Introduction

1. Purpose of this document

   The purpose of this document is to provide a detailed overview of the architecture of our matching engine. This document aims to illustrate how the different components and elements of the matching engine are structured and how they interact to perform their functions. By outlining the internal workings of the engine, this document will help stakeholders, interested parties gain a clear understanding of its design and capabilities.

2. A brief overview of the system functionality

   The matching engine is a real-time system designed to process and match incoming requests efficiently. It handles various types of requests, manages orders, and ensures that all entities are updated promptly. The engine continuously monitors incoming data, applies business logic, and performs necessary checks to find optimal matches.

3. Documentation Approach

   Our matching engine architecture uses a viewpoint-based approach for documentation. This method breaks down the system into different views, each focusing on a specific aspect like code structure, runtime behavior, or deployment. By doing this, we can address the concerns of various stakeholders more effectively.

4. Structure of this document

   This document provides a comprehensive overview of the matching engine architecture, focusing on three key viewpoints: Information, Concurrency, and Development.

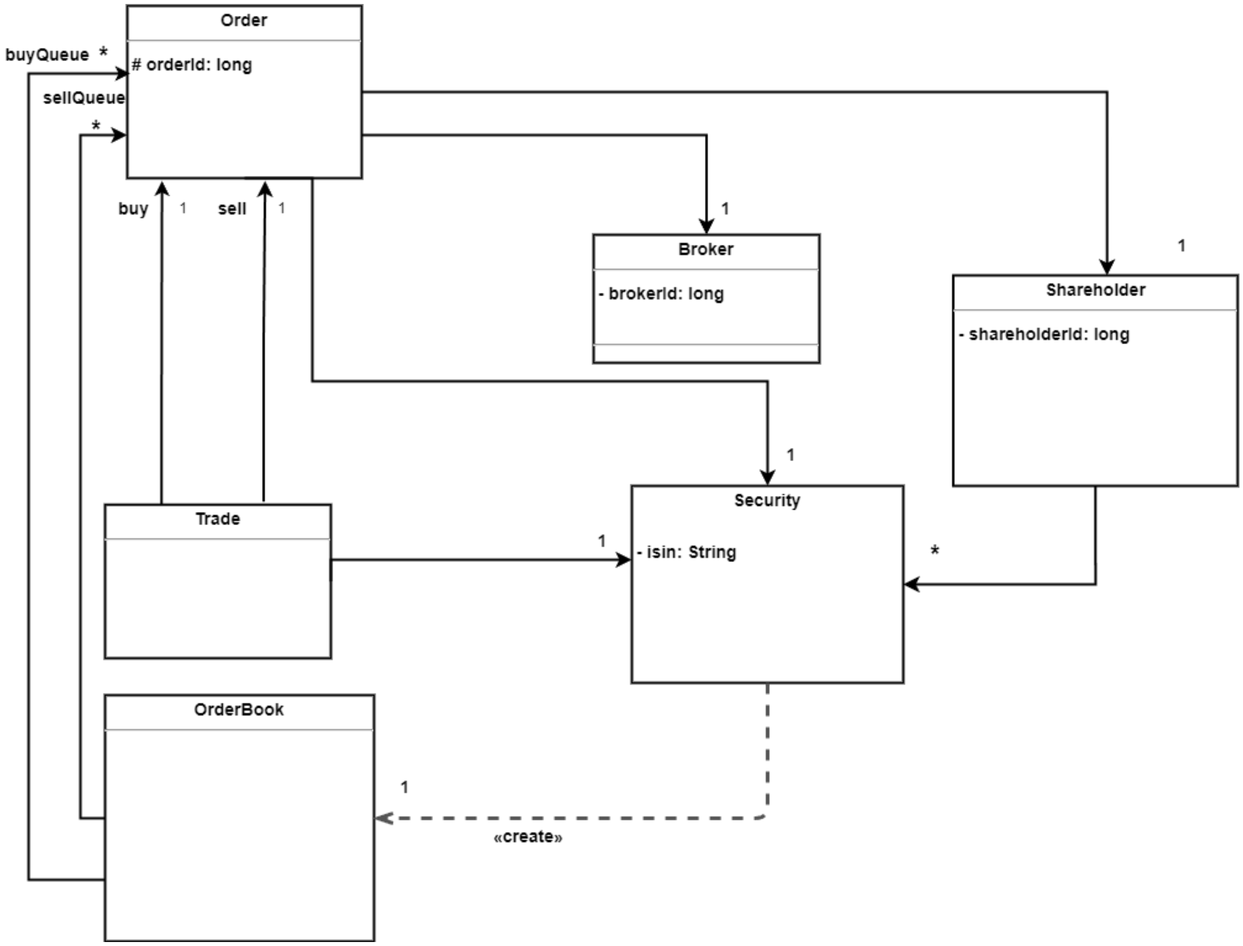
   In the **Information Viewpoint**, the architecture is detailed through class diagrams that illustrate the relationships between various entities such as OrderBook, Trade, Order, Security, Broker, and Shareholder. Key architectural decisions in this viewpoint include the approaches used for data storage, the management of entity life cycles, and the standardization of unique identifiers. These elements are critical for maintaining data integrity and ensuring the seamless flow of information across the system.

The **Concurrency Viewpoint** addresses how the system handles simultaneous operations and the interaction between different components. This viewpoint is supported by diagrams that map out component interactions and thread management. Core elements include the Matching Engine, Credit Service, and Message Queue, which work together to facilitate real-time transaction processing. Architectural decisions here focus on implementing asynchronous communication to optimize performance and responsiveness, as well as introducing a separate credit management service to ensure efficient handling of credit checks and balances.

Finally, the **Development Viewpoint** outlines the structural and technological foundations of the system. Diagrams such as package and class diagrams provide insight into the organization of code and the integration of various components. Key elements in this viewpoint include Maven modules, Spring Boot components, and JMS messaging, which together support the modularity and scalability of the application. Architectural decisions are centered on choices related to the build process, the selection of a messaging framework, and the adoption of specific design patterns and testing strategies to enhance the reliability and maintainability of the system.

Information viewpoint

In this section we decribe the way that the system stores, manipulates, manages, and distributes information



Entity	Description
Broker	A broker acts as an intermediary between the matching engine and the market players. Brokers send buy and sell orders on behalf of shareholders and are required to maintain enough credit to buy shares and accumulate credit when selling shares.

Shareholder	A shareholder is an individual or entity that owns security shares. Shareholders place buy or sell orders through brokers and must own enough shares to meet the quantity specified in sell orders.
Order	An order is a request sent by market players, via brokers, to buy or sell a specific quantity of a security. Orders have characteristics such as type (e.g., limit, iceberg, stop limit), time-in-force, and minimum execution quantity, which define their behavior in the order book.
Order Book	The order book is a data structure that maintains lists of active buy and sell orders for a specific security. Buy orders are sorted from high to low price, while sell orders are sorted from low to high price. Orders with the same price are ordered by their time of entry.
Security	A security represents a stock traded on the stock exchange, identified by an ISIN. Each security has specific properties like tick size, which determines price steps, and lot size, which dictates quantity steps. The security's order book tracks buy and sell orders for that stock.
Trade	A trade is the result of matching two orders,one buy and one sell. The trade consists of a buy and sell order, with the trade's quantity being the minimum of the two orders' quantities, and its price determined based on the matching algorithm and the characteristics of the orders.

Decisions

1) Orders in the OrderBook are maintained in two separate queues, organized first by price and then by time. The buy queue is sorted in descending order based on price, while the sell queue is sorted in ascending order based on price. For orders with the same price, the order that entered the queue earlier has higher priority.

2) We use CSV files for loading and saving data to disk because our data format is simple, and we have limited read/write operations during the program's execution. The choice

of CSV is appropriate due to its simplicity, compatibility across various platforms, and lightweight.

3) We store the data related to various entities, such as Trade, Order, OrderBook, Security, and others, in memory. This decision is made to ensure faster and easier access to these entities during the program's execution. Storing data in memory optimizes performance by reducing the time required for data retrieval compared to disk-based storage solutions.

4) The MatchingEngine is designated as the authoritative owner of data pertaining to transactions, orders, shareholders, and similar entities, ensuring centralized management and consistency within the matching process. Meanwhile, the CreditService assumes responsibility for managing data related to brokers and their credits. This separation of data ownership aligns with the respective functional domains, promoting modularity, clear boundaries, and facilitating easier maintenance and scalability of the system.

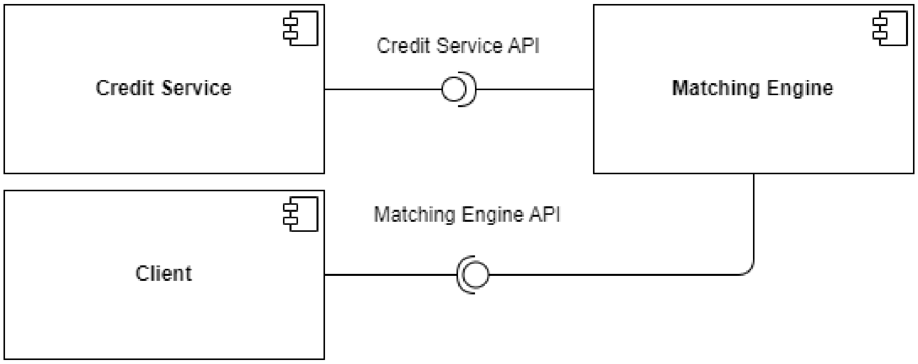
5) We have standardized unique identifiers across different entities to ensure clear and unambiguous distinction within the system. The ISIN (International Securities Identification Number) is utilized as the unique identifier for Securities, providing a globally recognized standard. For Brokers, we have defined brokerId, for Orders, orderId, and for Shareholders, shareholderId. Each of these identifiers serves to uniquely distinguish their respective entities and consistent data management throughout the application.

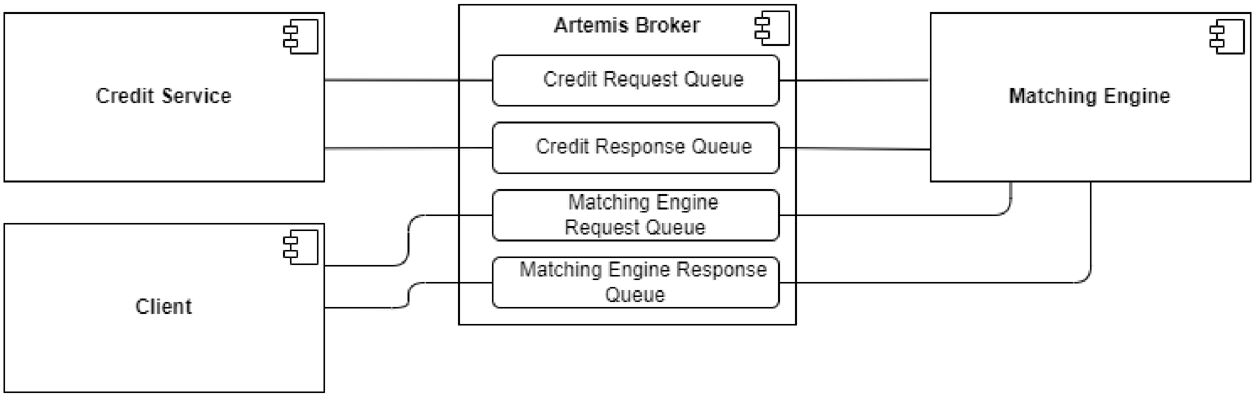
6) The system manages the lifecycle of different entities based on their importance during operations. Trades are temporary and are generated by the Matcher; they are deleted once order matching is complete, and after sending the necessary messages, keeping the system clean and efficient. Orders, created by the NewOrderProcessor, are also removed when fully executed.

On the other hand, data like brokers, securities, and shareholders are persistent, remaining available for the entire runtime of the program. The OrderBook keeps any unexecuted orders for future matching attempts. These persistent entities , brokers, securities, shareholders, and unexecuted orders can also be created and loaded into the system using the DataLoader. This approach ensures the system handles data efficiently, with temporary data removed when no longer needed and important data consistently available.

Concurrency viewpoint

In this section we describe the concurrency structure of the system and map functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled.





Component	Description

Client	Initiates requests for orders and processes the responses. It interacts with the system using the Artemis Broker queues for sending and receiving messages.
Artemis Broker	Acts as a message broker that handles communication between different components. It includes several specific queues to manage different types of requests and responses:
Credit Request Queue	A queue within the Artemis Broker that receives credit check requests from the Matching Engine to the Credit Service.
Credit Response Queue	A queue within the Artemis Broker that sends the response of the credit check from the Credit Service back to the Matching Engine.
Matching Engine Request Queue	A queue within the Artemis Broker that receives order requests from the Client and sends them to the Matching Engine.
Matching Engine Response Queue	A queue within the Artemis Broker that sends the order response from the Matching Engine back to the Client.
Matching Engine	Processes the orders received from the Client and interacts with the Credit Service to manage brokers' credit by sending and receiving credit requests and responses.
Credit Service	Responsible for managing and verifying the credit status of brokers. It handles credit-related operations to ensure that brokers have the necessary credit balance for transactions, maintaining the integrity of credit information across the system.

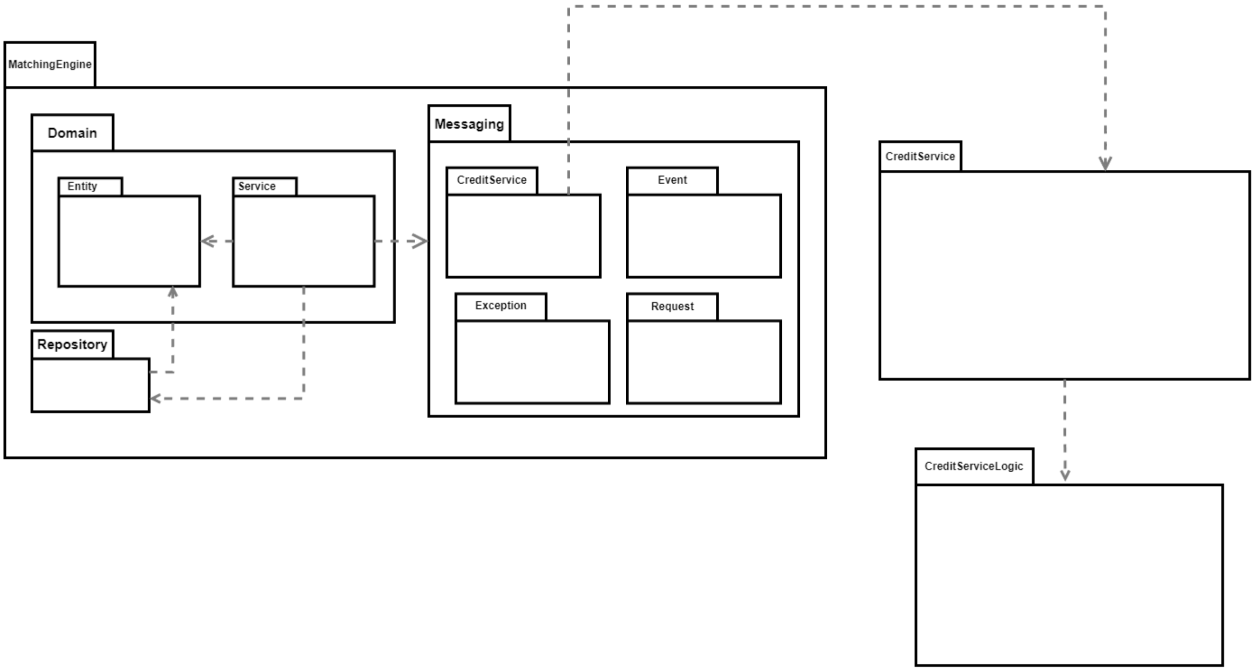
1) Broker credit management is handled by a separate service that operates independently from the matching engine. This separation allows each component

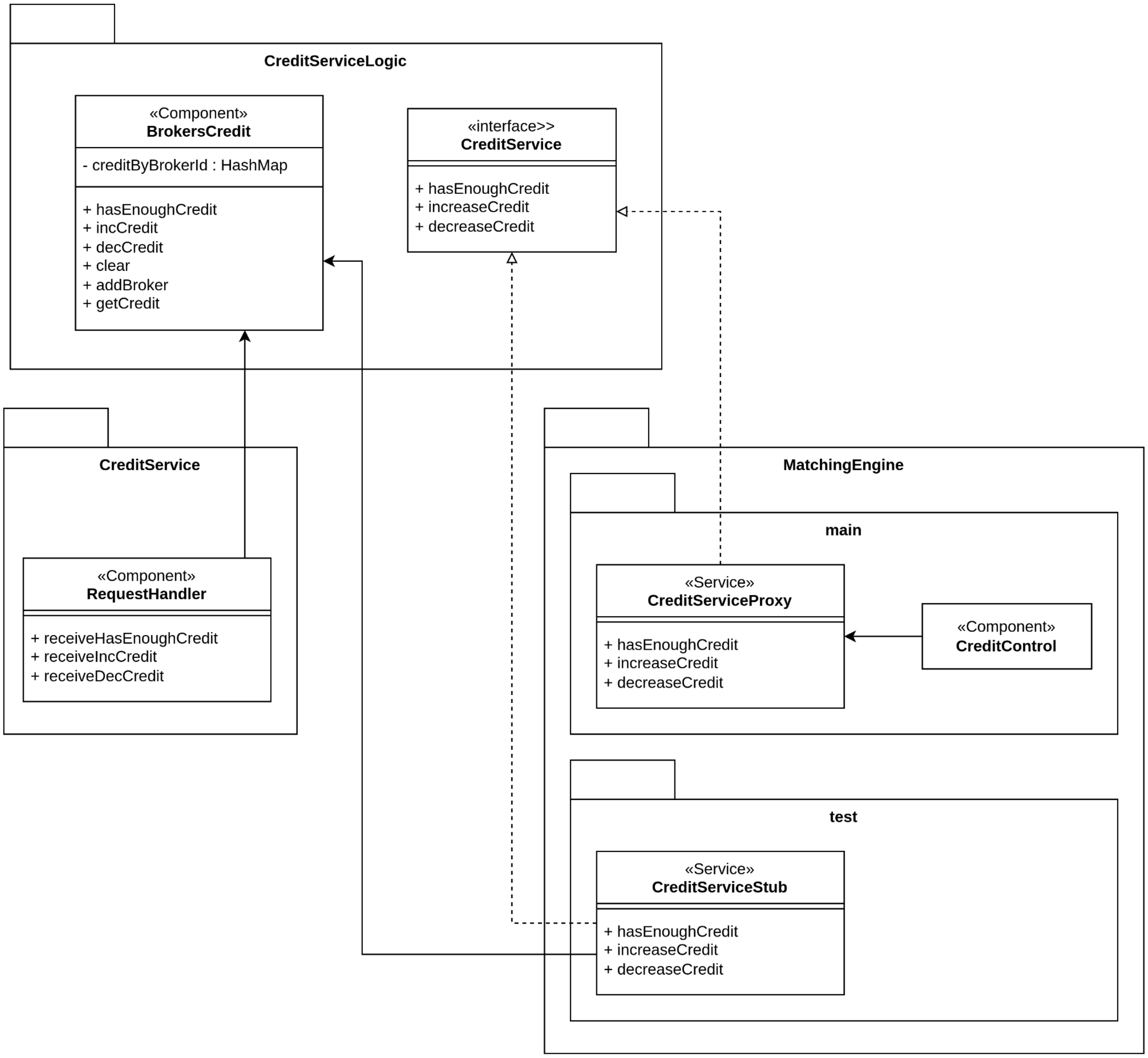
to specialize and perform its role more effectively without interfering with one another.

2) The credit service communicates with the matching engine using a message queue. This setup ensures that interactions are asynchronous and decoupled, allowing both components to exchange information efficiently without being directly dependent on each other.

Development viewpoint

In this section we describe the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.





Package	Description
MatchingEngine	Central package that manages the main functions of the system.
Domain	Holds the core business rules and data structures.

Entity	Defines the main data objects or business entities.
Service	Offers business services and performs operations on entities.
Repository	Manages saving and retrieving data.
Messaging	Handles communication between different parts of the system.
CreditService	Manages credit-related operations.
Event	Handles event-driven workflows, defining the structure and handling of system events.
Exception	Manages errors and custom exceptions.
Request	Represents incoming requests or commands to the system.
CreditServiceLogic	Contains specific rules and logic for handling credit operations.

Decisions

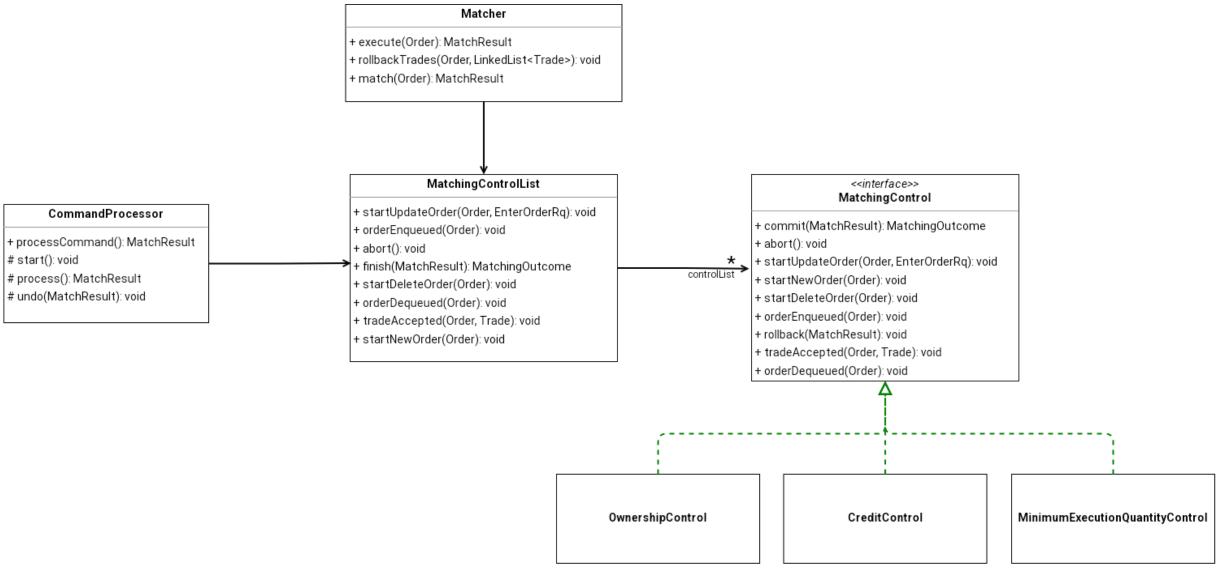
1) We have chosen to use Maven for the build process and project dependency management. Maven simplifies the build lifecycle by providing a standardized way to compile, package, and deploy the application, ensuring consistency across different environments. Additionally, Maven's dependency management features allow us to efficiently handle external libraries and modules, automatically resolving and integrating them into our project, which reduces overhead.

By utilizing Maven's multi-module capabilities, we can organize our application into several sub-projects under a single parent project, enhancing modularity and separation of concerns. In our software, this approach allows us to manage modules such as `MatchingEngine`, `CreditService`, and `CreditServiceLogic` effectively. Each of these modules can be developed, tested, and deployed independently while maintaining a coherent structure and shared configuration across the entire project.

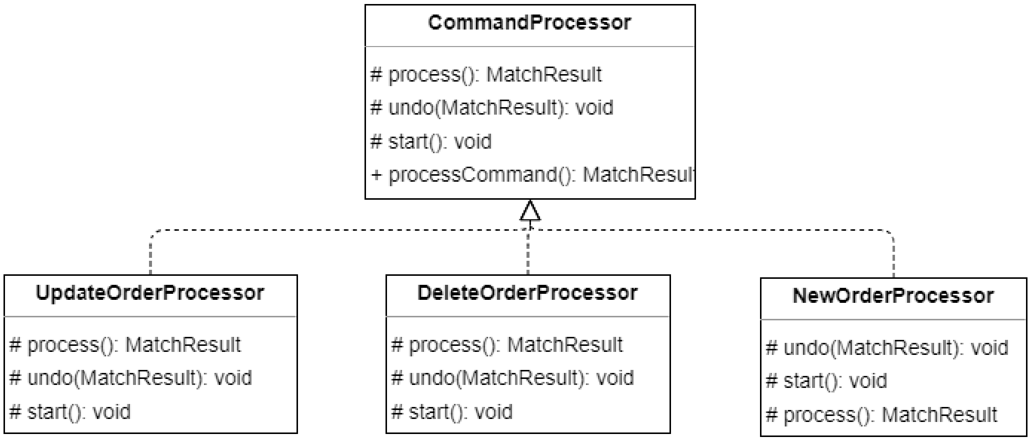
2) We have decided to use JMS (Java Message Service) for messaging within our architecture to enable reliable, asynchronous communication between different components of the system. JMS provides a robust framework for message exchange, ensuring that messages are delivered securely and in the correct order, even in distributed environments. For message serialization, we will use Jackson to convert Java objects to JSON format and vice versa. Jackson is a widely-used, efficient library that ensures seamless serialization and deserialization

3) Spring Boot is used to simplify the setup and configuration of our application. It streamlines the development process by offering a comprehensive framework with pre-configured settings and defaults, which reduces the need for extensive boilerplate code. This approach accelerates development, improves productivity, and ensures a consistent and scalable application structure.

4) We have a list of classes in Matcher, all of which implement the MatchingControl interface. By invoking the methods of MatchingControl on each of them, the relevant conditions for execution are checked, and changes are applied to the system accordingly.


Matcher
+ execute(Order): MatchResult
+ rollbackTrades(Order, LinkedList<Trade>): void
+ match(Order): MatchResult
CommandProcessor
+ processCommand(): MatchResult
# start(): void
# process(): MatchResult
# undo(MatchResult): void
MatchingControlList
+ startUpdateOrder(Order, EnterOrderRq): void
+ orderEnqueued(Order): void
+ abort(): void
+ finish(MatchResult): MatchingOutcome
+ startDeleteOrder(Order): void
+ orderDequeued(Order): void
+ tradeAccepted(Order, Trade): void
+ startNewOrder(Order): void
*
controlList
<<interface>>
MatchingControl
+ commit(MatchResult): MatchingOutcome
+ abort(): void
+ startUpdateOrder(Order, EnterOrderRq): void
+ startNewOrder(Order): void
+ startDeleteOrder(Order): void
+ orderEnqueued(Order): void
+ rollback(MatchResult): void
+ tradeAccepted(Order, Trade): void
+ orderDequeued(Order): void
OwnershipControl
CreditControl
MinimumExecutionQuantityControl

5) The matching engine uses a CommandProcessor class to handle orders in a standard way. This class sets up a step-by-step process for all types of orders (update, delete, new). It starts the process, does the main work, and can undo changes if needed. Each specific type of order (like updating or deleting) has its own version of this process, but they all follow the same basic steps. This makes it easy to add new types of orders and keeps the system organized. It also helps catch and fix errors in a consistent way for all orders.



6) All input and output data exchanged via the message queue within the system are logged using Java Logger. This logging mechanism ensures that every message sent to or received from the queue is recorded, providing a detailed audit trail for monitoring, debugging, and troubleshooting purposes.

7) We have selected JUnit Jupiter as our testing framework and AssertJ for assertions in our testing strategy. JUnit Jupiter provides a modern, feature-rich platform for writing and running tests, offering flexibility and support for both standard and parameterized tests.

8) Our testing strategy involves using in-memory objects instead of external data sources. This approach allows tests to run quickly and consistently, without dependencies on external systems, such as databases or file systems.

9) We have implemented the Builder pattern, leveraging Lombok's @Builder annotation, for constructing Security, Broker, and other entities in our system. This design choice facilitates the creation of instances of these entities in a clear, concise, and flexible manner, ensuring that all required fields are set while maintaining immutability.

10) Our testing approach focuses on validating and verifying the behavior of modules rather than their internal structure. This strategy emphasizes testing the functionality and outputs of the modules to ensure they meet the specified requirements and perform as expected in various scenarios.

11) Services that interact with the message queue are required to implement specific interfaces for both receiving and sending messages. This design ensures a consistent and standardized approach to message handling across different services, promoting interoperability and modularity within the system.