Assignment 2

Deadline: 6:00pm Tuesday 11 October 2016

1. Problem: Double Auction Simulation System

Double auction is a market mechanism widely used in financial markets. Almost all stock exchange markets, such as NYSE, NASDQ, ASX and so on, use double auction mechanism for stock exchange. In a double auction market, *traders* (*buyers* or *sellers*) buy or sell a type of products, say shares, under the coordination of an *auctioneer* (broker). Assume that we consider the exchange of a single type of shares, say BHP. In each business session, buyers and sellers simultaneously submit their *buy bids* and *ask bids* to the auctioneer for buying or selling the share by specifying their bid prices and quantities (units of the share). The auctioneer then tries to match these bids and determines a clearing price for each matched pair of bids. The target of this assignment is to develop a simulation system that can mimic a double auction market (for single product only).

1.1 Bids

A bid has the form: (*traderName*, *bidId*, *bidType*, *price, quantity*), where

- *trderName* is a string, representing the name of a trader who made the bid
- *bidId* is an integer (in type of int), which is the identity of the bid. Different bid should have difference *bidId*.
- *bidType* can be either 'A' (for ask bid) or 'B' (for buy bid) in type of char.
- *price* represents the highest amount to buy or the lowest amount to sell (in type of double)[1].
- *quantity* represents how many units of the share the trader wants to buy or sell (in type of int).

For instance, *("Dongmo", 4, 'A', 95.60, 35*) represents an ask bid with id 4. The bid is from "*Dongmo*" who is going to sell 35 units of the share at a price no less than $95.60. *("Jane", 26, 'B', 105.10, 21*) represents a buy bid from "*Jane*" with id 26, which is to buy 21 units of the product at a price no more than $105.10.

[1] Note that a buyer will accept a deal if its clearing price is less than the bidding price, similar to a seller.

1.2 Matching

To reduce the complexity of the problem, you may start with a partial solution by assuming that the quantity of each bid is always 1 (for Tasks 1-5 and Task 6 option 2). In this case, matching will be much easier. Once you finished Tasks 1-5, you can either complete Task 6 option 1, which allows to sell or buy multiple units, or convert your code into Java (Task 6 option 2).

1.2.1 Matching bids with single unit

Assume that each bid is to buy or sell one unit of the share. The auctioneer collects all bids from the traders and tries to match them. Two bids can be matched if the following conditions are satisfied:

(1). *One is an ask bid and the other is a buy bid*;
(2). *The ask price is no higher than the buy price*.

For instance, among the following bids, Bid 312 can match either Bid 105 or Bid 680.

1. ("John", 105, 'B', 105.05, 1)
2. ("Bill", 236, 'B', 89.0, 1)
3. ("Alice", 312, 'A', 94.10, 1)
4. ("Tom", 680, 'B', 110.95, 1)
5. ("Sasa", 171, 'A', 160.20, 1)

No other matches can be made. The offer price of Bid 236 is too low. The ask price of Bid 171 is too high. Note that one ask bid can only match one buy bid.

1.2.2 Matching for bids with multiple units

If we allow a bid to buy or sell more than one units of the share, matching can be more complicated. We not only have to match their prices but also match their quantities. We say that two bids are *matchable* if they satisfy the above-mentioned two conditions plus the following condition:

(3). *The quantities of the bids are the same.*

To make matching possible, we assume that a bid can be replaced by several bids if their total quantity is the same as the original one and other information is the same. For instance, (*"Dongmo", 4, 'A', 95.60, 35*) can be replaced by two bids (*"Dongmo", 4, 'A', 95.60, 21*) and (*"Dongmo", 4, 'A', 95.60, 14*). Based on this assumption, in order to match (*"Dongmo", 4, 'A', 95.60, 35*) and *("Jane", 26, 'B', 105.10, 21*), we simply match (*"Dongmo", 4, 'A', 95.60, 21*) and *("Jane", 26, 'B', 105.10, 21*), and leave (*"Dongmo", 4, 'A', 95.60, 14*) as residual.

1.3 Clearing price

For each matched pair of bids, the *clearing price* of the match can be any value between the buying price and the asking price. For instance, the clearing price for the match *("John", 5, 'B', 105.05, 1*) and *("Alice", 1, 'A', 94.10, 1*) can be anything between $94.10 and $105.05. Normally we use the middle price, i.e. (105.05 + 94.10)/2 = 99.575, as the clearing price but not necessary.

2. Program tasks and testing

Task 1 (10%): Create a class, named `Bid`, with the data members: *`traderName`*, *`bidId`*, *`bidType`*, *`price`* and *`quantity`*. Define any member functions if necessary. Write a driver to test your class.

Task 2 (15%): Create a class, named `Trader`, to simulate a trader. The class should contain at lest two data members: *`traderName`* and *`TraderType`*. The class should also contain a member function: *Bid* *`generateBid()`*, which randomly generates a bid with price between MINPRICE and MAXPRICE, and quantity between MINQUANTITY and MAXQUANTITY[2]. This function can be implemented in this class. It can also be purely virtual in this class and implemented in its derived classes.

Task 3 (10%): Extend `Trader` class into two classes, called `Buyer` and `Seller`, using inheritance. If a trader is a buyer, the bid that is generated should be a buy bid. If a trader is a seller, the bid that is generated should be an ask bid. The bid id should be automatically generated so that the bids from different traders (objects of `Trader` class) have different bid ids. Write a driver to test your classes by generating ten ask bids and ten sell bids.

Task 4 (30%): Create a class, named `Auctioneer`, to implement an auctioneer. The class should take a set of buy bids and a set of sell bids as input, try to match them and determine the clearing price for each match (you may need another class to specify matches). You can design your own algorithm for matching and clearing price. If you do not want to attempt Task 6 (Option 1), you may set the quantity of each bid to be 1 or ignore the values of quantity.

Task 5 (20%): Create a class, named `Simulator`, to simulate a double auction market. The class should contain an object of `Auctioneer`, an array of `Buyer` objects, and an array of `Seller` objects. The class collects all bids from the trader objects and pass them to the auctioneer object for matching. All the results, including all generated

---

[2] MINPRICE, MAXPRICE, MINQUANTITY and MAXQUANTITY are constants. Their values can be hard-coded in the driver. You may preset the quantity to be 1 if you do not want to attempt Task 6 (option 1).

bids, matched bids and unmatched bids should be output into a text file. Write a driver to test your class[3].

Task 6 - Option 1 (10%): Remove the restriction that each bid can only be one unit of the share to allow bids with multiple units and redo Task 4. In this case, you have to redesign your match algorithm to allow a bid can be partially matched and leave the residual quantity for further matching.

Task 6 - Option 2 (10%): Change your code into Java. In other words, you need to submit two separate programs, one in C++ and one in Java. You may also complete the assignment purely in Java, which will be marked in the same way as you do it in C++. In this case you will not receive 10% extra marks.

Task 7 (5%): Refine your code to make it with better abstraction, encapsulation, readability and simplicity. You can only claim this mark if you have completed all the tasks (Tasks 1-5 plus either Task 6 option1 or Task 6 option 2).

Testing:

After you complete all the tasks specified above, you will end up with a simulation system of a double auction market. In this system, you simulate a number of traders (buyers and sellers) and a single auctioneer. Each trader can generate a bid. The bids from all traders are sent to the auctioneer. The auctioneer tries to match the bids as many as possible and determines the clearing price for each match. Your system should output the lists of all the generated bids, matched bids with clearing price and unmatched bids into a text file.

Please use the following setting when you demonstrate your program:

NUMSELLER = 10
NUMBUYER = 10
MINPRICE = 50
MAXPRICE = 150
MINQUANTITY = 1
MAXQUANTITY = 50

I will put the executable file of a sample solution in vUWS. Once it available, download and run it for your reference.

[3] If it is too hard for you to manage so many classes, you may implement the functionalities of task 5 in class auctioneer, which means the auctioneer collect all the bids, match them and output the results. In this case, you will have 10% marks less.

3. Submission

You need to submit your source code to vUWS for documentation purpose but you will gain your marks and feedback from your demonstration. The code should be purely written by yourself. No part of the code can be written by any other persons or copied from any other sources. Submit the following declaration with your code (in a text file or world file).

DECLARATION

I hereby certify that no part of this assignment has been copied from any other student's work or from any other source. No part of the code has been written/produced for me by another person or copied from any other source.

I hold a copy of this assignment that I can produce if the original is lost or damaged.

Both the declaration and source code should be submitted via vUWS before the deadline. Your programs (.h, .cpp or .java) can be put in separate files (executable file is not required). All these files should be zipped into one file **with your student id as the zipped file name**. Submission that does not follow the format is not acceptable.

Email submissions are not acceptable.

5. Demonstration

You are required to demonstrate your program during **your scheduled** practical session in Week 13 (Wednesday 12 or Friday 14 **October 2016**). **You will receive no marks if you fail the demonstration.** Note that it is your responsibility to get the appropriate compilers or IDEs to run your program. You are allowed to run your program from your laptop. **The feedback to your work will be delivered orally during the demonstration.** No further feedback or comments are given afterward.

The program you demonstrate should be the same as the one you submit except that the comments in your program should be removed before the demonstration.

Please read this specification at least three times to get a full understanding of the problem.

6. Extra tasks for advanced computer science students or the students who like to take more challenges, you can complete the following extra tasks (these extra tasks can be completed after the deadline of assignment 2):

Task 8: Extend your class `Trader` to a class `SmartTrader` using inheritance to implement a *profit-aware trader*. Your trader should be able to determine the bidding price for each bid it creates by using the random generated price as reserve price. For instance, assume you are a seller. When you create a bid, the random generator gives you a price, say 78. This will be your cost per share. You do not have to sell your items exactly at this price. To make more profit, you could use a higher price as your ask price.

However, if your reserve price has been already very high, say 124, you might have to use this price as your ask price (any price lower will have a risk of negative profit).