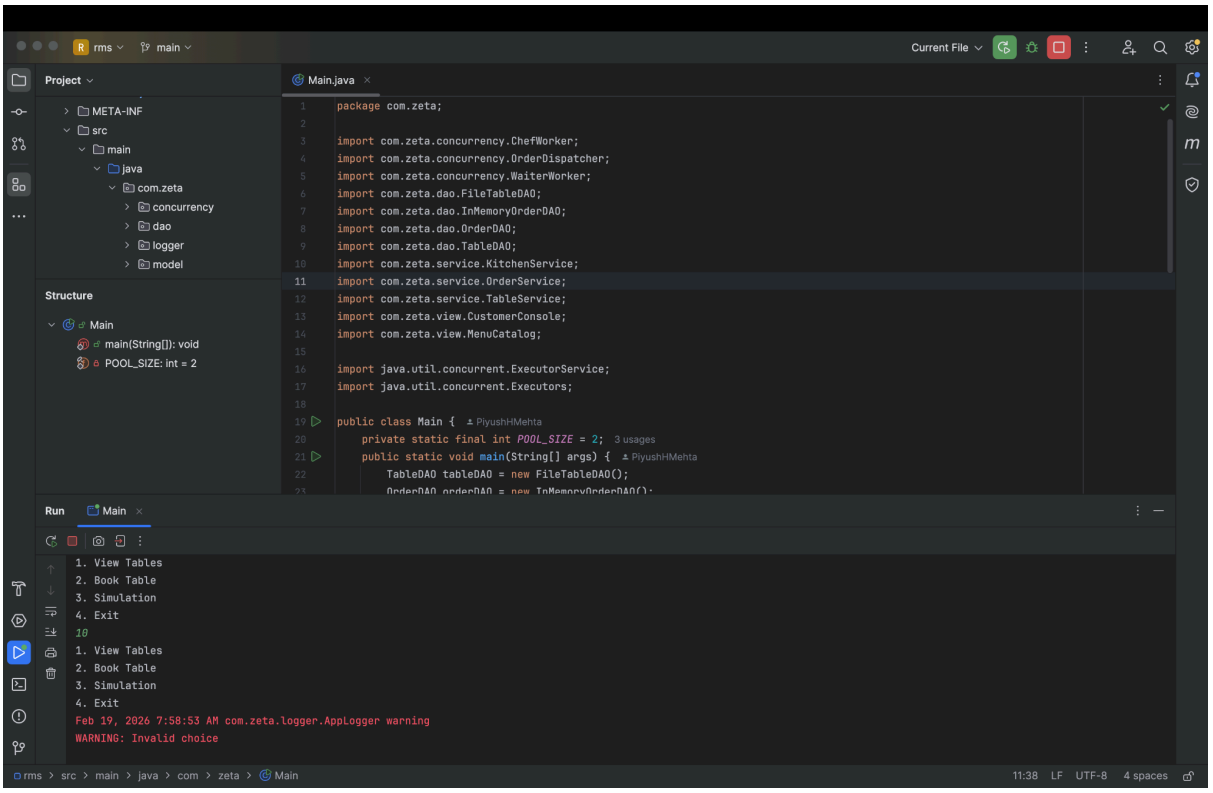

Fig 7 : Booking is not allowed for Invalid table Id

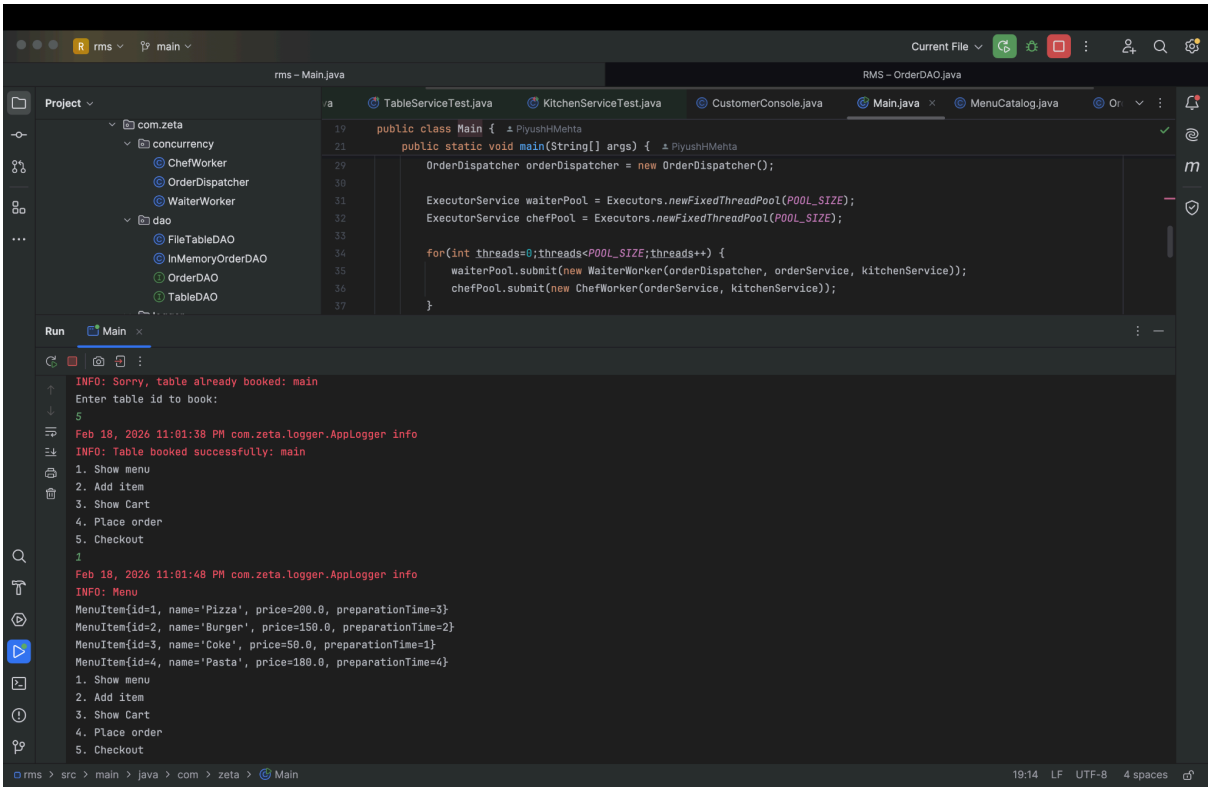

Fig 8 : Table Booking for valid input

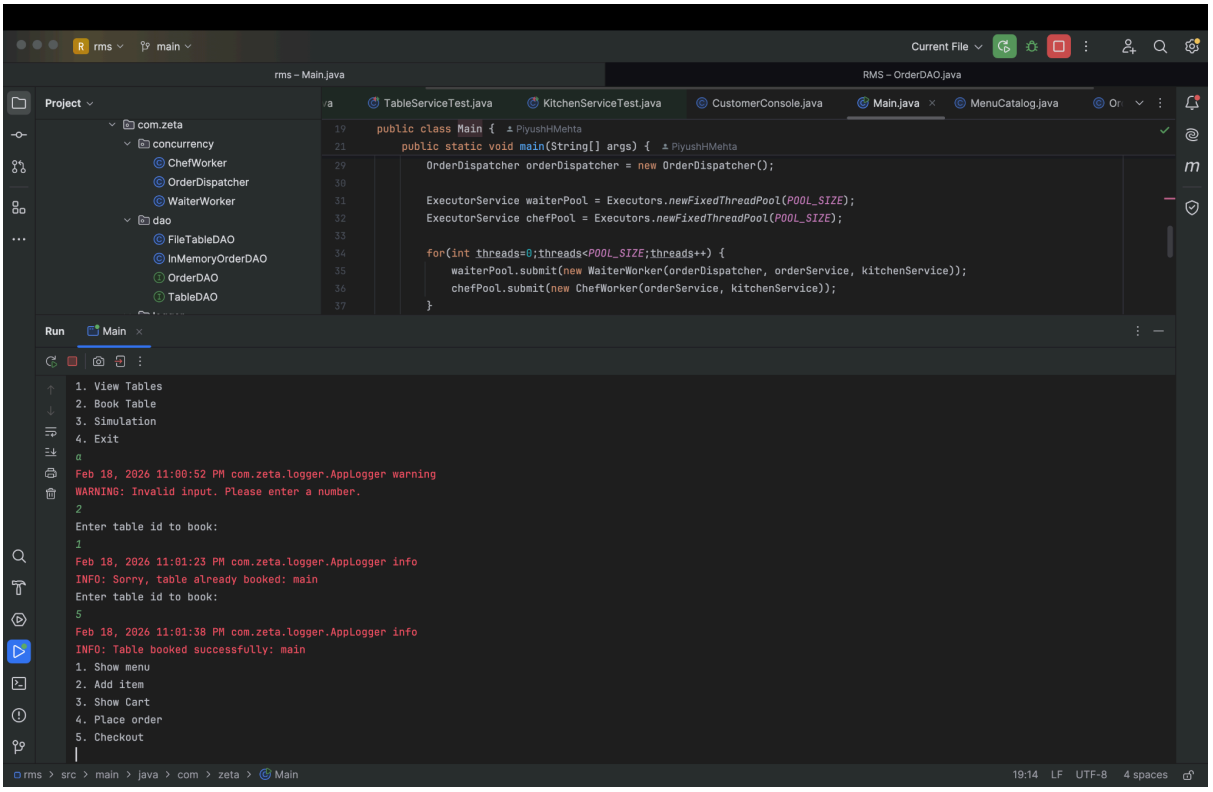

Fig 9 : Customer Console for order Giving

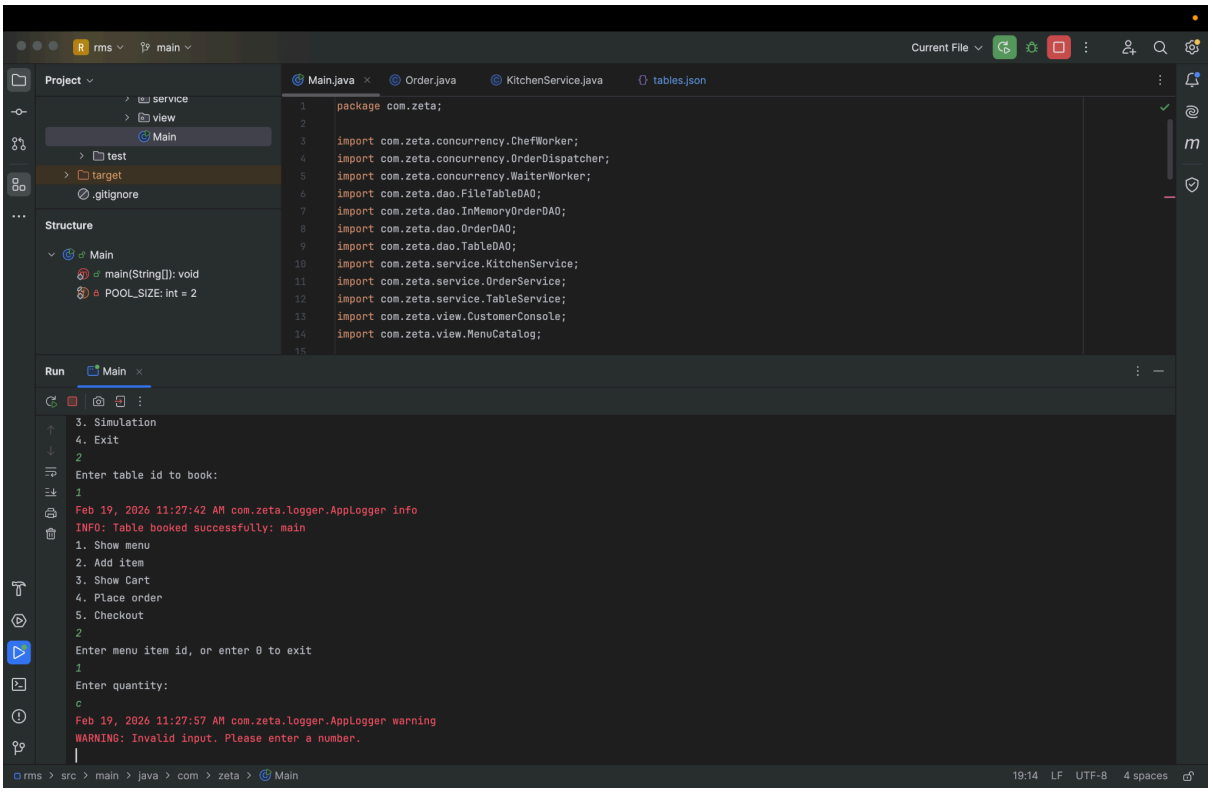

Fig 10 : validation for quantity

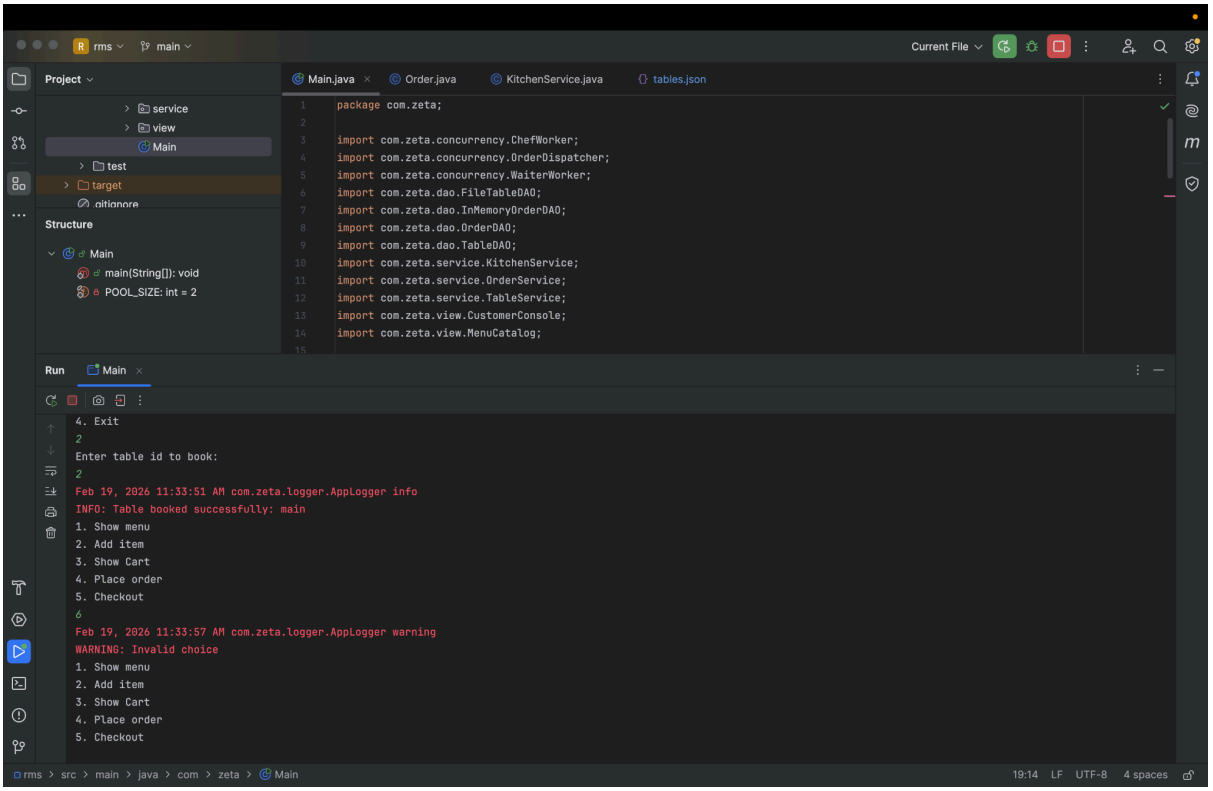

Fig 11 : Validating invalid choice in menu

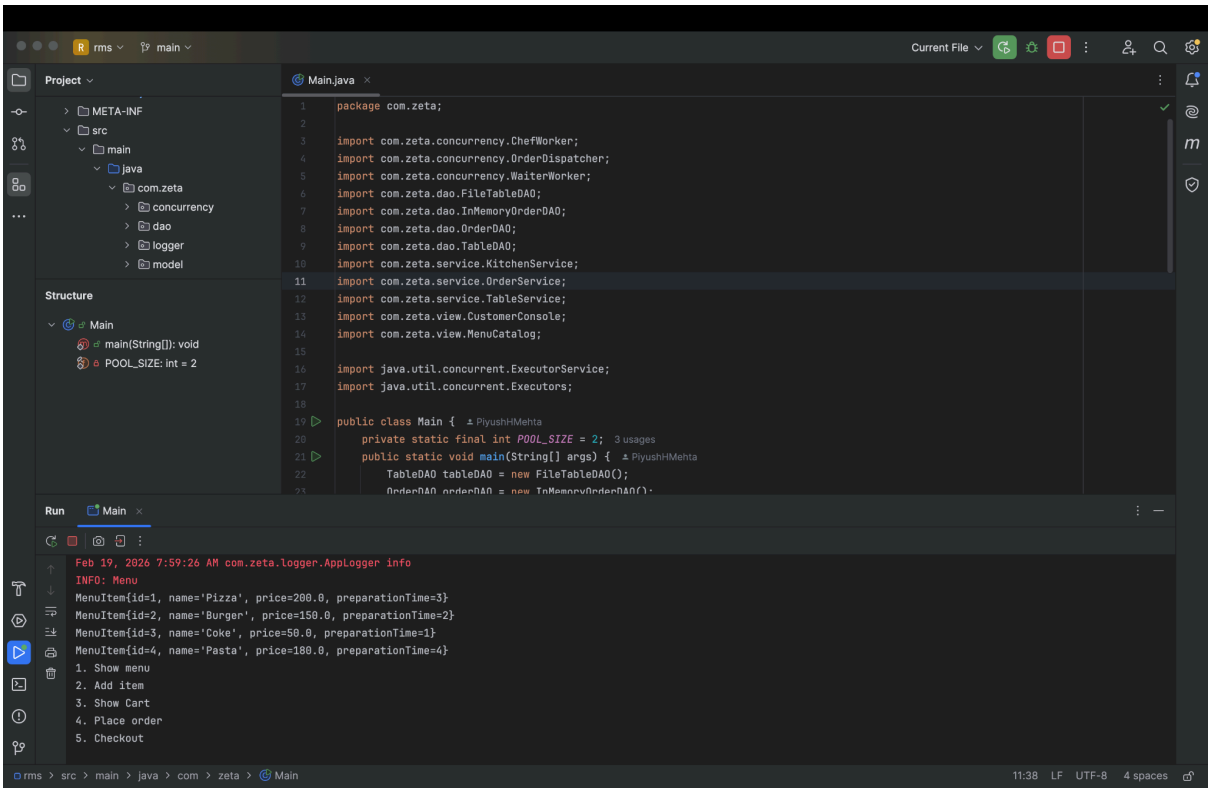

Fig 12 : Printing Menu

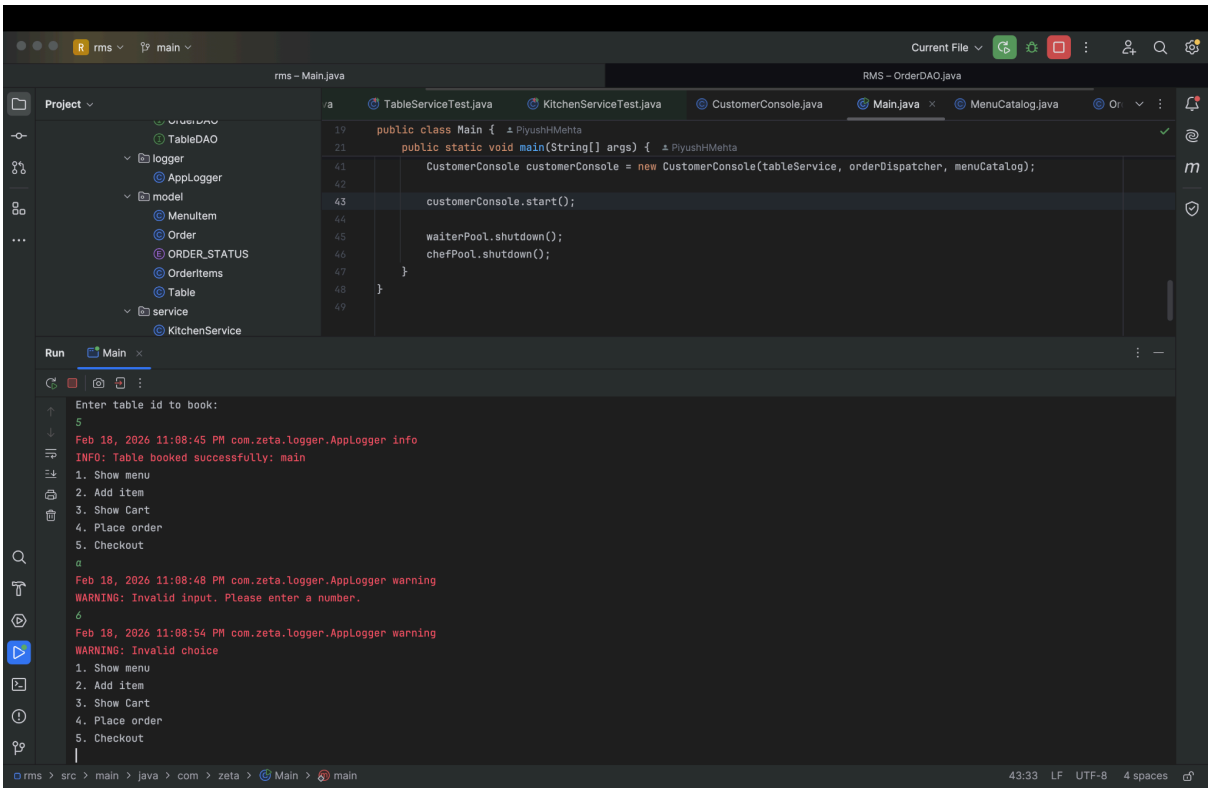

Fig 13 : Verifying Invalid input in menu

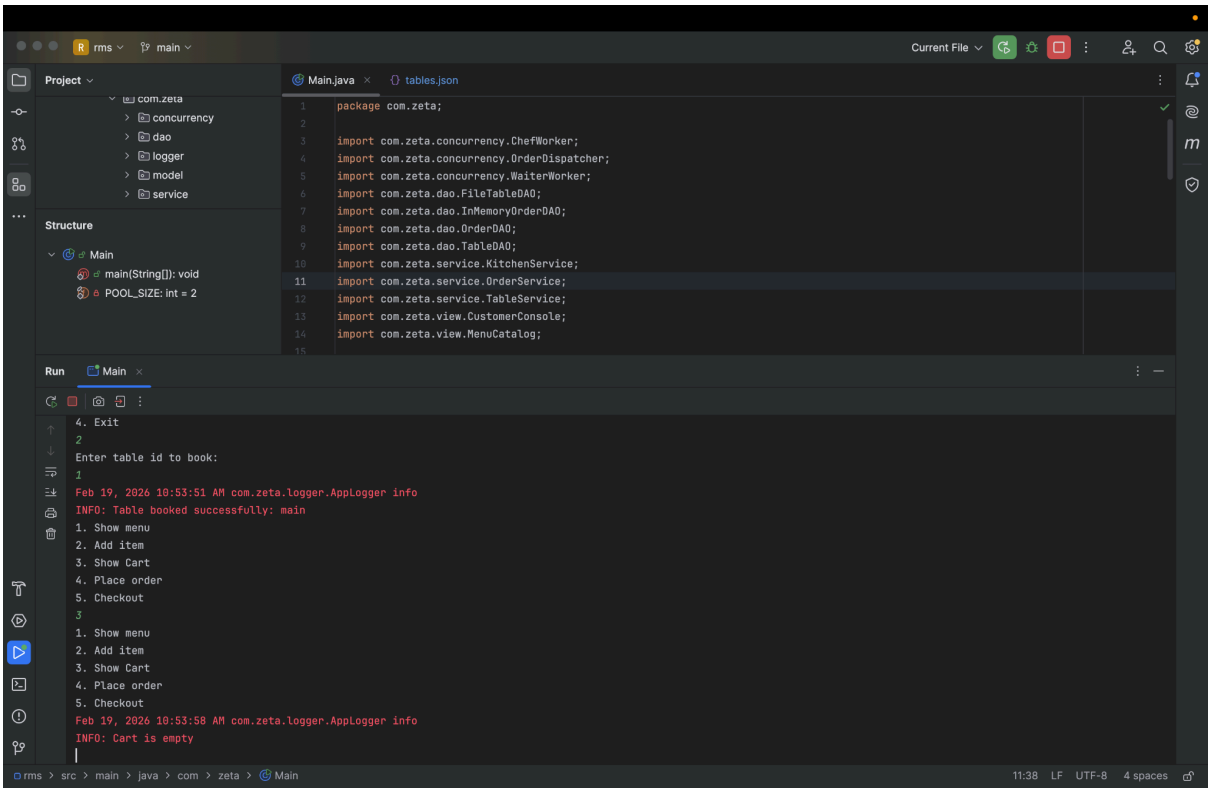

Fig 14 : show cart , cart is empty items not added to cart

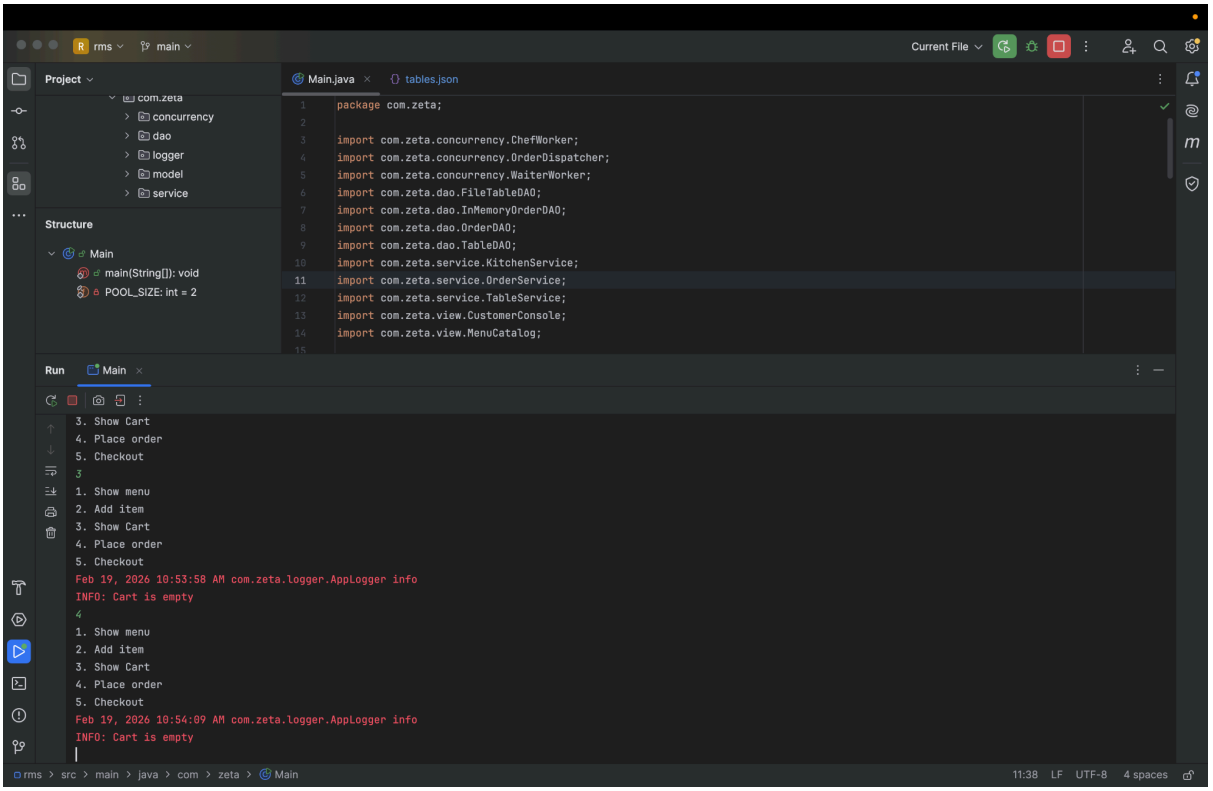

Fig 15 : Place order , order will not get placed cart is empty

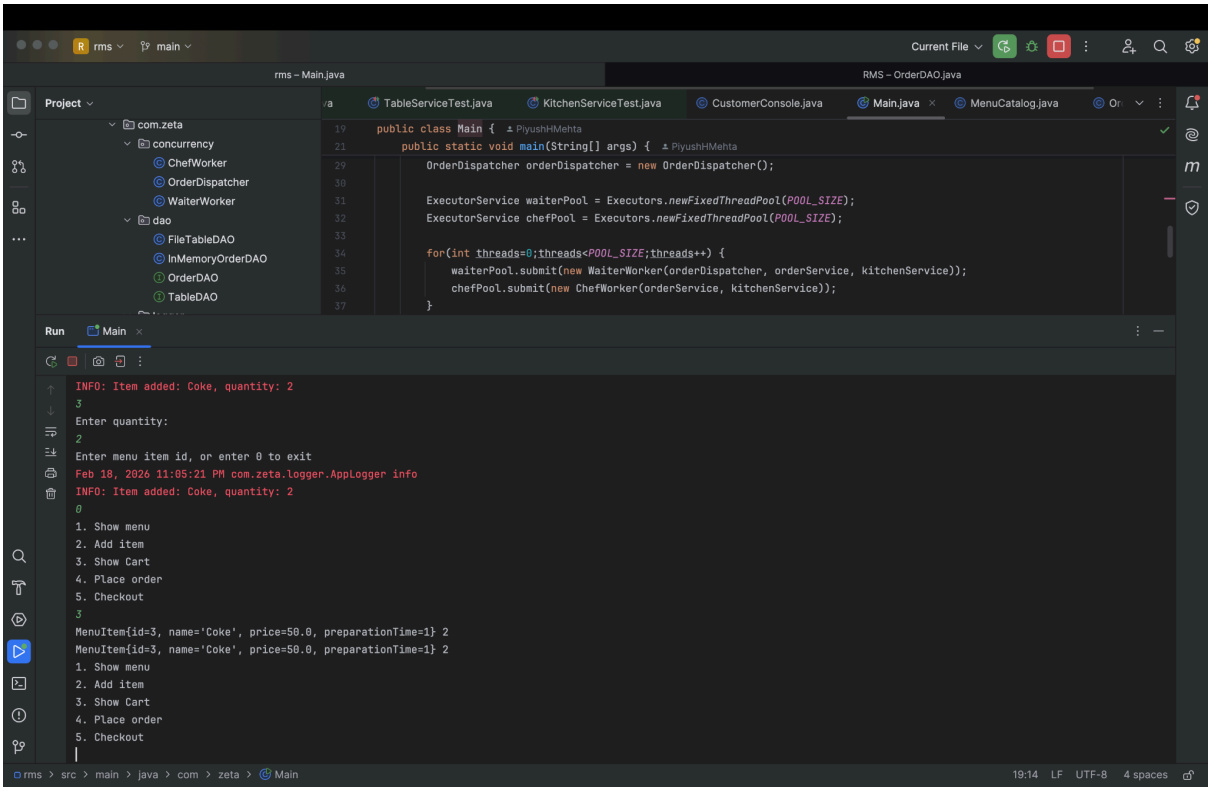

Fig 16 : Adding items along with Quantity to Cart

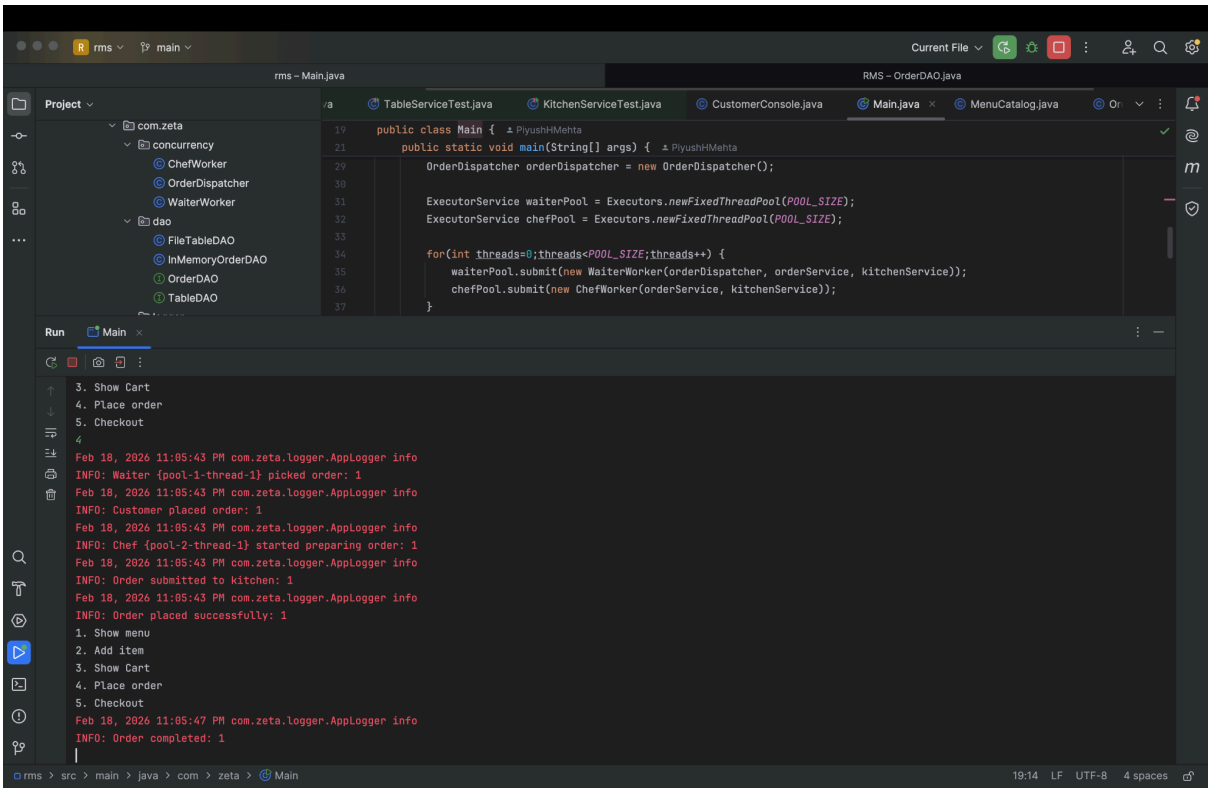

Fig 17 : The customer clicks on Place Order.

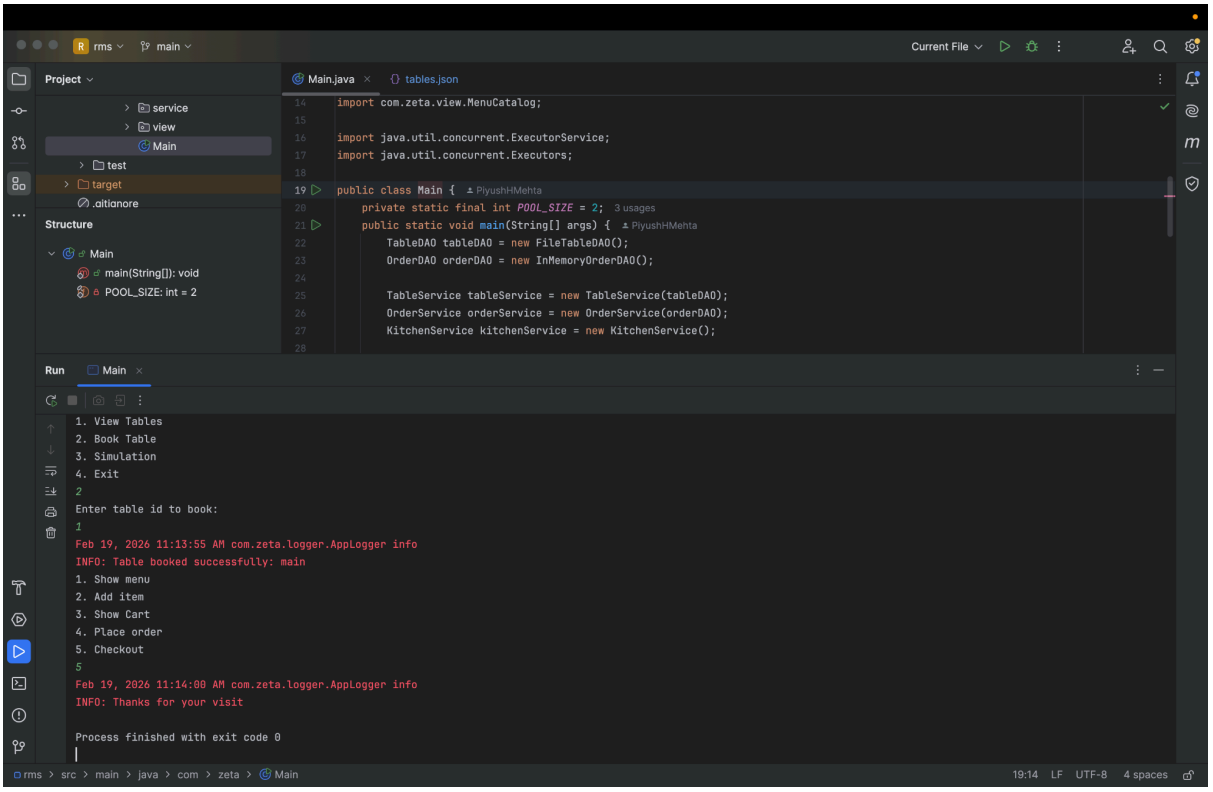

Fig 18 : Checkout

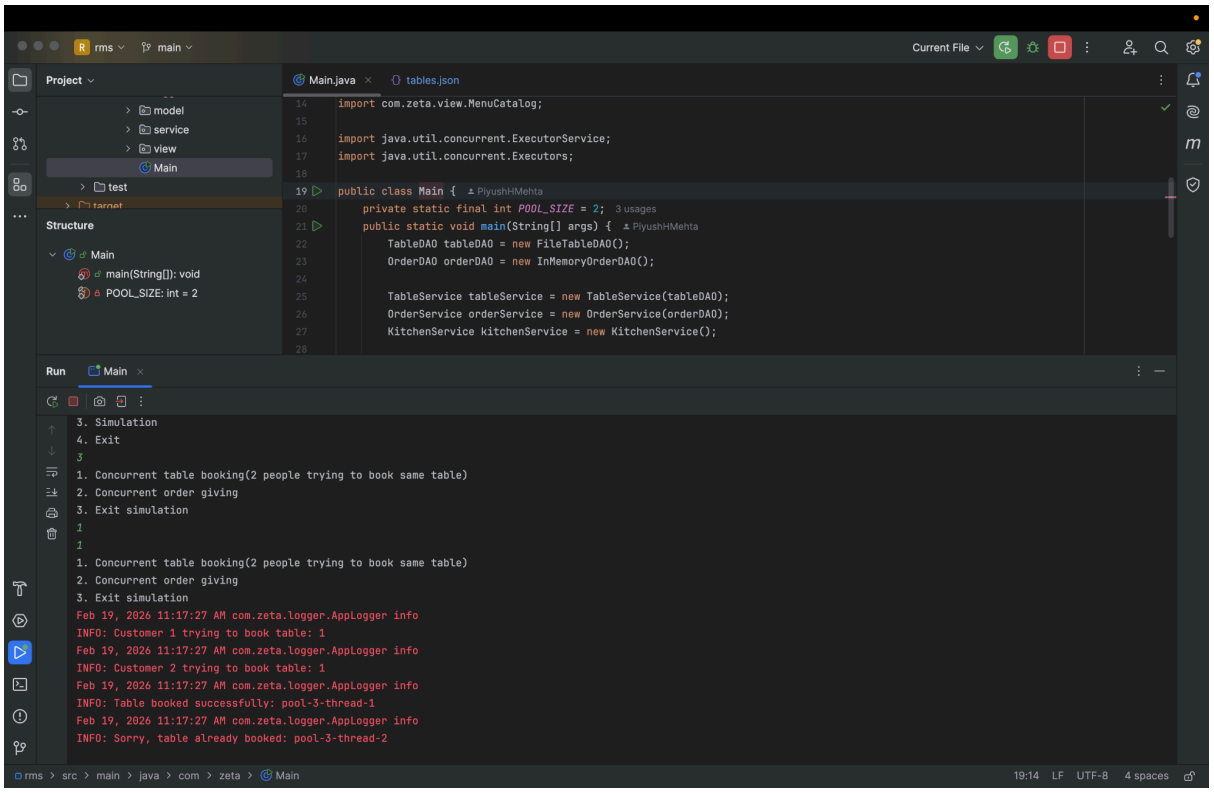

Fig 19 : Simulating Concurrent table Booking on available table

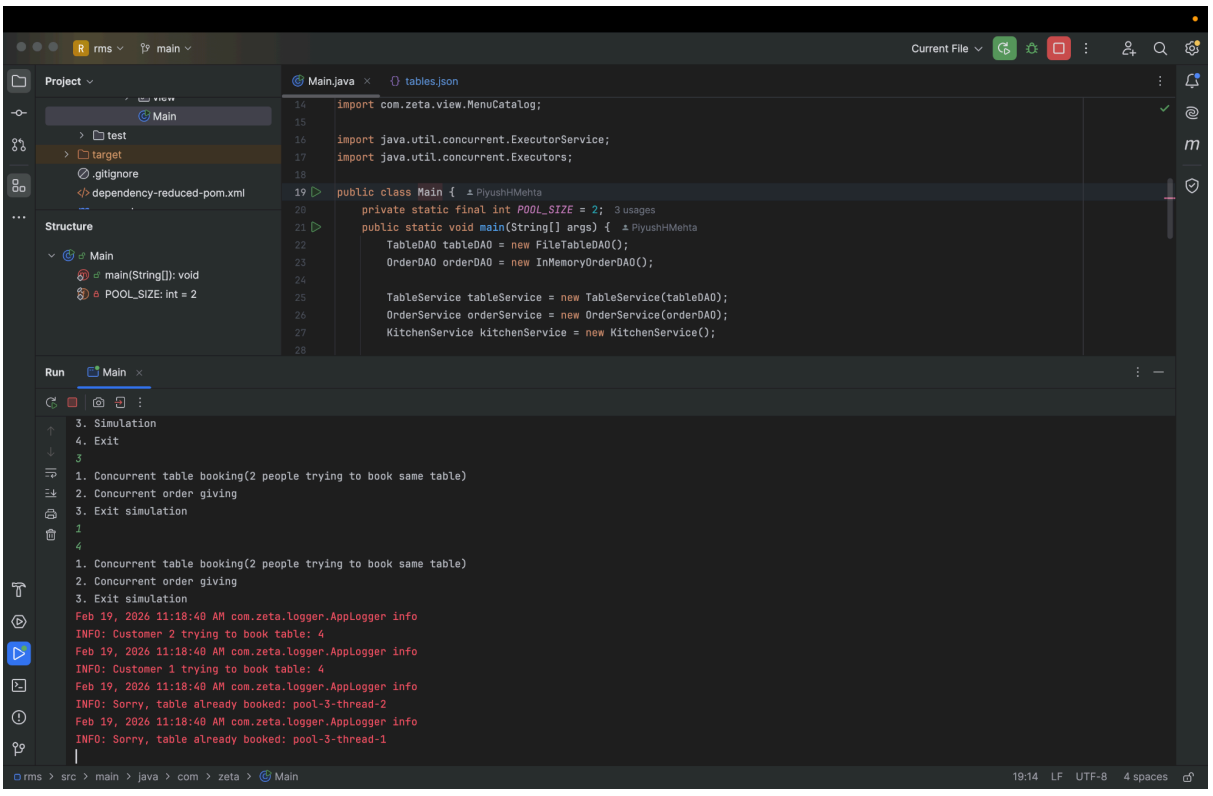

Fig 20 : Simulating Concurrent table Booking on not available table

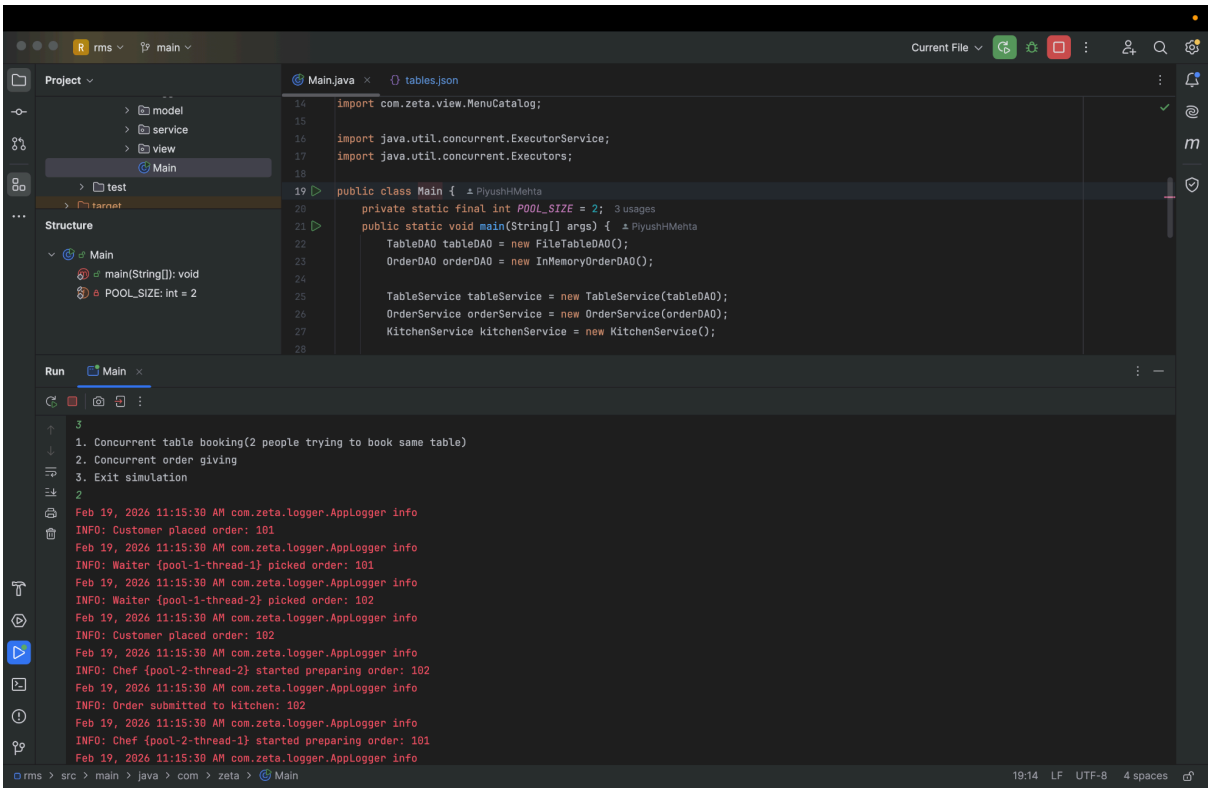

Fig 21 : Simulating Concurrent orders

Code coverage:

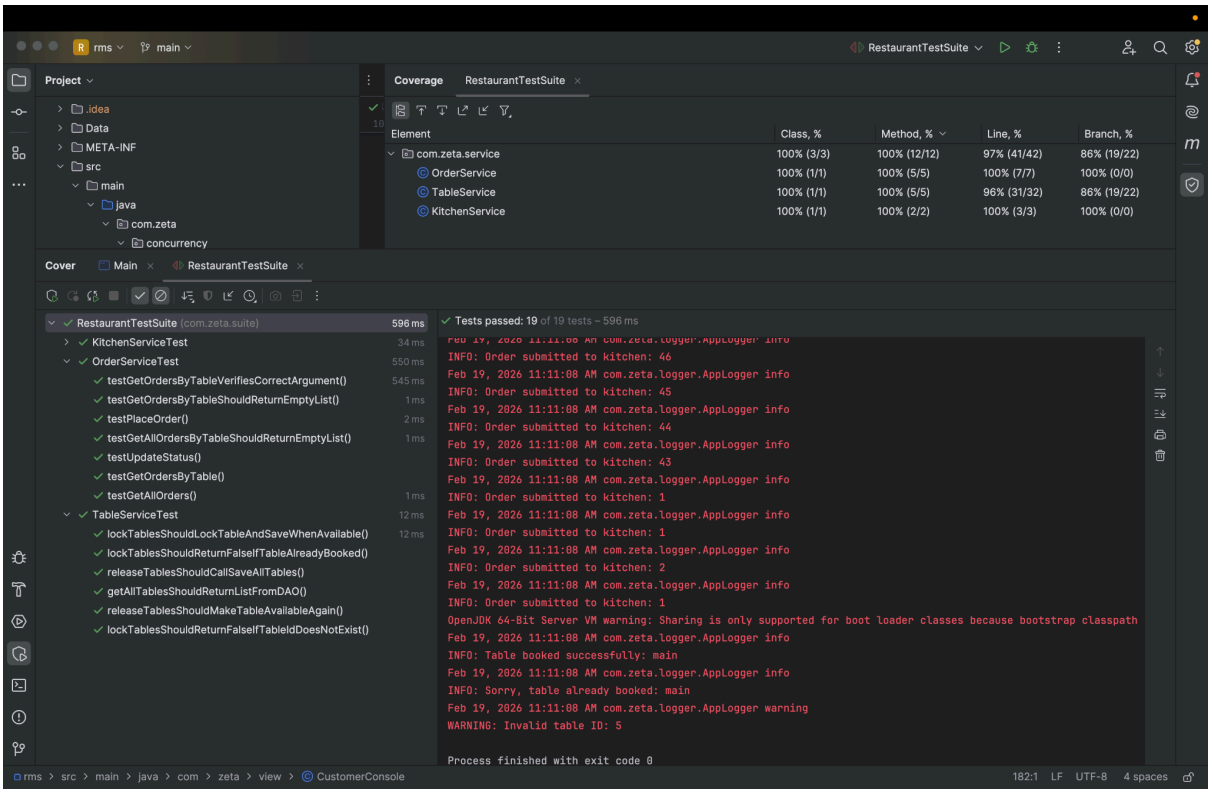

Fig 22