T3 APD Project Documentation

1. Problem Definition

Scenario:

This project simulates a **stock exchange system** where multiple brokers, represented by threads, **concurrently place buy and sell orders** on a centralised order book. The system models a simplified version of how **real-world stock exchanges** operate, focusing on how multiple brokers interact with a central order matching mechanism.

Key Components:

Shared Resources:

- **Centralised Order Book**: A shared data structure that manages buy and sell orders for multiple stocks (e.g., AAPL, GOOGL). Each stock has its own order book.
- **Order Queues:** Each stock's order book contains:
    - A **buy order queue**, prioritising orders with the **highest price**.
    - A **sell order queue**, prioritising orders with the **lowest price**.

Threads:

- **OrderProducers (Brokers):** Each **OrderProducer** represents a broker, running in its own thread and placing random buy or sell orders for a set of stocks. Multiple **OrderProducers** (brokers) run concurrently to simulate multiple brokers placing orders simultaneously.
- **OrderMatchingService**: Each OrderBook has a dedicated thread that handles matching orders in the background.

Outcome Objective:

1. Brokers can **place orders concurrently**, and the order matching mechanism will ensure correct **trade execution** based on price and time.
2. The system guarantees **safe and consistent access to shared resources** through proper locking mechanisms, ensuring **no data races or deadlocks**.

2. Design and Implementation

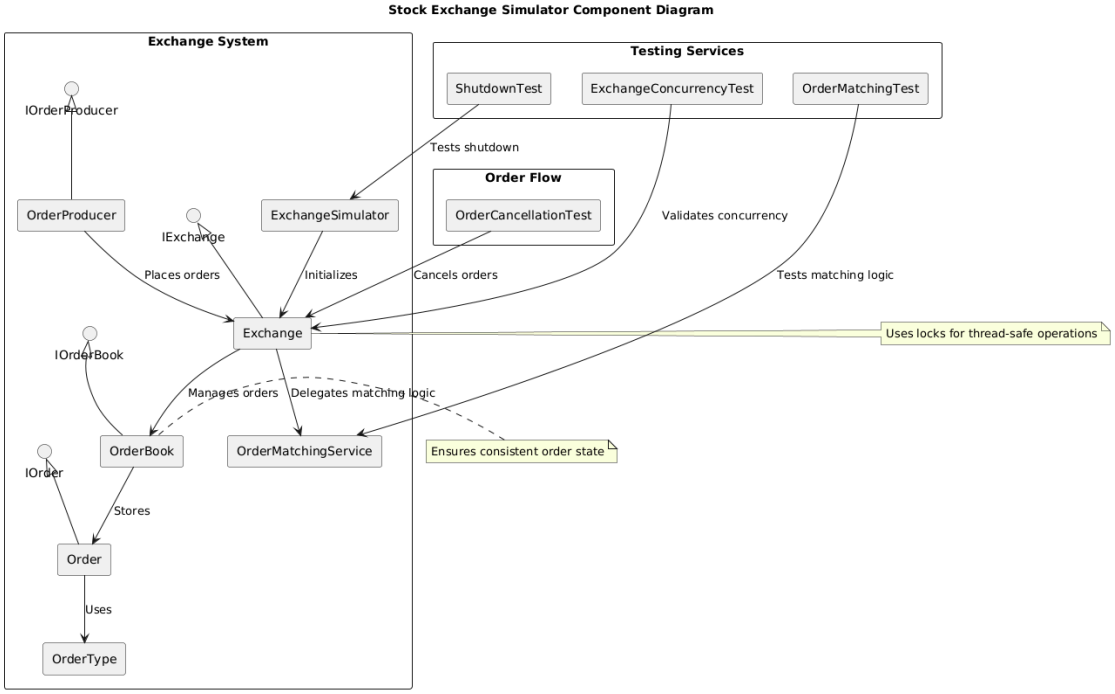
Design Overview:

The design consists of several key components that work together to handle broker interactions, order placement, and order matching:

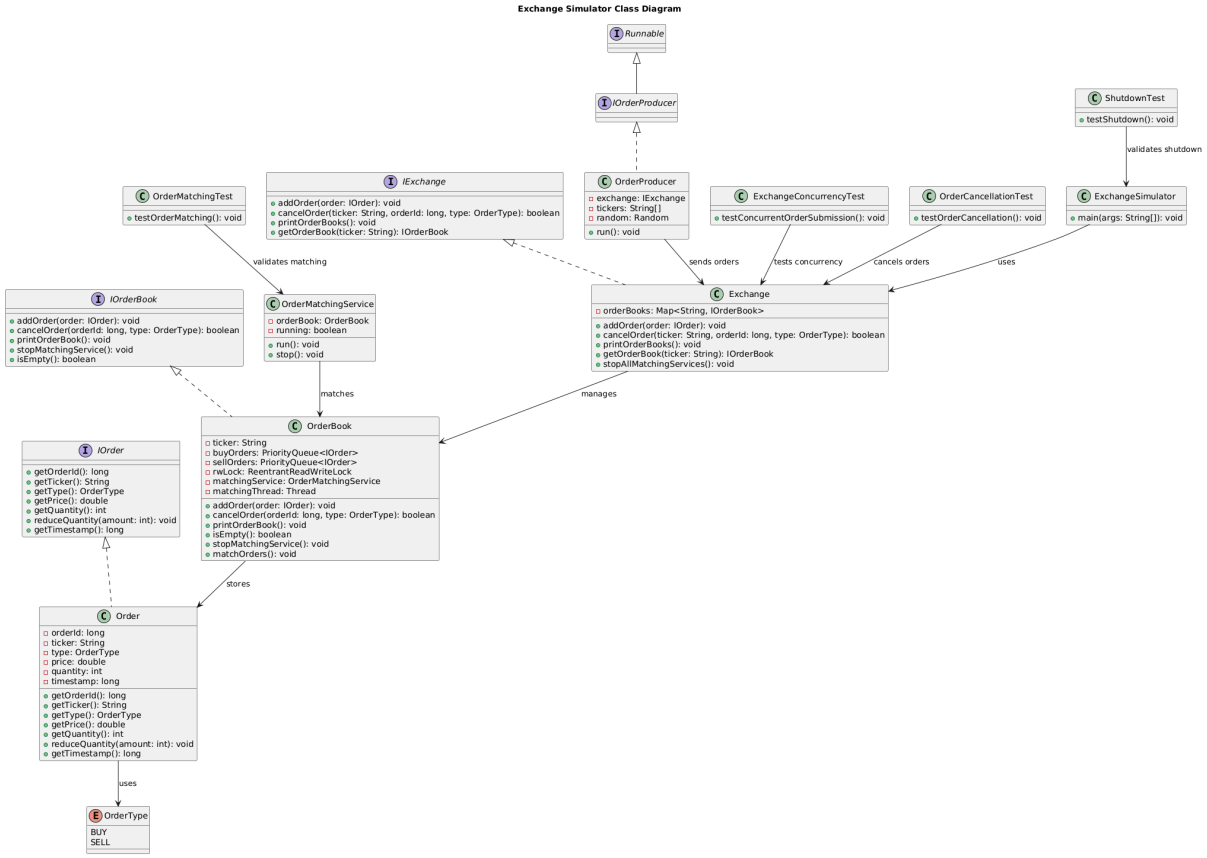
1. **OrderProducer** (Broker):
    - Simulates a broker that runs in its own thread and continuously generates random buy or sell orders.
    - Each OrderProducer places orders on different stocks until it is interrupted, simulating real-time market activity.
2. **OrderBook**:
    - Each stock has its own **OrderBook** that maintains two priority queues: one for buy orders and one for sell orders.
    - Buy orders are prioritised by the highest price, and sell orders are prioritised by the lowest price.
    - The OrderBook processes and matches orders whenever buy and sell prices align.
3. **Exchange**:
    - The Exchange manages multiple order books, each representing a stock.
    - It delegates incoming buy and sell orders to the appropriate **OrderBook** based on the stock ticker.
    - The exchange also allows brokers to cancel orders and print the current state of all order books.
4. **OrderMatchingService** (Thread):
    - This service runs in the background and matches buy and sell orders within each order book.

Diagrams

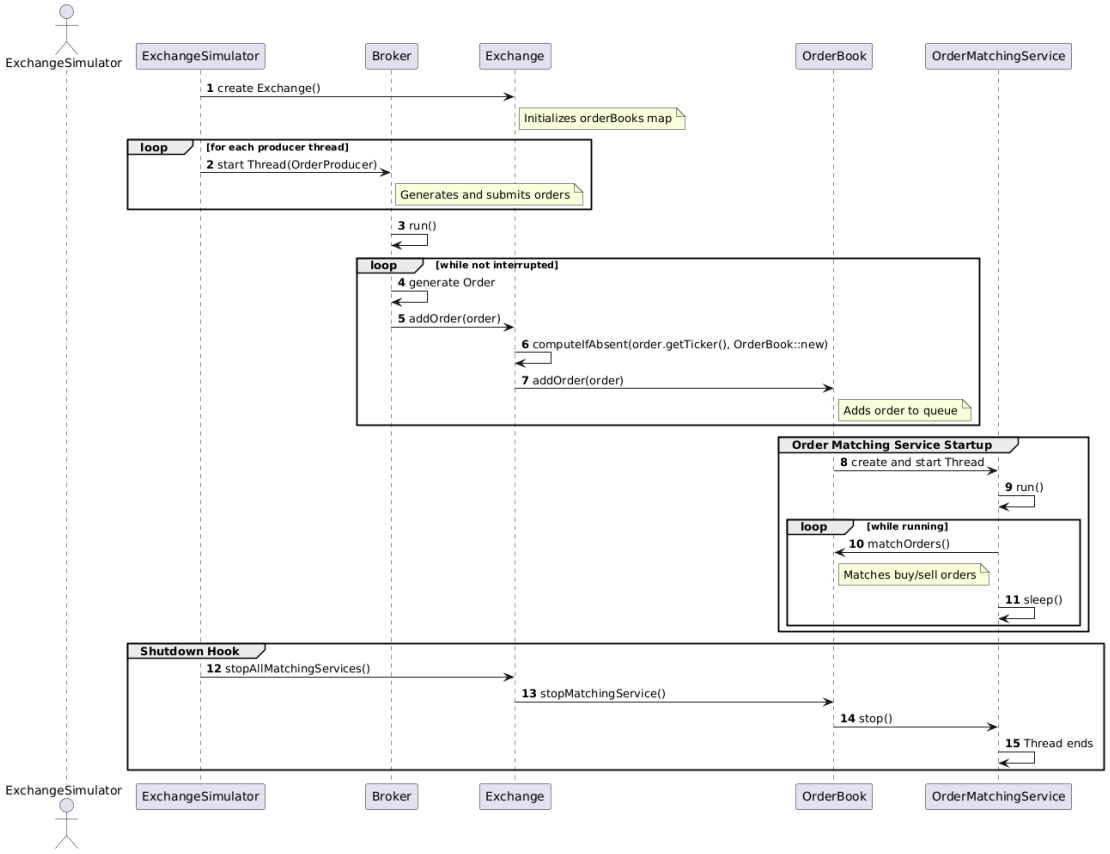
Component Diagram:



Class Diagram:



Sequence Diagram:



3. Key Design Considerations

- **Encapsulation:**
  All classes encapsulate their internal data and expose only the necessary operations (e.g., placing or cancelling orders) through **public interfaces**, following object-oriented principles to prevent unnecessary dependencies.
- **Concurrency Control:**
  The system uses **ReentrantReadWriteLock** to manage concurrent access to order books. This allows **multiple threads to read** the order book simultaneously but ensures that **only one thread can modify** it at a time, preventing race conditions.
- **Custom Thread Management:**
  Each broker (represented by an **OrderProducer**) runs as a **separate thread**, with **manual thread management** to simulate real-world concurrency.

4. Design Patterns and Best Practices

- **Interface Segregation:**
  The system adheres to the **Interface Segregation Principle (ISP)** by defining specific interfaces such as *IOrder, IOrderBook, and IExchange*, ensuring that each component only interacts with the methods relevant to it.
- **Encapsulation:**
  All classes encapsulate their **internal state** and expose **minimal interfaces**, ensuring clean interaction and better maintainability.
- **Separation of Concerns:**
  Different responsibilities are divided among classes, such as brokers for **order production**, the exchange for **order management**, and the **OrderMatchingService** for processing trades, making the system easier to understand and maintain.
- **Flexibility:**
  The design allows for **easy extension** by adding **new order types**, **matching algorithms**, or other functionality without requiring major modifications to existing code.

5. Concurrency Control

Handling Race Conditions:

- **ReentrantReadWriteLock:**
  Ensures that **only one thread** can modify the order book at a time, while allowing **multiple threads to read** from it concurrently. This prevents race conditions where simultaneous modifications could lead to **inconsistent states**.
    - **Write Lock:**
      Acquired when an **order is added, modified, or cancelled**, blocking other threads from reading or writing until the operation completes.
    - **Read Lock:**
      Acquired when the **order book is accessed without modification** (e.g., printing), allowing multiple threads to read concurrently but blocking write operations.

Atomic Operations:

- **AtomicLong:**
  The *Order* class uses **AtomicLong** to generate **unique order IDs** in a **thread-safe manner**, ensuring IDs remain consistent even under concurrent access.

Concurrent Data Structures:

- **ConcurrentHashMap:**
  The *Exchange* class manages multiple order books using **ConcurrentHashMap**. This allows **independent trading on different stocks** without blocking other operations across the exchange, ensuring efficient performance under concurrent workloads.

6. Testing and Validation

Test Cases:

1. **Concurrency Test:**
    - **Objective:** Simulate multiple brokers placing orders simultaneously.
    - **Procedure:** Run multiple *OrderProducer* threads to generate buy and sell orders concurrently for different stocks.
    - **Expected Outcome:**
        - All orders are accurately reflected in the respective order books.
        - No missing or duplicated orders occur, and data consistency is maintained.
2. **Order Matching Test:**
    - **Objective:** Ensure that buy and sell orders are matched correctly based on price and quantity.
    - **Procedure:** Place multiple buy and sell orders with varying prices and quantities, and observe the matching process.
    - **Expected Outcome:**
        - Orders are matched according to **price priority** (highest buy price, lowest sell price) and quantity.
        - Partial orders are correctly tracked with updated quantities.
3. **Order Cancel Test:**
    - **Objective:** Verify that brokers can cancel their orders and the order book updates correctly.
    - **Procedure:** Add several orders to the order book and cancel selected orders.
    - **Expected Outcome:**
        - Cancelled orders are removed, with no lingering references.
        - The remaining orders retain their correct positions in the priority queue.
4. **Shutdown Test:**
    - **Objective:** Ensure the system shuts down gracefully, with all services terminating properly.
    - **Procedure:** Start the exchange simulator and initiate shutdown (e.g., via a termination signal).
    - **Expected Outcome:**
        - All matching services and threads stop without errors or hanging operations.
        - No further orders are processed after the shutdown signal.

Validation:

- **Method:** Check the output of the order books at each stage to ensure correctness.

- **Focus Areas:** Ensure there are no **inconsistencies, race conditions, or deadlocks** during concurrent operations.
- **Final Output:** Verify that all orders were processed as expected, with accurate matching, cancellations, and a smooth shutdown.

7. Running the Application and Tests

Application run steps:

1. **Clone the Project:**

Clone the project repository from the version control system (e.g., GitHub).

2. **Build the Project:**

Build the project using Maven by navigating to the project root and running:

mvn clean install

3. **Run the Exchange Simulator:**

Run the main class ExchangeSimulator to start the simulation of multiple brokers placing orders:

mvn exec:java -Dexec.mainClass="com.ordebook.exchange.simulator"

4. **Monitor Output:**

The order book will be printed every 5 seconds, showing the current state of buy and sell orders. You should see matched orders and their details.

Test run steps:

1. **Run All Tests:**

mvn test

2. **Run a Specific Test:**
   To run a specific test (e.g., *ExchangeConcurrencyTest***), use:**

mvn -Dtest=com.orderbook.ExchangeConcurrencyTest test

3. **Monitor Output:**

The results will display in the console. Ensure all tests pass without errors.

8. Interpretation of Results

Order Matching:
The system will print matched orders, showing the quantity traded, the price, and the involved order IDs.

Concurrency Behaviour:
Multiple brokers (threads) will place orders concurrently. The locking mechanism ensures no data races occur, and the order book maintains consistency.

Final Order Book:
At the end of the simulation, you can check the final state of the order book to verify that all orders have been processed correctly.