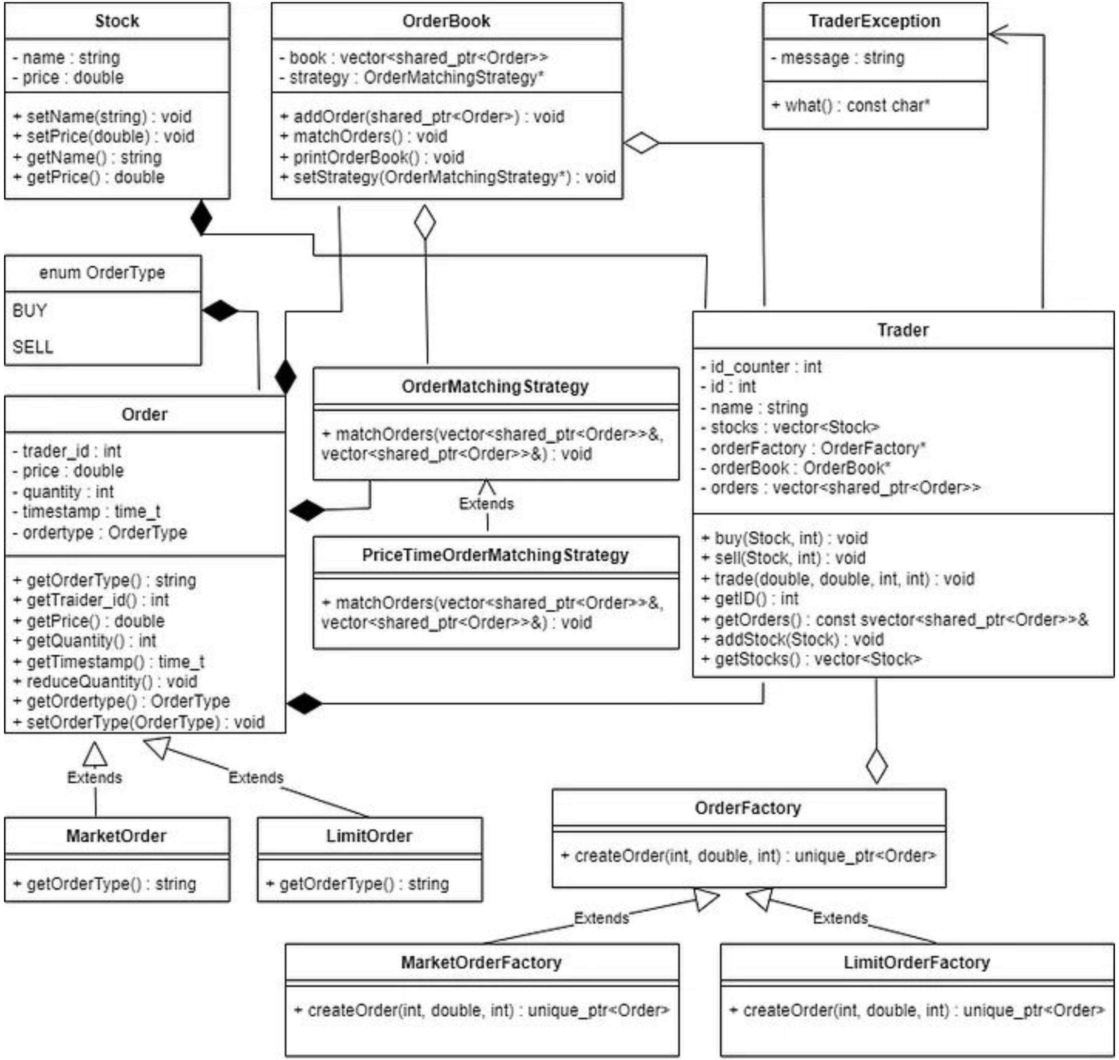
High Performance Stock Trading System

CS205 Course Project Proposal

Group 8

Overview

- Simulation of realistic stock trading platform
- Core functionalities:
    - Receive, put, and call orders
    - Access stock status
    - Update orderbook according to the received requests
    - Match buy and sell orders according to order types (e.g. market/limit) and specified strategies
    - Execute matched orders and update orderbook and stock status accordingly



Stock

- name : string
- price : double

+ setName(string) : void
+ setPrice(double) : void
+ getName() : string
+ getPrice() : double

OrderBook

- book : vector<shared_ptr<Order>>
- strategy : OrderMatchingStrategy*

+ addOrder(shared_ptr<Order>) : void
+ matchOrders() : void
+ printOrderBook() : void
+ setStrategy(OrderMatchingStrategy*) : void

TraderException

- message : string

+ what() : const char*

enum OrderType

BUY

SELL

Order

- trader_id : int
- price : double
- quantity : int
- timestamp : time_t
- ordertype : OrderType

+ getOrderType() : string
+ getTraider_id() : int
+ getPrice() : double
+ getQuantity() : int
+ getTimestamp() : time_t
+ reduceQuantity() : void
+ getOrdertype() : OrderType
+ setOrderType(OrderType) : void

OrderMatchingStrategy

+ matchOrders(vector<shared_ptr<Order>>&, vector<shared_ptr<Order>>&) : void

PriceTimeOrderMatchingStrategy

+ matchOrders(vector<shared_ptr<Order>>&, vector<shared_ptr<Order>>&) : void

Trader

- id_counter : int
- id : int
- name : string
- stocks : vector<Stock>
- orderFactory : OrderFactory*
- orderBook : OrderBook*
- orders : vector<shared_ptr<Order>>

+ buy(Stock, int) : void
+ sell(Stock, int) : void
+ trade(double, double, int, int) : void
+ getID() : int
+ getOrders() : const svector<shared_ptr<Order>>&
+ addStock(Stock) : void
+ getStocks() : vector<Stock>

MarketOrder

+ getOrderType() : string

LimitOrder

+ getOrderType() : string

OrderFactory

+ createOrder(int, double, int) : unique_ptr<Order>

MarketOrderFactory

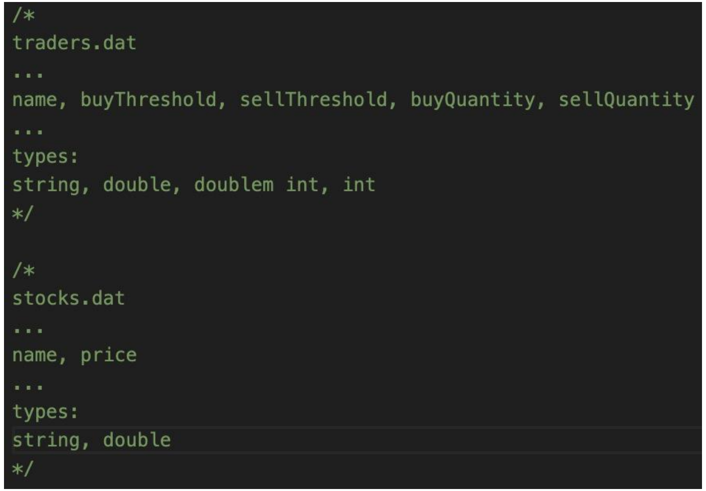
+ createOrder(int, double, int) : unique_ptr<Order>

LimitOrderFactory

+ createOrder(int, double, int) : unique_ptr<Order>

Simulated Trading Data

- Data Generation
  - We wrote a script that can generate large datasets of traders and stocks with varying quantities and complexities
  - Ensures a high degree of randomness, essential for simulating realistic financial market behaviors.
  - **Traders**
    - unique names & randomized financial thresholds and quantities, to simulate diverse market participant behaviors
  - **Stocks**
    - unique tickers and prices, using uniform distribution to model market price variations realistically.
  - avoids the complexities and unpredictabilities of real-time financial data, such as network latencies and data inconsistencies
  - prioritizes the creation of a controlled, predictable environment for reliable HPC benchmarking and testing
  - stocks number from **5k to 500k**, so size from **2*5K to 2*500K**, and traders number from **1K to 14M**, so size from **5*1K to 5*14M**



```
/*
traders.dat
...
name, buyThreshold, sellThreshold, buyQuantity, sellQuantity
...
types:
string, double, doublem int, int
*/

/*
stocks.dat
...
name, price
...
types:
string, double
*/
```

Limitations of Sequential Implementation

Suppose we have N stocks, T traders, and M orders in the system:

- Operations done by each trader:
  - Retrieve a shared list of stocks: O(N * T)
  - Put orders into orderbook: O(N * T)
- Match and execute orders: O(M^2)
- Update the orderbook, the stocks, and the user accounts: O(M+N+T)

The upper bound of M is also O(N * T)

Performance would degrade when these numbers are large.

Parallelization (Part 1)

- Order Creation Process
  - User access to current stock status can also be performed in parallel
  - Putting and calling orders can be read in parallel and sorted in the orderbook
- Order-Matching Process
  - Order-Level Parallelization **(OpenMP, MPI)**
    - Maybe partially parallelized depending on matching strategy
    - Unordered requests can be segmented into chunks by timestamp and handled separately by each thread
    - Final ordering w.r.t timestamp chunks needs to be done in serial manner

Parallelization (Part 2)

- Execution (Update) Process
  - Stock-Level Parallelization **(MPI)**
    - Potentially fully parallelized
    - Different stocks are independent, orders w.r.t each stock type can be totally separated and updated in parallel
  - User-Level Parallelization (*bonus*: if bandwidth allowed) **(MPI)**
    - Potentially fully parallelized
    - Keep track of the income and spend amount of each user (i.e., user accounts)
    - Putting and calling results can be stored in random order
    - Only needs 1 final integration step to update user account information at the end of a certain time period (i.e., one trading day).