Nichada Wongrassamee

Trade Engine

1. General Topic

This project involves designing and implementing a real-time statistics generator for a distributed trading engine system. The system is intended to manage and analyze financial data across multiple global financial centers including New York, London, and Tokyo. By leveraging a combination of Enterprise Integration Patterns (EIP) and Design Patterns, the system will handle vast streams of financial data, such as stock tic data, and provide valuable statistical insights to facilitate trading decisions. This setup aims to enhance trading strategies through advanced data analysis and real-time information dissemination across different markets.

2. Problem Statement

Financial markets are dynamic and highly volatile environments where trading decisions must be made swiftly and based on the most accurate and up-to-date information available. Traders and financial analysts often struggle with the large amount of data generated from multiple markets around the world. Hence, there is a need for a system that can handle the scale and speed of data while also transforming this data into statistical insights efficiently. The challenge lies in collecting, processing, and analyzing this data to produce real-time statistical outputs such as minimum, maximum, average, standard deviation, and more complex metrics like regression and moving averages.

Furthermore, the system must be capable of supporting multiple trading engines [Polling Consumers] which may have different statistical needs [Strategies] based on the specific stocks or financial instruments they monitor. Each trading engine should be able to subscribe to updates on just the specific stocks it requires [PubSub], without being overwhelmed by irrelevant data. This necessitates a highly efficient, scalable, and flexible data routing and processing architecture. The solution must effectively manage data integrity and accuracy, ensure timely updates, and support various trading strategies implemented by different engines in diverse locations. The ultimate goal is to empower trading engines with the ability to make informed decisions rapidly by providing them with precise, computed statistical data tailored to their operational needs.

3. EIP Patterns

- **PubSub Channel** (@subscriber)
    - Usage : Broadcasts new stock tick data to various trade engines, such as Tokyo, London, and New York engines, based on their subscriptions (e.g., IBM, MSFT).
    - Benefits : Ensures efficient, simultaneous delivery of real-time data updates to multiple subscribers, crucial for timely trading decisions.
- **Point-to-Point Channel**
    - Usage : Enque and Deque each tic message sequentially
    - Benefits : Maintains the integrity and order of time-based transactions, ensuring that messages are processed exactly as sent.
- **Message Translator** (@subscriber)
    - Usage : convert types from csv string to float data for further calculation.

    - Benefits : data must be in float to perform calculation
- **Message Routers** (@consumer router)
    - Usage : Filtering tic message to the appropriate stock channel based on stock Name.
    - Benefits : Enhances system modularity by organizing stocks into dedicated channels, simplifying subscription management for trade engines.
- **Invalid Message Channel** (@consumer router)
    - Usage : filter out unreadable or improperly formatted tic message to be treated separately in other channel
    - Benefits : will not confuse the consumer down the path
- **Content Enricher**
    - Usage:
      Incoming Message: Augments incoming messages with computed statistical data such as minimum, maximum, average, variance, and standard deviation.
    - Benefits: Provides enriched data that is essential for further analysis, enhancing the decision-making process in subsequent stages.

4. **Design Patterns**

- **Strategy**
    - Usage : Each trade engine updates stock statistics differently; for example, the Tokyo Trade Engine might present simpler or different statistics compared to the London Trade Engine.
    - Benefits : flexibility to change the statsupdate() modes without altering the clients that use them
- **Singleton**
    - Usage : Ensures a single instance of the Report Engine across the application.
    - Benefits : for consistency.
- **Iterator**
    - Usage : for iterating through each trade engine's stock collection to create report.
    - Benefits : Can access stocks without having to concern about the data structure.
- **Template**
    - Usage: Trade Engine Template that provides a skeleton for subclasses `TokyoTradeEngine`, `LondonTradeEngine`, and `NewYorkTradeEngine`.
    - Benefits : promotes code reuse, reduces duplication, and provides a flexible way to accommodate implementation of engine-specific behaviors.

5. **Anticipated deliverable**

Producer :

- ProducerRoute.java

Consumer :

Camel

- OrderRouter.java

- ConsumerProcessor.java

Stockstats

- StockStats.java : Calculate statistics

Subscriber :

Trade Engine :

- TradeEngine.java (Template - abstract class)
  - TOKTradeEngine.java (concrete sub class)
  - LDNTradeEngine.java (concrete sub class)
  - NYCTradeEngine.java (concrete sub class)
- SubscriberProcessor.java

StockStrategy

- PremiumStats.java
- SimpleStats.java
- StockStrategy.java

Finiterator

- Aggragate.java (Abstract Aggregate)
- StockStatsAggregate.java (Concrete Aggregate)
- Iterator.java (Abstract Iterator)
- CustomIterator.java (Concrete Iterator)
- LinkedList.java (data structure for StockStatsMap collection)

Report Engine Singleton

- Print report for each Trade Engine

Stockstats

- StockStats.java : Class that stores statistics data like min, max, ave, etc.
- StockStatsMap.java : Class that stores stockstats, it implements a linked list for data structure.

Main

- Main.java : run the subscriber program and print report of each trade engine.

Total : 21 classes
Total : 21 sourcefiles

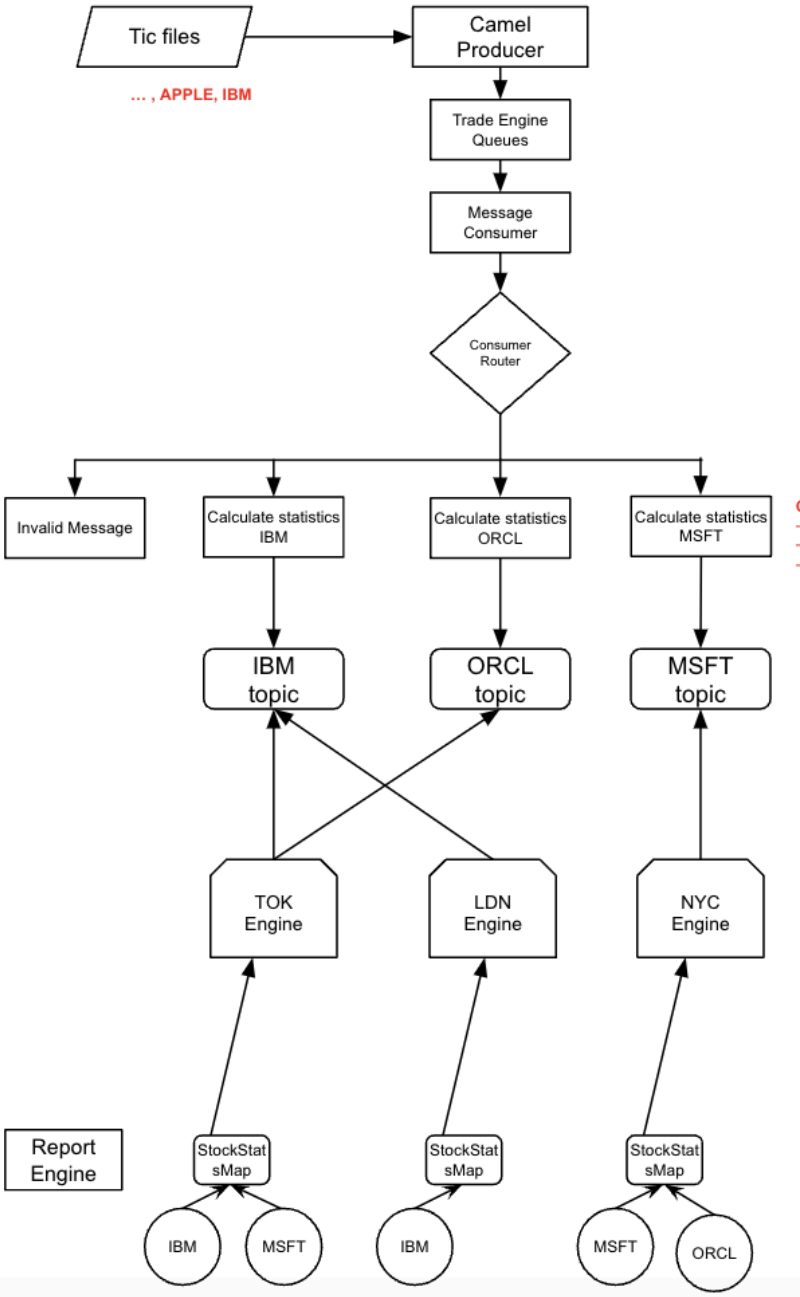
Practical Implementation in a Trading System

- Producer and Consumer Setup:
  - ProducerRouter: Manages data input from `data/inbox` to a queue, ensuring that the OrderRouter exclusively consumes these messages.
  - OrderRouter: Implements content routing to classify and publish stocks to specific topics (IBM, MSFT, ORCL) and handles unmatched messages via an Invalid Message Channel.
- Subscriber Design:

        - Trade Engine Subscription: Each engine (e.g., Tokyo, London) subscribes to specific stock topics based on configured interests, ensuring they receive relevant updates.
- Statistical Updates
    - StockStats and ConsumerProcessor: Perform computations on incoming data to update stock statistics, employing design patterns like Strategy for different statistical models.
- Reporting:
    - Employ Singleton for a centralized reporting mechanism.

Nichada Wongrassamee

6. Drawing (UML preferred) of solution



Enterprise Design :

Camel Endpoint
: enque messages

1. P2P message
: producer - consumer

2. Message Translator
Convert csv string to float data

3. Message Enricher
Calculate Stats and put into topic

4. Content Based Router

5. Invalid Message

Calculate:
- Min
- Max
- Std

6. Pub-Sub Channel
: Trade Engine (Topic)
Broadcast to engine that subscribes the stocks

Design Patterns :

1. Strategies
: Each concrete Trade Engine class apply simple strategy or premium strategy.

2. Trade Engine Template
TOK engine, LDN engine and NYC engine are concrete sub classes (Subscribers)

3. Report Engine
Singleton : Only one report engine for the entire application

4. Iterator
: iterate through portfolio to print report

Nichada Wongrassamee

7. **Tech Stack**

- Apache Camel : define routing and mediation rules. Provides interfaces for EIP messaging design pattern. Ie. Camel context and camel jms.
- Java : implement OO classes. Its portability across multiple platform makes its suitable for this project.
- JMS for messaging : standard API that allows application asynchronous messaging ie. allowing trade engines to subscribe to specific stock updates without being directly connected to the message producers.
- Eclipse IDE : manage java project, easy navigation and debugging.

8. **Things I learn**

Improvement:

- Would love to add more functionality including adding more types of financial instruments apart from stocks like bonds.

Challenge:

- Designing and deciding on trade off between scalability and simplicity

Improvements:

- Enhancing the system's scalability or performance.
- Adding more features like real-time alerts based on specific stock performance criteria.
- Connect API to read real life stock ticks