Course Programming Project

DePaul Stock Exchange (DSX)

Phase 4

I have made a few updates to this document where students have pointed out typos ro areas that needed clarification. To help you identify these changes they are highlighted in yellow in this document, and I've listed their locations below:

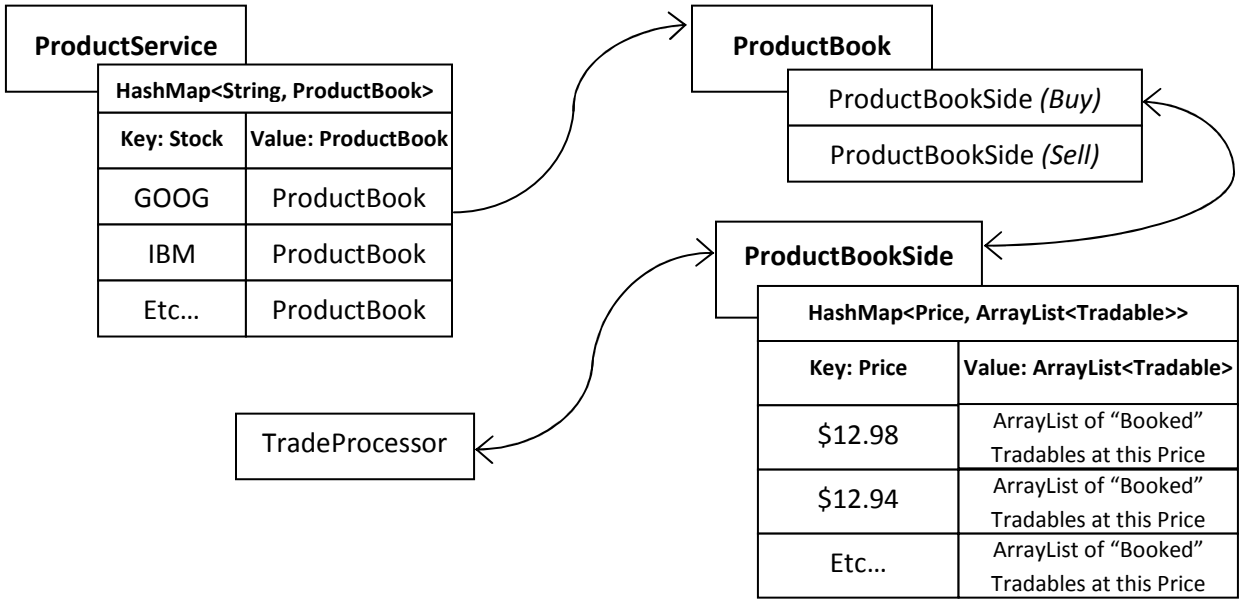
- Page 8, in section 2.9, "mergeFills"
- Page 13, in section 3.3, "openMarket" & 3.4 "closeMarket"
- Page 14, in section 3.7, "addToBook(Quote q)" & 3.9, "updateCurrentMarket"
- Page 15, in section 3.10, "determineLastSalePrice" & 3.11, "determineLastSaleQuantity"
- Page 15, in section 3.12, "addToBook(BookSide side, Tradable trd)"
- Page 16/17, in section 4.4, "getBookDepth(String product)"
- Page 17, in section 4.1, "setMarketState(MarketState ms)"
- Page 18, in section 4.5, "submitOrder(Order o)"
- Page 19, section 5.3, "addFillMessage(FillMessage fm)"
- Page 23, in Step #3 of the "doTrade" walkthrough.

Last updated 10/28 at 8:56pm

ProductService, ProductBook, ProductBookSide & TradeProcessor

Phase 4 of the Course Programming Project involves the creation of the classes that will represent the ProductService and Product Books, and the Trade Processor. The largest part of the trading system's functionality resides in these classes. As these classes involve new functionality, put them in a new package (i.e., "book").

- The **ProductService** is the Façade to the Product Books ("booked" Tradable objects on the Buy and Sell side).
- The **ProductBook** is the list of not-yet-traded (i.e., "booked") Orders and Quote Sides (Tradables) for the Buy side and for the Sell side. To represent this, the ProductBook will own two ProductBookSide objects – one for the BUY side and one for the SELL side.
- The **ProductBookSide** owns a list of the "booked" Tradable objects, organized by Price and by time of arrival. An individual ProductBookSide represents the Buy side or the Sell side. Each ProductBookSide also owns a TradeProcessor object.
- The **TradeProcessor** is an interface that defines the functionality needed to "execute" the actual trades between Tradable objects in this book side.
- The **TradeProcessorPriceTimeImpl** class implements the TradeProcessor interface. This implementor will execute trades in Price-Time priority, which means according to price and time of arrival. *Other trade execution algorithms exist (like "ProRata" – trades by equal proportions) but we will not implement these as a part of this project, though your design should allow for the addition of new potential TradeProcessor* implementers.



Phase 3 Class Detail

1. TradeProcessor *interface*

The **TradeProcessor** is an interface that defines the functionality needed to "execute" the actual trades between Tradable objects in this book side. There is only one method on this interface:

1.1 public HashMap<String, FillMessage> doTrade(Tradable trd)

This TradeProcessor method will be called when it has been determined that a Tradable (i.e., a Buy Order, a Sell QuoteSide, etc.) can trade against the content of the book. The return value from this function will be a HashMap<String, FillMessage> containing String trade identifiers (the key) and a Fill Message object (the value).

NOTE: There will initially be only ONE implementer of the TradeProcessor interface called TradeProcessorPriceTimeImpl (seen later in this handout). In future, there could be other implementers as well.

2. ProductBookSide *class*

A ProductBookSide object maintains the content of one side (Buy or Sell) of a Stock (product) "book". Recall that a stock's "book" holds the buy and sell orders and quote sides for a stock that are not yet tradable. The buy-side of the book contains all buy-side orders and quote sides that are not yet tradable in descending order. The sell-side of the book contains all buy-side orders and quote sides that are not yet tradable in ascending order. The "book" is often visually represented as follows, with a product book side circled:

MSFT (Microsoft) Book			
BUY		SELL	
1000	$29.05	$29.07	1000
120	$29.01	$29.08	120
210	$28.98	$29.10	210
80	$28.94	$29.12	80
...	...	...	...

The ProductBookSide class needs the following **data elements** to represent one side of a product's book:

2.1 The "side" that this ProductBookSide represents – BUY or SELL. *You should represent this as you have represented BUY & SELL in previous Phases (enumeration, constants, etc.).*

2.2 A collection of book entries for this side, represented in a way that organizes the book entries by Price, and then by time of arrival. This can most easily be represented using a HashMap that uses a Price object as a key and has an ArrayList of Tradables as the value. I refer to this data member as "bookEntries" in this handout. Example:

private HashMap<Price, ArrayList<Tradable>> **bookEntries** = new HashMap< Price, ArrayList<Tradable>>();

HashMap< Price, ArrayList<Tradable>>	
Key: Price	Value: ArrayList<Tradable>
$12.98	[Tradable, Tradable, Tradable, Tradable, Tradable]
$12.95	[Tradable]
$12.90	[Tradable, Tradable, Tradable]

2.3 A reference to a "TradeProcessor" (interface) object. This TradeProcessor will be used to execute trades against this book side. The TradeProcessor data member should be set in the ProductBookSide constructor. *Currently the only implementer of the TradeProcessor is the TradeProcessorPriceTimeImpl class but in the future there could be others so the TradeProcessor data member should be set using a factory that makes TradeProcessor objects.*

2.4 A reference back to the ProductBook object that this ProductBookSide belongs to. I refer to this as the "parent" product book in this handout. This should be set in the constructor (and should not be null).

Required public ProductBookSide functionality:

- Constructor – This should accept a reference to the ProductBook object that this ProductBookSide belongs to, and a side indicator (BUY or SELL) that specifies the "side" that this ProductBookSide represents. The TradeProcessor data member should be setup in the constructor, setting it to a new TradeProcessorPriceTimeImpl object (that class is described later in this handout).

- ProductBookSide Query Methods:
  These methods perform various queries against the ProductBookSide and return the results of those queries so that the Trading System can make decisions on how to behave.
  Note that most of the methods here require the "synchronized" keyword. An overview of what this synchronized keyword does and why it is needed can be found in Appendix A of this document "Blocking Multithreaded Execution Using 'synchronized'".

  2.1 public *synchronized* ArrayList<TradableDTO> getOrdersWithRemainingQty(String userName)
  This method will generate and return an ArrayList of TradableDTO's containing information on all the orders in this ProductBookSide that have remaining quantity for the specified user.
  - Since this method returns an ArrayList of TradableDTO objects, the first thing you should do is create a new (empty) ArrayList of TradableDTO objects.
  - Then, for every *Order* in the book (in "the "bookEntries" HashMap) for the specified userName that has remaining volume > 0, create a new TradableDTO using the data from that Order, and add that TradableDTO to your new ArrayList of TradableDTO's.
  - Once complete, return the ArrayList of TradableDTO objects.

  2.2 *synchronized* ArrayList<Tradable> getEntriesAtTopOfBook()
  This method should return an ArrayList of the Tradables that are at the best price in the "bookEntries" HashMap. To do this:
  - *Note this method should be "package-visible" so no "public" or "private" keyword should be used in the method declaration.*
  - If the "bookEntries" HashMap is empty, then return null.
  - Otherwise, create a sorted ArrayList of Prices in the book using the keys to the "bookEntries" HashMap as the source of Prices. This is done as follows:

```
ArrayList<Price> sorted = new ArrayList<Price>(bookEntries.keySet()); // Get prices
Collections.sort(sorted); // Sort them
if (side == BookSide.BUY) {
```

```
        Collections.reverse(sorted); // Reverse them
    }
```

Now "sorted" is a sorted ArrayList of Prices. Note - you need to import java.util.Collections when you do this.

- Get the first Price in that ArrayList, and use that as a key to the "bookEntries" HashMap and return the ArrayList of Tradables associated with that Price (i.e., return bookEntries.get(sorted.get(0))).

2.3 public *synchronized* String[] getBookDepth()

This method should return an array of Strings, where each index holds a "Price x Volume" String. This is done as follows:

- If the this product book side is empty (if the "bookEntries" HashMap is empty), make a new String array of size one, add the String "<Empty>" to it, and return that array. (i.e., *return new String[]{"<Empty>"};*)
- Otherwise, make a new String array – the size should be same size as the "bookEntries" HashMap's size. We will fill this array in the next few steps.
- Then, create a sorted ArrayList of the prices in this *book (like you did in the previous "getEntriesAtTopOfBook()" method).*
- For each price in that sorted ArrayList
  - Use the price as the key into the "bookEntries" HashMap and get the associated ArrayList of Tradables
  - Sum up the remaining volume of all the Tradables in that ArrayList – that is the volume at this price.
  - Then once you have summed those up, make a String: Price + " x " + volume sum (i.e., "$641.20 x 700", "$22.90 x 120", etc). Add that String to the String array.
- Once complete, return the String array.

2.4 *synchronized* ArrayList<Tradable> getEntriesAtPrice(Price price)

Note this method should be "package-visible" so no "public" or "private" keyword should be used in the method declaration.

This method should return all the Tradables in this book side at the specified price.

- If the provided price is not a key in the "bookEntries" HashMap then return null.
- Otherwise, return the ArrayList of Tradables associated with that Price. (i.e., return bookEntries.get(price)).

2.5 public *synchronized* boolean hasMarketPrice()

This method should return true if the product book (the "bookEntries" HashMap) contains a Market Price (if a MarketPrice is a key in the "bookEntries" HashMap you return true, otherwise false).

2.6 public *synchronized* boolean hasOnlyMarketPrice()

This method should return true if the ONLY Price in this product's book is a Market Price (if there is *only one key* in the "bookEntries" HashMap and it is a Market Price, you return true, otherwise false).

2.7 public *synchronized* Price topOfBookPrice()

This method should return the best Price in the book side. If the "bookEntries" HashMap is empty, then return null. Otherwise, create a sorted ArrayList of Prices *like you did earlier in the "getEntriesAtTopOfBook" method.* Then return the first Price in that list.

2.8 public *synchronized* int topOfBookVolume()

This method should return the volume associated with the best Price in the book side. If the "bookEntries" HashMap is empty, then return zero. Otherwise, create a sorted ArrayList of Prices *like you did earlier in the "getEntriesAtTopOfBook" method.* Get the first Price in that ArrayList, and use that as a key to the

"bookEntries" HashMap and get the ArrayList of Tradables associated with that Price. Then sum up all the remaining volume values for the Tradables in that ArrayLisy and return that sum.

2.9 public *synchronized* boolean isEmpty()

Returns true if the product book (the "bookEntries" HashMap) is empty, false otherwise.

- ProductBookSide Manipulation Methods:

These methods perform various operations that will add and remove Tradables in the ProductBookSide. Additionally, some of these methods will work with the TradeProcessor object in performing the execution of a trade.
Note that most of the methods here require the "synchronized" keyword. An overview of what this synchronized keyword does and why it is needed can be found in Appendix A of this document "Blocking Multithreaded Execution Using 'synchronized'".

2.1 public *synchronized* void cancelAll()

This method should cancel every Order or QuoteSide at every price in the book (in "the "bookEntries" HashMap). This should make use of the submitOrderCancel(String orderId) and submitQuoteCancel(String userName) methods.

2.2 public *synchronized* TradableDTO removeQuote(String user)

This method should search the book (the "bookEntries" HashMap) for a Quote from the specified user (a user can only have one Quote per product so there will either be none, or one). Once found, remove the Quote from the book, and create a TradableDTO using data from that QuoteSide, and return the DTO from the method. If no quote is found, return null. *Note, if the Quote was the last Tradable in the ArrayList of Tradables at that price, remove the price entry from the "bookEntries" HashMap (i.e., bookEntries.remove(price))*

2.3 public *synchronized* void submitOrderCancel(String orderId)

This method should cancel the Order (if possible) that has the specified identifier. This method should search the book at all prices (the "bookEntries" HashMap) for the Order with the specified identifier.

- For each Price key in the "bookEntries" HashMap, get the ArrayList of Tradable objects (the "value") and search the ArrayList for an Order with the same Id as the ordered passed in.
- Once found, do the following:
  - Remove it from the ArrayList of Tradable objects at that price
  - Publish a Cancel message
  - Call the "addOldEntry(Tradable)" method (in this class), passing a reference to the order you found with the requested identifier.
  - *If the Order was the last Tradable in the ArrayList of Tradables at that price, remove the price entry from the "bookEntries" HashMap (i.e., bookEntries.remove(price)).*
  - Then return.
- If NO order is found with the specified Id, then it was already filled or cancelled.
  - Call the "parent" ProductBook's "checkTooLateToCancel(orderId)" method. Then return.

2.4 public *synchronized* void submitQuoteCancel(String userName)

This method should cancel the QuoteSide (if possible) that has the specified userName. To do this, call the "removeQuote(username)" method (in this class). That will return a TradableDTO object. If that TradableDTO object is *not* null, create and publish a cancel message (using the data from the DTO). If the "removeQuote(username)" method returns null, then there's nothing to do so just return.

To create a CancelMessage, you build it using the data from the TradableDTO object you received back from the "removeQuote(username)" call (i.e., dto.user, dto.product, dto.price, dto.remainingVolume, etc.). You should use the String *"Quote " + side + "-Side Cancelled"* as the "details" String in the cancel message, where "side" is the BUY/SELL side that the QuoteSide belongs to – from the DTO. To publish a cancel message, you ask the message publisher to publish a CancelMessage object:

MessagePublisher.getInstance().publishCancel(aCancelMessageObject);

2.5 public void addOldEntry(Tradable t)

This method should add the Tradable passed in to the "parent" product book's "old entries" list. To do this:

- There is a data member in this class that refers to the "parent ProductBook" for this ProductBookSide. Call that ProductBook's "addOldEntry" method, passing it the Tradable passed in.

2.6 public *synchronized* void addToBook(Tradable trd)

This method should add the Tradable passed in to the book (the "bookEntries" HashMap). To do so:

- First check if the HashMap has a "key" of the Tradable's Price. If not, create a new ArrayList of Tradables and add it to the "bookEntries" HashMap using the Tradable's Price as the key.
- Now, get the ArrayList or Tradables from the "bookEntries" HashMap for that Price key and add the Tradable to that ArrayList.

2.7 public HashMap<String, FillMessage> tryTrade(Tradable trd)

This method will attempt a trade the provided Tradable against entries in this ProductBookSide. To do this, you need the following:

- Create a temporary "HashMap<String, FillMessage>" variable (to hold potential FillMessages) but do not yet initialize it to anything – I called mine "allFills".
- If the "side" of this ProductBookSide is BUY, then set the "allFills" HashMap you just created to the results of a call to the "trySellAgainstBuySideTrade(Tradable)" method (that method is in this class). Use the Tradable "trd" passed in as the parameter.
- Else if the "side" of this ProductBookSide is SELL, then set the "allFills" HashMap you just created to the results of a call to the "tryBuyAgainstSellSideTrade(Tradable)" method (that method is in this class). Use the Tradable "trd" passed in as the parameter.
- Then, after the IF/ELSE - for each FillMessage in the "allFills" HashMap, call the MessagePublisher to publish the fills. To publish a Fill Message, you do something like this:

  MessagePublisher.getInstance().publishFill(aFillMessage);
- Finally, after the for each loop, return the "allFills" HashMap.

2.8 public *synchronized* HashMap<String, FillMessage> trySellAgainstBuySideTrade(Tradable trd)

This method will try to fill the SELL side Tradable passed in against the content of the book. To do this,

- First, create 2 new HashMaps that will be used to hold then return FillMessages. You can declare these as:

  HashMap<String, FillMessage> *allFills* = new HashMap<String, FillMessage>();

  HashMap<String, FillMessage> *fillMsgs* = new HashMap<String, FillMessage>();
- Then, you need to iteratively try to trade the Tradable until there is nothing left in this book side or until no more trades can occur at the requested price. To do this you need a "while" loop:
  - **While (**the tradable "trd" passed in has remaining volume, AND this book side is not empty (use the "isEmpty" method), AND the price of the tradable passed is **less than or equal to** the "topOfBookPrice()" of this book (that is a method in this class**)**
  - OR

- (the tradable "trd" passed in has remaining volume, AND this book side is not empty (use the "isEmpty" method), AND the price of the tradable passed is a Market Price)
  - Call the TradeProcessor's "doTrade(Tradable)" method passing it the Tradable "trd" as its parameter. That method will return a HashMap<String, FillMessage>, so make a temporary HashMap<String, FillMessage> variable and set it equal to the "doTrade" results. Do not use the HashMaps "allFills" or "fillMsgs" from the first step to do this, make a new variable: i.e., *HashMap<String, FillMessage> someMsgs.*
  - After that, set the "fillMsgs" HashMap (from the first step) to the results of a call to "mergeFills" -- fillMsgs = mergeFills(fillMsgs, someMsgs). This will add the content of "someMsgs" to "fillMsgs" (see the "mergeFills" method for the details on what that process involves).
- End the While loop

- After the While loop, add all the FillMessages from the "fillMsgs" HashMap to the "allFills" HashMap, then return the "allFills" HashMap. This can be easily done as follows: allFills.putAll(fillMsgs);

2.9 private HashMap<String, FillMessage> mergeFills(HashMap<String, FillMessage> existing, HashMap<String, FillMessage> newOnes)

This method is designed to merge multiple fill messages together into one consistent list. When one user's Order of QuoteSide trades against multiple Tradable objects at the same price – only ONE fill should be sent (for the total traded volume). For example, if a user's BUY Order for 200@$12.00 trades with 2 orders on the SELL side (both those orders are for 100@$12.00), then the user should only get ONE fill message for 200, not TWO fills each for 100. Without merging multiple fills together, the user would get many fills at the same price for their same tradable. Do merge these fills together, do the following:

- If the "existing" HashMap parameter passed in is empty, then make a new HashMap that contains the content of the "newOnes" HashMap parameter and return it. That means the target HashMap has no fill messages in it do there's nothing to merge. To do this:, you just need to do this:

  return new HashMap<String, FillMessage>(newOnes);
- Otherwise – there are fills to merge with. Make a new HashMap<String, FillMessage> (i.e., called "results") that contains the content of the "existing" HashMap:

  HashMap<String, FillMessage> results = new HashMap<>(existing);
- Next, you should add the following "for each" loop. Examine this code to be sure you understand what it is doing. You may need to modify method names in this to match your method names:

```
for (String key : newOnes.keySet()) {   // For each Trade Id key in the "newOnes" HashMap
    if (!existing.containsKey(key)) {  // If the "existing" HashMap does not have that key…
        results.put(key, newOnes.get(key));  // …then simply add this entry to the "results" HashMap
    } else {   // Otherwise, the "existing" HashMap does have that key – we need to update the data
        FillMessage fm = results.get(key);  // Get the FillMessage from the "results" HashMap
        // NOTE – for the below, you will need to make these 2 FillMessage methods "public"!
        fm.setFillVolume(newOnes.get(key).getFillVolume());  // Update the fill volume
        fm.setDetails(newOnes.get(key).getDetails()); // Update the fill details
    }
}
```

- Then finally, return the "results" HashMap.

2.10 public *synchronized* HashMap<String, FillMessage> tryBuyAgainstSellSideTrade(Tradable trd)

This method will try to fill the BUY side Tradable passed in against the content of the book. To do this,

- First, create 2 new HashMaps that will be used to hold then return FillMessages. You can declare these as:
  HashMap<String, FillMessage> *allFills* = new HashMap<String, FillMessage>();
  HashMap<String, FillMessage> *fillMsgs* = new HashMap<String, FillMessage>();
- Then, you need to iteratively try to trade the Tradable until there is nothing left in this book side or until no more trades can occur at the requested price. To do this you need a "while" loop:
  - **While (**the tradable "trd" passed in has remaining volume, AND this book side is not empty (use the "isEmpty" method), AND the price of the tradable passed is **greater than or equal to** the "topOfBookPrice()" of this book (that is a method in this class**)**
  - OR
  - (the tradable "trd" passed in has remaining volume, AND this book side is not empty (use the "isEmpty" method), AND the price of the tradable passed is a Market Price)
    - Call the TradeProcessor's "doTrade(Tradable)" method passing it the Tradable "trd" as its parameter. That method will return a HashMap<String, FillMessage>, so make a temporary HashMap<String, FillMessage> variable and set it equal to the "doTrade" results. Do not use the HashMaps "allFills" or "fillMsgs" from the first step to do this, make a new variable: i.e., *HashMap<String, FillMessage> someMsgs.*
    - After that, set the "fillMsgs" HashMap (from the first step) to the results of a call to "mergeFills" -- fillMsgs = mergeFills(fillMsgs, someMsgs). This will add the content of "someMsgs" to "fillMsgs" (see the "mergeFills" method for the details on what that process involves).
  - End the While loop
- After the While loop, add all the FillMessages from the "fillMsgs" HashMap to the "allFills" HashMap, then return the "allFills" HashMap. This can be easily done as follows: allFills.putAll(fillMsgs);

2.11 public *synchronized* void clearIfEmpty(Price p)

This method will remove an key/value pair from the book (the "bookEntries" HashMap) if the ArrayList associated with the Price passed in is empty. To do so, get the ArrayList of Tradables from the "bookEntries" HashMap using the Price passed in as the key. If the ArrayList of Tradables is empty, then remove the Price key from the Hashmap.

2.12 public *synchronized* void removeTradable(Tradable t)

This method is design to remove the Tradable passed in from the book (when it has been traded or cancelled). To do this:

- Get the ArrayList of Tradables associated with the Price from the Tradable passed in:
  ArrayList<Tradable> entries = bookEntries.get(t.getPrice());
- If the ArrayList of Tradables you get back is null, then just return – there is nothing to remove at the Tradable's price.
- Otherwise, remove the Tradable from the "entries" ArrayList you just retrieved by calling the "remove" method of the ArrayList, passing it the Tradable parameter. Save the boolean result of the "remove" operation.
- If that boolean result is false, then the Tradable wasn't there so just return.
- Then, if "entries" ArrayList is left empty after this remove operation, then call "clearIfEmpty" passing the price of the Tradable as the parameter.

Done with ProductBookSide

3. ProductBook *class*

A ProductBook object maintains the Buy and Sell sides of a stock's "book". Many of the functions owned by the ProductBook class are designed to simply pass along that same call to either the Buy or Sell side of the book. Recall that a stock's "book" holds the buy and sell orders and quote sides for a stock that are not yet tradable. The buy-side of the book contains all buy-side orders and quote sides that are not yet tradable in descending order. The sell-side of the book contains all buy-side orders and quote sides that are not yet tradable in ascending order. The "book" is often visually represented as follows:

MSFT (Microsoft) Book			
BUY		SELL	
1000	$29.05	$29.07	1000
120	$29.01	$29.08	120
210	$28.98	$29.10	210
80	$28.94	$29.12	80
…	…	…	…

The ProductBook will need the following **data** to properly represent a product's book:

3.1 The String stock symbol that this book represents (i.e., MSFT, IBM, AAPL, etc).

3.2 A ProductBookSide that maintains the Buy side of this book.

3.3 A ProductBookSide that maintains the Sell side of this book.

3.4 A String that will hold the toString results of the latest Market Data values (the prices and the volumes at the top of the buy and sell sides). This is explained later in the method descriptions

3.5 A list of the current quotes in this book for each user. This is best done using a HashSet:

```
private HashSet<String> userQuotes = new HashSet<>();
```

3.6 A list of "old" Tradables (those that have been completely traded or cancelled) organized by price and by user. This should be represented like you represented the book entries for one side of the book in the ProductBookSide class:

```
private HashMap<Price, ArrayList<Tradable>> oldEntries = new HashMap< Price, ArrayList<Tradable>>();
```

Required public ProductBook functionality:

- Constructor – This only needs to accept the String stock symbol this book represents. That should be used to set the stock symbol data member. The Buy and Sell side books should be created here in the ProductBook constructor as well. Nothing else needs to be initialized here.

- Query Methods:
  These methods perform various queries against the ProductBook and return the results of those queries so that the Trading System can make decisions on how to behave.

Note that most of the methods here require the "synchronized" keyword. An overview of what this synchronized keyword does and why it is needed can be found in Appendix A of this document "Blocking Multithreaded Execution Using 'synchronized'"

3.1 public *synchronized* ArrayList<TradableDTO> getOrdersWithRemainingQty(String userName)

This method should return an ArrayList containing any orders for the specified user that have remaining quantity. To do this,

- First create a new ArrayList of TradableDTO objects. We will add DTOs to this list that represent the remaining orders.
- Then call the BUY side ProductSideBook's "getOrdersWithRemainingQty(String)" method passing the userName parameter – this will return an ArrayList of TradableDTO's. All them all to the ArrayList of TradableDTO objects created in the first step.
- Then call the SELL side ProductSideBook's "getOrdersWithRemainingQty(String)" method passing the userName parameter – this will return an ArrayList of TradableDTO's. All them all to the ArrayList of TradableDTO objects created in the first step.
- Return that ArrayList of TradableDTO objects created in the first step.

3.2 public *synchronized* void checkTooLateToCancel(String orderId)

This method id designed to determine if it is too late to cancel an order (meaning it has already been traded out or cancelled). To do this, check the "oldEntries" HashMap to see if any of the Orders there have an Id that matches the order Id passed in. If so, then that order has already been fully traded or cancelled. In that case, create and publish a Cancel Message, using the data from the order you found and passing the String "Too Late to Cancel" for the cancel detail. If it is not in the "oldEntries" HashMap, throw an exception (i.e., "OrderNotFoundException") with a message indicating that the requested order could not be found.

3.3 public *synchronized* String[ ][ ] getBookDepth()

This method is should return a 2-dimensional array of Strings that contain the prices and volumes at all prices present in the buy and sell sides of the book. To do this, create a new 2 dimensional array as follows: String[][] bd = new String[2][]; Here, index "0" will hold the Buy side array of Strings, and index "1" will hold the Buy side. Set "bd[0]" to the results of a call to the buy-side ProductBookSide's "getBookDepth()" method, and set Set "bd[1]" to the results of a call to the sell-side ProductBookSide's "getBookDepth()" method. Then return the "bd" array.

3.4 public *synchronized* MarketDataDTO getMarketData()

This method should create a MarketDataDTO containing the best buy side price and volume, and the best sell side price an volume. To do this:

- Get the best buy side price by calling the buy-side ProductBookSide's "topOfBookPrice()" method.
- Then get the best sell side price by calling the sell-side ProductBookSide's "topOfBookPrice()" method.
- If either of these is null, reset the variable you are using to hold that price to a limit price of "0.00" – we don't want to return "null" as a price.
- Then, get the best buy side volume by calling the buy-side ProductBookSide's "topOfBookVolume ()" method.
- Additionally, get the best sell side volume by calling the sell-side ProductBookSide's "topOfBookVolume ()" method.

- Now create (and then return) a new MarketDataDTO using the stock symbol (a data member of this ProductBook), and the buy and sell prices and volumes as the parameters to the MarketDataDTO constructor.

- Book Side Manipulation Methods:

3.1 public *synchronized* ArrayList<TradableDTO> getOrdersWithRemainingQty(String userName)

This method is should return an ArrayList<TradableDTO> objects that contain the details on all orders from the specified user that are still active with remaining volume from both sides of the book (Buy/Sell). To do this:

- First create a new ArrayList of TradableDTO objects (we will fill this list with TradableDTOs).
- Call the buy side ProductBookSide "getOrdersWithRemainingQty(userName)" method to get an ArrayList of TradableDTO objects from the Buy side. Add the content of this ArrayList to the new Arraylist you created in the first step (i.e., newList.addAll(otherList);).
- Then call the sell side ProductBookSide "getOrdersWithRemainingQty(userName)" method to get an ArrayList of TradableDTO objects from the Sell side. Add the content of this ArrayList to the new Arraylist you created in the first step.
- Now return that Arraylist you created in the first step.

3.2 public *synchronized* void addOldEntry(Tradable t)

This method should add the Tradable passed in to the "oldEntries" HashMap. To do this:

- If the "oldEntries" HashMap does not have the Price of the Tradable passed in as a key, add a new ArrayList<Tradable> to that HashMap using the tradable's price as the key.
- Set the cancelled volume of the tradable to be the same as its remaining volume.
- Now set the remaining volume of the tradable to zero.
- Finally, get the ArrayList of Tradables from the "oldEntries" HashMap using the Price of the Tradable passed in as a key, and add the Tradable parameter to that ArrayList.

3.3 public *synchronized* void openMarket()

This method will "Open" the book for trading. Any resting Order and QuoteSides that are immediately tradable upon opening should be traded. To do this process, do the following *(there are many steps here, but none are particularly complex so just take them one at a time and be careful not to miss any):*

- Get and hold (using a new temporary Price variable "buyPrice") the best Buy price by calling the "topOfBookPrice()" of the Buy-side ProductBookSide.
- Get and hold (using a new temporary Price variable "sellPrice") the best Sell price by calling the "topOfBookPrice()" of the Sell-side ProductBookSide.
- If either of these Prices is null then one of these sides has no Tradables so there will be no opening trade, so just return.
- Now we need to iteratively attempt to trade the reating tradables if possible. To do this, you need a while loop:
- WHILE (the buyPrice is greater than or equal to the sell price OR the buyPrice is a MKT price OR the sellPrice is MKT):
  - Create a new ArrayList of Tradables variable to hold the entries at the top of the buy-side book (I called this "topOfBuySide"), and set that to the results of a call to the buy ProductSideBook's "getEntriesAtPrice" method, passing it the "buyPrice" you got earlier.
  - Create a new HashMap<String, FillMessage> "allFills" variable to hold FillMessages. Leave this as null for now.
  - Then create another new ArrayList of Tradables variable "toRemove" to hold the entries that fully trade as a result of this processing. These are the Tradables that will be removed from the book when the opening trading is complete.

- For each Tradable "t" in the "topOfBuySide" ArrayList, do the following:
  - Set the "allFills" variable created earlier to the results of calling the "tryTrade(Tradable)" method of the SELL side ProductBookSide, passing it the "t" tradable.
  - If the Tradable "t"s remaining volume is now zero, add "t" to the "toRemove" ArrayList.
- End For each
- Now, for each Tradable "t" in the "toRemove" ArrayList, do the following:
  - Call the Buy ProductSideBook's "removeTradable" method passing it the "t" tradable as the parameter. This will remove the Traded items from the book.
- End For each
- Then – call "updateCurrentMarket()" (that methods is in this class).
- Now, set a new temporary Price variable "lastSalePrice" to the results of a call to the "determineLastSalePrice" method (that method is in this class) passing the "allFills" HashMap as its parameter.
- Then, set a new temporary int variable "lastSaleVolume" to the results of a call to the "determineLastSaleQuantity" method (that method is in this class) passing the "allFills" HashMap as its parameter.
- Now publish a Last Sale message passing this book's product symbol, the lastSalePrice, and the lastSaleVolume.
- Reset the temporary Price variable "buyPrice" by calling the "topOfBookPrice()" of the Buy-side ProductBookSide.
- Reset the temporary Price variable "sellPrice" by calling the "topOfBookPrice()" of the Sell-side ProductBookSide.
- If either of these Prices is null then one of these sides has no Tradables so there will be no more trades, so just "break" from the WHILE loop.

- End of WHILE loop

3.4 public *synchronized* void closeMarket()

This method will "Close" the book for trading. To do this, all you need to do is call the "cancelAll()" method of the BUY and SELL ProductBookSides. After that – call "updateCurrentMarket()".

3.5 public *synchronized* void cancelOrder(BookSide side, String orderId)

This method will cancel the Order specified by the provided orderId on the specified side. To do this, call the "submitOrderCancel(orderId)" method of the ProductBookSide specified by the "side" parameter. Then call the "updateCurrentMarket()" method.

3.6 public *synchronized* void cancelQuote(String userName)

This method will cancel the specified user's Quote on the both the BUY and SELL sides. To do this, first call the "submitQuoteCancel(userName)" method of the Buy ProductBookSide. Then call the "submitQuoteCancel(userName)" method of the Sell ProductBookSide. Then call the "updateCurrentMarket()" method.

3.7 public *synchronized* void addToBook(Quote q)

This method should add the provided Quote's sides to the Buy and Sell ProductSideBooks. To do this:

- First verify that the Quote's quote sides are valid:
  - If the SELL QuoteSide's price is less than or equal to the BUY QuoteSide's price, throw an exception (i.e., "DataValidationException") with a message indicating that the sell price is less than or equal to the buy price. That is an illegal Quote - Sell price must be greater than Buy price.

- If either the BUY or SELL QuoteSide's Price is less than or equal to zero, throw an exception (i.e., "DataValidationException") with a message indicating that the price of the BUY or SELL side is less than or equal to zero. That is an illegal Quote price.
- If either the BUY or SELL QuoteSide's original volume is less than or equal to zero, throw an exception (i.e., "DataValidationException") with a message indicating that the volume of the BUY or SELL side is less than or equal to zero. That is an illegal Quote volume.

- If the Quote passes these validations, then check if the "userQuotes" HashSet (a data member of this class) contains the userName found in the Quote parameter. This is done as follows:

  if (userQuotes.contains(q.getUserName()))

  If it does contain the userName, call "removeQuote" (on both the BUY and SELL ProductBookSides) passing the Quote's userName as the parameter. Then call "updateCurrentMarket()".
- Now, call "addToBook(Side, Tradable)" method using BUY as the side and the BUY QuoteSide object from the Quote parameter passed in.
- Then, call "addToBook(Side, Tradable)" method using SELL as the side and the SELL QuoteSide object from the Quote parameter passed in.
- Now add the quote parameter's userName to the "userQuotes" HashSet:

  userQuotes.add(q.getUserName())
- Finally, call "updateCurrentMarket()".

3.8 public *synchronized* void addToBook(Order o)

This method should add the provided Order to the appropriate ProductSideBook. To do this:

- Call "addToBook(Side, Tradable)" method using Order parameter's Side as the side and Order parameter as the Tradable.
- Then, call "updateCurrentMarket()".

3.9 public *synchronized* void updateCurrentMarket()

This method needs to determine if the "market" for this stock product has been updated by some market action. To do this:

- Create a String variable and set it to the following (no spaces in the String):
  The Buy-side ProductBookSide's "topOfBookPrice()" + the Buy-side ProductBookSide's topOfBookVolume() + the Sell-side ProductBookSide's "topOfBookPrice()" + the Sell-side ProductBookSide's topOfBookVolume().
  Example String: "$641.10120$641.15150", "$51.13444$5.15150", etc.
- If the "lastCurrentMarket" String (a data member of this class) does *not* equals the String you just created, then the market has changed, so do the following:
  - Create a MarketDataDTO using this ProductBook's stock symbol, the Buy-side ProductBookSide's "topOfBookPrice()", the Buy-side ProductBookSide's topOfBookVolume(), the Sell-side ProductBookSide's "topOfBookPrice()"and the Sell-side ProductBookSide's topOfBookVolume().
  - Publish a Current Market message (using the CurrentMarketPublisher) using the MarketDataDTO you just created.
  - Set the "lastCurrentMarket" String to the String value you created in the first step.

3.10 <u>private</u> synchronized Price determineLastSalePrice(HashMap<String, FillMessage> <u>fills</u>)

This method will take a HashMap of FillMessages passed in and determine from the information it contains what the Last Sale price is. This is done as follows:

- Create a new temporary ArrayList containing the FillMessages (the "values" from the HashMap):

  ArrayList<FillMessage> msgs = new ArrayList<>(fills.values());

- Sort those FillMessage by calling "Collections.sort(msgs);".
- Return the Price of that first Fill Message in that sorted Arraylist.

3.11 private synchronized int determineLastSaleQuantity(HashMap<String, FillMessage> fills)

This method will take a HashMap of FillMessages passed in and determine from the information it contains what the Last Sale quantity (volume) is. This is done as follows:

- Create a new temporary ArrayList containing the FillMessages (the "values" from the HashMap):

  ArrayList<FillMessage> msgs = new ArrayList<>(fills.values());
- Sort those FillMessage by calling "Collections.sort(msgs);".
- Return the fill volume of the first Fill Message in that sorted Arraylist.

3.12 private *synchronized* void addToBook(BookSide side, Tradable trd)

This method is a key part of the trading system. This method deals with the addition of Tradables to the Buy/Sell ProductSideBook and handles the results of any trades the result from that addition. This method must do the following:

- If the market state (which can be gotten from the ProductService) is PREOPEN, then just call the "addToBook" of the ProductBookSide (the side specified by the "side" parameter) and pass to it the Tradable "trd" parameter. Then return.
- Create a new temporary HashMap<String, FillMessage> variable to collect the potential Fill Messages, but leave it null for now:

  HashMap<String, FillMessage> allFills = null;
- Now, set that "allFills" HashMap you just created to the results of a call to the "tryTrade(Tradable)" method of the ProductBookSide *on the opposite side* of the "side" parameter passed - in passing the Tradable "trd" parameter. (Opposite Side means: use the BUY side if the "side" passed in is SELL, use the SELL side if the side passed in is BUY)
- Next, if the "allFills" HashMap if *not* null and it is *not* empty (then one or more trades occurred!)
  - Call "updateCurrentMarket()"
  - Calculate and save (as an int) the difference between the Tradable's original volume and the remaining volume (that's the amount that traded).
  - Now, set a new temporary Price variable "lastSalePrice" to the results of a call to the "determineLastSalePrice" method (that method is in this class) passing the "allFills" HashMap as its parameter.
  - Publish a Last Sale message, passing the product symbol from this ProductBook, the last sale price you just determined, and the int quantity traded you previously calculated.
- End of IF (no ELSE needed).
- If the Tradable's remaining volume is greater than zero, then:
  - If the Tradable's price is a MKT Price:
    - Build a new CancelMessage using the data from the "trd" Tradable, and use "Cancelled" as the Cancel Message detail.
    - Publish a Cancel Message (using the MessagePublisher) using the Cancel Message you just created as the message to publish.
  - Else (not MKT) using the "side" parameter passed in, call the appropriate ProductBookSide's "addToBook" method, passing it the Tradable as the parameter.
- End of IF (no ELSE needed).

Done with ProductBook

4. ProductService *class*

The ProductService is the Façade to the entities that make up the Products (Stocks), and the Product Books ("booked" tradables on the Buy and Sell side). All interaction with the product books and the buy and sell sides of a stock's book will go through this Façade. As this is a Façade, this class should be implemented as a thread-safe singleton.

Besides the singleton-related data member, the ProductService will need the following **data** to properly manage the product books:

4.1 As this class must own all the product books, you will need a structure that contains all product books, accessible by the stock symbol name. This can most easily be accomplished by using a HashMap where the key is the String stock symbol, and the value is the prodyctBook for that stock.
Example: private HashMap<String, ProductBook> allBooks = new HashMap< String, ProductBook >();

4.2 As this class must maintain a data member that holds the current market state (CLOSED, PREOPEN, or OPEN). This should be initialized to CLOSED.

Required public ProductService functionality:

- Query Methods:
  These methods perform various queries against the ProductService and return the results of those queries so that the Trading System can make decisions on how to behave.
  Note that most of the methods here require the "synchronized" keyword. An overview of what this synchronized keyword does and why it is needed can be found in Appendix A of this document "Blocking Multithreaded Execution Using 'synchronized'"

4.1 public *synchronized* ArrayList<TradableDTO> getOrdersWithRemainingQty(String userName, String product)
This method will return a List of TradableDTOs containing any orders with remaining quantity for the user and the stock specified. To do this, get the ProductBook from the "allBooks" HashMap using the String stock symbol passed in as the key, and call it's "getOrdersWithRemainingQty(userName)" method. Return the ArrayList of TradableDTOs that method returns.

4.2 public *synchronized* MarketDataDTO getMarketData(String product)
This method will return a List of MarketDataDTO containing the best buy price/volume and sell price/volume for the specified stock product. To do this, get the ProductBook from the "allBooks" HashMap using the String stock symbol passed in as the key, and call it's "getMarketData()" method. Return the MarketDataDTO that method returns.

4.3 public synchronized MarketState getMarketState()
This method should simply return the current product state (OPEN, PREOPEN, CLOSED) value.

4.4 public synchronized String[][] getBookDepth(String product)
This method should first check that the String product symbol is a real stock symbol (check if it is a key in the "allBooks" HashMap") If not throw an exception (i.e., "NoSuchProductException"). Otherwise, get the ProductBook

from the "allBooks" HashMap using the String stock symbol passed in as the key, and call it's "getBookDepth()" method. Return the String[][] array that method returns.

4.5 public synchronized ArrayList<String> getProductList()

This method should simply return an Arraylist containing all the keys in the "allBooks" HashMap (return new ArrayList<String>(allBooks.keySet());).

- Market and Product Service Manipulation Methods:

4.1 public synchronized void setMarketState(MarketState ms)

This method should update the market state to the new value passed in. Some checks are needed though, as the market state has only certain transitions that are legal: CLOSED -> PREOPEN -> OPEN -> CLOSED. You cannot go from CLOSED directly to OPEN, you cannot go from PREOPEN directly to closed:

- First, verify the requested transition is valid using the guidelines above. If not, throw an exception (i.e., "InvalidMarketStateTransition").
- If it is a valid transition, then set the market state data member to the new value.
- Publish a market message, putting the new state value in the Market message.
- If the new state is OPEN, call "openMarket()" on every ProductBook in the "allBooks" HashMap.
- If the new state is CLOSED, call "closeMarket()"on every ProductBook in the "allBooks" HashMap.

4.2 public synchronized void createProduct(String product)

This method will create a new stock product that can be used for trading. This will result in the creation of a ProductBook object, and a new entry in the "allBooks" HashMap. If the String symbol passed in is null or empty, throw an exception (i.e., "DataValidationException"). If the "allBooks" HashMap already has an entry for that stiock symbol, throw an exception(i.e., "ProductAlreadyExistsException"). Otherwise, create a new ProductBook object, and add it to the "allBooks" hashMap using the String stock symbol as the key.

4.3 public synchronized void submitQuote(Quote q)

This method should forward the provided Quote to the appropriate product book.

- First, check if the market is CLOSED. If so, throw an exception (i.e., "InvalidMarketStateException")
- Next, check that the product symbol contained within the Quote object is a real stock symbol (check if it is a key in the "allBooks" HashMap") If not throw an exception (i.e., "NoSuchProductException").
- Otherwise, get the ProductBook from the "allBooks" HashMap using the product symbol contained within the Quote object as the key, and call it's "addToBook(Quote)" method, passing it the Quote object passed into this method.

4.4 public synchronized String submitOrder(Order o)

This method should forward the provided Order to the appropriate product book.

- First, check if the market is CLOSED. If so, throw an exception (i.e., "InvalidMarketStateException")
- Next, check if the market is in PREOPEN state AND the Order's price is a MKT price. You cannot submit MKT orders during PREOPEN. If so, throw an exception (i.e., "InvalidMarketStateException")
- Next, check that the product symbol contained within the Order object is a real stock symbol (check if it is a key in the "allBooks" HashMap") If not throw an exception (i.e., "NoSuchProductException").
- Otherwise, get the ProductBook from the "allBooks" HashMap using the product symbol contained within the Order object as the key, and call it's "addToBook(Order)" method, passing it the Order object passed into this method. Return the String id of the Order.

4.5 public synchronized void submitOrderCancel(String product, BookSide side, String orderId)

This method should forward the provided Order Cancel to the appropriate product book.

- First, check if the market is CLOSED. If so, throw an exception (i.e., "InvalidMarketStateException")
- Next, check that the product symbol passed in is a real stock symbol (check if it is a key in the "allBooks" HashMap") If not throw an exception (i.e., "NoSuchProductException").
- Otherwise, get the ProductBook from the "allBooks" HashMap using the product symbol passed in as the key, and call it's "cancelOrder (Side, String)" method, passing it the Side value and the String id passed into this method.

4.6 public synchronized void submitQuoteCancel(String userName, String product)

This method should forward the provided Quote Cancel to the appropriate product book.

- First, check if the market is CLOSED. If so, throw an exception (i.e., "InvalidMarketStateException")
- Next, check that the product symbol passed in is a real stock symbol (check if it is a key in the "allBooks" HashMap") If not throw an exception (i.e., "NoSuchProductException").
- Otherwise, get the ProductBook from the "allBooks" HashMap using the product symbol passed in as the key, and call it's "cancelQuote (String)" method, passing it the String username value passed into this method.

Done with ProductService

5 TradeProcessorPriceTimeImpl *class* *(implements the TradeProcessor interface)*

The TradeProcessorPriceTimeImpl interface is the interface that a class that will act as a trading processor needs to implement. This Tradeprocessor implementer contains the functionality needed to "execute" actual trades between Tradable objects in this book side using a price-time algorithm (orders are traded in order of price, then in order of arrival).

The TradeProcessorPriceTimeImpl will need the following **data** to properly manage the product books:

5.1 These objects will need to keep track of fill messages, so a HashMap is needed indexed by trade identifier:

```
private HashMap<String, FillMessage> fillMessages = new HashMap<String, FillMessage>();
```

5.2 A TradeProcessorPriceTimeImpl needs to maintain a reference to the ProductBookSide that this object belongs to, so you need a ProductBookSide data member to refer to the book side this object belongs to.

Required TradeProcessorPriceTimeImpl functionality:

- Constructor – This constructor needs to accept a ProductBookSide parameter – a reference to the book side this TradeProcessor belongs to.

- Private "utility" methods:

5.1 *private* String makeFillKey(FillMessage fm)

This method will be used at various times when executing a trade. All trades result in Fill Messages. All Fill Messages need a Trade Key value so the system can tell them apart (which is different than the Id found in the FillMessage). This method should accpet a FillMessage parameter, and from that it will generate and return a String key.

- The Key should be the userName from the fill message PLUS the id from the fill message PLUS the price from the fill message. Key examples: “ANNANNGOOG1238349456467200$641.15”, "REXREXGOOG$641.301238419121813075$641.30”

5.2 *private* boolean isNewFill(FillMessage fm)

This Boolean method checks the content of the “fillMessages” HashMap to see if the FillMessage passed in is a fill message for an existing known trade or if it is for a new previously unrecorded trade. This is done as follows:

- Make a String key for this Fillmessage by calling the “makeFillKey(FillMessage)” method, passing it the “fm” fill message parameter passed in. Save the resulting String.
- If the “fillMessage” HashMap does not have the key you just generated as a key, then return true. This is a new trade.
- Get the FillMessage from the “fillMessage” HashMap by using the String key you just generated as the key to the HashMap. Save the resulting FillMessage in a temporary variable (i.e., “oldFill”)
- If the side in “oldFill” does not equal the side in the Fill Message parameter passed in, return true. This is a new trade.
- If the id in “oldFill” does not equal the id in the Fill Message parameter passed in, return true. This is a new trade.
- Finally, if all these checks pass – this is not a new trade, so return false.

5.3 *private* void addFillMessage(FillMessage fm)

This method should add a FillMessage either to the “fillMessages” HashMap if it is a new trade, or should update an existing fill message is another part of an existing trade. To do this:

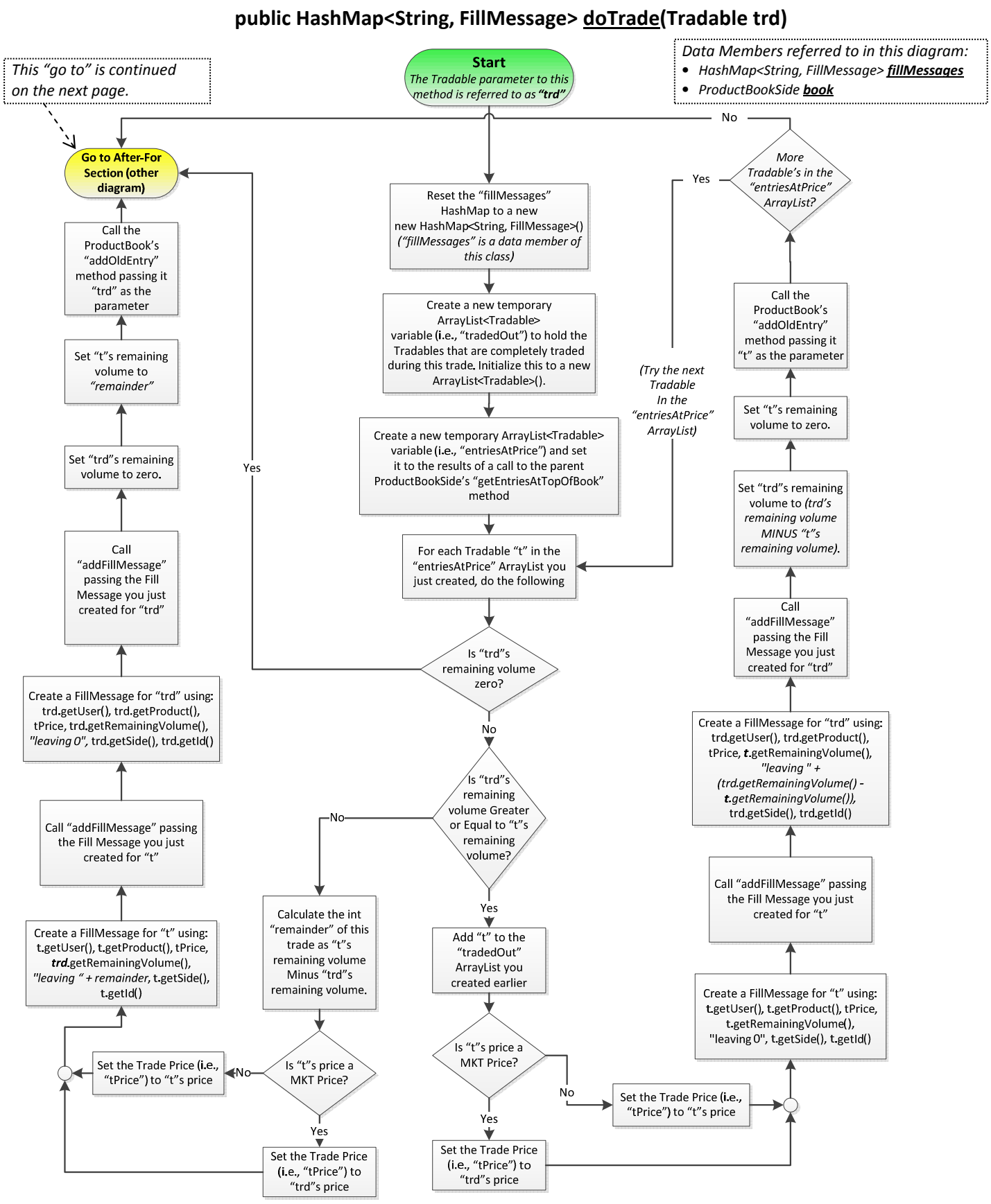
- If a call to “isNewFill(FillMessage)” is true (passing the fill message parameter passed in), then generate a key using “makeFillKey(FillMessage fm)”, and then add this fill message to the “fillMessages” HashMap using the key you generated as the key to the HashMap and the FillMessage as the value.
- Else (not new):
  - Generate a key using “makeFillKey(FillMessage fm)”, and then use that key to get the fill message from the “fillMessages” HashMap.
  - Update the fill volume of that fill message to be the existing FillMessage's volume PLUS the fill volume of the fill message passed in.
  - Update the details of that fill message to be the details of the fill message passed in.
  - *Note – you will need to change the “set” methods for the FillMessage’s fill volume and details data members to “public” so they can be externally modified.*

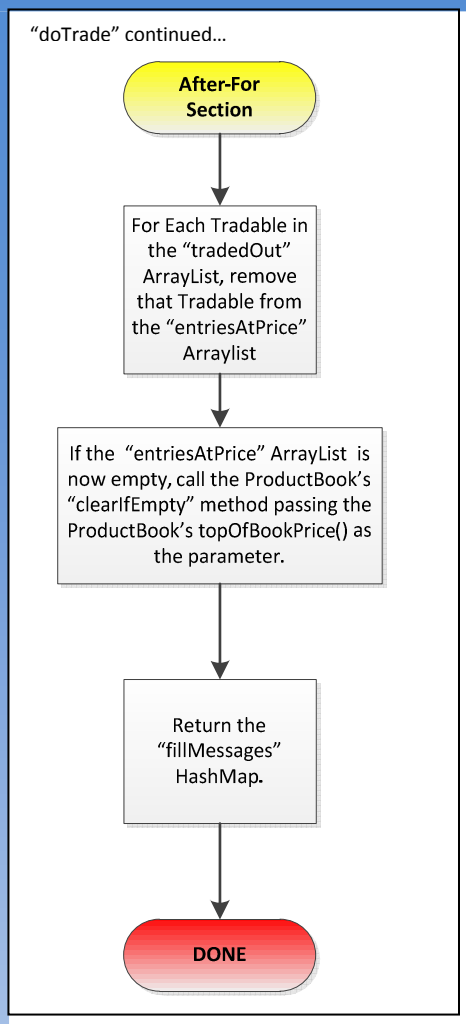
- Interface “doTrade” method:

public HashMap<String, FillMessage> doTrade(Tradable trd)

This TradeProcessor method will be called when it has been determined that a Tradable (i.e., a Buy Order, a Sell QuoteSide, etc.) can trade against the content of the book. The return value from this function will be a HashMap<String, FillMessage> containing String trade identifiers (the key) and a Fill Message object (the value).

The functionality of this method is described on the next page(s) in the form of a flowchart. As the behavior here is a little complex, a diagram is the best way to convey to you its required functionality. Appenxid B contains a walk-thru example of this algorithm. *If you have any questions on this or you get a bit stuck in this implementation please email me.*


public HashMap<String, FillMessage> doTrade(Tradable trd)
This "go to" is continued on the next page.
Start
The Tradable parameter to this method is referred to as "trd"
Data Members referred to in this diagram:
HashMap<String, FillMessage> fillMessages
ProductBookSide book
Go to After-For Section (other diagram)
Reset the "fillMessages" HashMap to a new new HashMap<String, FillMessage>() ("fillMessages" is a data member of this class)
Create a new temporary ArrayList<Tradable> variable (i.e., "tradedOut") to hold the Tradables that are completely traded during this trade. Initialize this to a new ArrayList<Tradable>().
Create a new temporary ArrayList<Tradable> variable (i.e., "entriesAtPrice") and set it to the results of a call to the parent ProductBookSide's "getEntriesAtTopOfBook" method
For each Tradable "t" in the "entriesAtPrice" ArrayList you just created, do the following
Is "trd"s remaining volume zero?
Yes
No
Is "trd"s remaining volume Greater or Equal to "t"s remaining volume?
No
Yes
Calculate the int "remainder" of this trade as "t"s remaining volume Minus "trd"s remaining volume.
Add "t" to the "tradedOut" ArrayList you created earlier
Is "t"s price a MKT Price?
No
Yes
Set the Trade Price (i.e., "tPrice") to "t"s price
Set the Trade Price (i.e., "tPrice") to "trd"s price
Create a FillMessage for "t" using: t.getUser(), t.getProduct(), tPrice, trd.getRemainingVolume(), "leaving " + remainder, t.getSide(), t.getId()
Call "addFillMessage" passing the Fill Message you just created for "t"
Create a FillMessage for "trd" using: trd.getUser(), trd.getProduct(), tPrice, trd.getRemainingVolume(), "leaving 0", trd.getSide(), trd.getId()
Call "addFillMessage" passing the Fill Message you just created for "trd"
Set "trd"s remaining volume to zero.
Set "t"s remaining volume to "remainder"
Call the ProductBook's "addOldEntry" method passing it "trd" as the parameter
(Try the next Tradable In the "entriesAtPrice" ArrayList)
More Tradable's in the "entriesAtPrice" ArrayList?
Yes
No
Create a FillMessage for "t" using: t.getUser(), t.getProduct(), tPrice, t.getRemainingVolume(), "leaving 0", t.getSide(), t.getId()
Call "addFillMessage" passing the Fill Message you just created for "t"
Create a FillMessage for "trd" using: trd.getUser(), trd.getProduct(), tPrice, t.getRemainingVolume(), "leaving " + (trd.getRemainingVolume() - t.getRemainingVolume()), trd.getSide(), trd.getId()
Call "addFillMessage" passing the Fill Message you just created for "trd"
Set "trd"s remaining volume to (trd's remaining volume MINUS "t"s remaining volume).
Set "t"s remaining volume to zero.
Call the ProductBook's "addOldEntry" method passing it "t" as the parameter



Appendix A: Blocking Multithreaded Execution Using 'synchronized'

Many of the methods in this project require the *"synchronized"* keyword. When you synchronize a method, you are indicating that while this method is executing, no other thread can execute a synchronized method at the same time. This prevents 2 or more threads simultaneously modifying the same data structure.
Look at the graphical representation of an ArrayList<String> below. If I wanted to remove element [2] and then insert "Ruby" at position [4] in a single threaded operation, then those operations would occur one-at-a-time:

Original:

ArrayList<String>
[0] "John"
[1] "Ravi"
[2] "Alex"
[3] "Kate"
[4] "Joan"
[5] "Ying"

After remove element [2]

ArrayList<String>
[0] "John"
[1] "Ravi"
[2] "Kate"
[3] "Joan"
[4] "Ying"

After insert "Ruby" at index [4]

ArrayList<String>
[0] "John"
[1] "Ravi"
[2] "Kate"
[3] "Joan"
[4] "Ruby"
[5] "Ying"

These results are predictable. Removing element [2] from the original list removes the "Alex" element. The elements that come after "Alex" ("Kate", "Joan" & "Ying") then all "move up" by one index value. Then inserting a new "Ruby" element at index [4] adds "Ruby" after "Joan" – and pushes "Ying" back by one index.

If these same operations were done in a "multithreaded" environment, then both the remove and the update could come at the same time. It is important to note that these updates – though they might each be done in a single line of code – are actually made up of many low level instructions. When 2 or more operations on the same data element happen at the same time, the low level instructions of the 2 operations become interleaved – and will conflict with each other.
For example, the beginning of the "insert" operation may determine the proper (memory) location to insert "Ruby" just as the "remove" operation begins the removal of element 2. Then the insert then puts "Ruby" at index [4] and will move "Joan" and "Ying" down one index. Then the "remove" operation completes, removing "Alex". But now the end of the list is different than the original example at index values 3 & 4 as can be seen in the below-right diagram.

ArrayList<String>
[0] "John"
[1] "Ravi"
[2] "Kate"
[3] "Ruby"
[4] "Joan"
[5] "Ying"

Only one "synchronized" method can be executed by any thread at one time. So if 2 threads want to execute "synchronized" methods that modify the same data structure, only one can do so at one time. The other thread(s) have to wait until that thread exits the "synchronized' method before it gets its turn.

Synchronization and other multithreading concepts will be discussed in an upcoming class.

Appendix B: A Walkthrough of the "doTrade" Algorithm

Assume that the orders and quotesides that have come in and booked for MSFT (Microsoft) stock have filled the MSFT book as is shown below:

MSFT (Microsoft) Book			
BUY		SELL	
120	$29.05	$29.07	90
100	$29.01	$29.08	110
90	$28.94	$29.10	70

Then, a BUY order to for 75 @ $29.07 arrives – which is tradable with the SELL order at $29.07 already in the book. That incoming BUY order to for 75 @ $29.07 is passed into the "doTrade" method as the "trd" parameter, and will be referred to as "trd" in the below:

doTrade(Tradable trd) *(Follow along with the flowchart for "doTrade")*

1. Reset the "fillMessages" data member: *fillMessages = new HashMap<String, FillMessage>();*
2. Create a new temporary ArrayList<Tradable> called "tradedOut" to hold the Tradables that are completely traded during this trade. Initialize this to a new ArrayList<Tradable>(): *ArrayList<Tradable> tradedOut = new ArrayList<Tradable>();*
3. Create a new temporary ArrayList<Tradable> variable called "entriesAtPrice" and set it to the results of a call to the parent ProductBookSide's "getEntriesAtTopOfBook" method: *ArrayList<Tradable> entriesAtPrice = book.getEntriesAtTopOfBook();*
   Assume that there are 2 orders in that entriesAtPrice ArrayList that make up the 90 volume at the $29.07 price: 50 qty (from user ABC), and 40 qty (from user XYZ)
4. For each Tradable "t" in the "entriesAtPrice" ArrayList, do the following: *(So in the first loop iteration, "t" is the resting order for 50 @ $29.07)*
5. If "trd"s remaining volume zero? No – it is the same as the original volume = 75. Follow the "No" path.
6. Is "trd"s remaining volume greater than or equal to "t"s remaining volume? Yes – 75 is greater than 50. Follow the "Yes" path.
7. Add "t" to the "tradedOut" ArrayList. `tradedout:` `[0] 50 @ $29.07`
8. Is "t"s price a MKT Price? No – it's a limit price of $29.07. follow the "No" path.
9. Set the "tPrice", the Trade Price, to "t"s price, $29.07: *Price tPrice = t.getPrice();*
10. Create a FillMessage for "t" using: t.getUser(), t.getProduct(), tPrice, t.getRemainingVolume(), "leaving 0", t.getSide(), t.getId()
    FillMessage tFill = new FillMessage(t.getUser(), t.getProduct(), tPrice, t.getRemainingVolume(), "leaving " + 0 , t.getSide(), t.getId());
11. Call "addFillMessage" passing the Fill Message you just created for "t". This adds the Fill Message to the "fillMessages" HashMap we initialized in Step #1.
12. Create a FillMessage for "trd" using: trd.getUser(), trd.getProduct(), tPrice, t.getRemainingVolume(), "leaving " + (trd.getRemainingVolume() - t.getRemainingVolume()), trd.getSide(), trd.getId()
    FillMessage trdFill =
    new FillMessage (trd.getUser(), t.getProduct(), tPrice, t.getRemainingVolume(), "leaving " + (trd.getRemainingVolume() - t.getRemainingVolume()), trd.getSide(), trd.getId());
13. Call "addFillMessage" passing the Fill Message you just created for "trd". This adds the Fill Message to the "fillMessages" HashMap we initialized in Step #1.
14. Set "trd"s remaining volume to (trd's remaining volume MINUS "t"s remaining volume). So "trd's remaining volume is 75 – 50 = 25.
    trd.setRemainingVolume(trd.getRemainingVolume() - t.getRemainingVolume());
15. Set "t"s remaining volume to zero.
16. Call the parent ProductBookSide's "addOldEntry" method, passing it the "t" Tradable as the parameter.

17. Are there more Tradables in the "entriesAtPrice" ArrayList? Yes – one more, 40 qty at $29.07. Go back to the top of the For Each loop (step #4 above) with the next Tradable (the 40 qty).
18. Back up at the For Each loop again. *(So in this iteration, "t" is the other resting order for 40 @ $29.07)*
19. If "trd"s remaining volume zero? No – it is now 25. Follow the "No" path.
20. Is "trd"s remaining volume greater than or equal to "t"s remaining volume? No – 25 is not greater than 40. Follow the "No" path.
21. Calculate the int "remainder" of this trade as "t"s remaining volume Minus "trd"s remaining volume. 40 – 25 = 15
    int remainder = t.getRemainingVolume() - trd.getRemainingVolume();
22. Is "t"s price a MKT Price? No – it's a limit price of $29.07. follow the "No" path.
23. Set the "tPrice", the Trade Price, to "t"s price, $29.07: *Price tPrice = t.getPrice();*
24. Create a FillMessage for "t" using: t.getUser(), t.getProduct(), tPrice, ***trd***.getRemainingVolume(), *"leaving " + remainder*, t.getSide(), t.getId()
    *FillMessage tFill = new FillMessage(t.getUser(), t.getProduct(), tPrice, **trd**.getRemainingVolume(), "leaving " + remainder, t.getSide(), t.getId());*
25. Call "addFillMessage" passing the Fill Message you just created for "t". This adds the Fill Message to the "fillMessages" HashMap we initialized in Step #1.
26. Create a FillMessage for "trd" using: trd.getUser(), trd.getProduct(), tPrice, trd.getRemainingVolume(), *"leaving 0"*, trd.getSide(), trd.getId()
    FillMessage trdFill = new FillMessage(trd.getUser(), t.getProduct(), tPrice, trd.getRemainingVolume(), "leaving " + 0, trd.getSide(), trd.getId());
27. Call "addFillMessage" passing the Fill Message you just created for "trd". This adds the Fill Message to the "fillMessages" HashMap we initialized in Step #1.
28. Set "trd"s remaining volume to zero. *trd.setRemainingVolume(0);*
29. Set "t"s remaining volume to remainder" (calculated in step #21). *t.setRemainingVolume(remainder);*
30. Call the parent ProductBookSide's "addOldEntry" method, passing it the "trd" Tradable as the parameter.
31. Go to After-For Section (other diagram)
32. For Each Tradable in the "tradedOut" ArrayList, remove that Tradable from the "entriesAtPrice" Arraylist. *In this case, there is only one – the 50 qty @ $29.07.*
33. If the "entriesAtPrice" ArrayList is now empty, call the ProductBook's "clearIfEmpty" method passing the ProductBook's topOfBookPrice() as the parameter. *It is not empty – we left 15 of the original 40 qty of that resting order.*
34. Return the "fillMessages" Hashmap
35. Done!

Testing Phase 4

A test "driver" class with a "main" method will be provided (*soon*) that will exercise the functionality of your Phase 4 classes. *This will not exhaustively test your classes* but successful execution is a good indicator that your classes are performing as expected. This driver will be posted soon.

Phases & Schedule

The Course Programming Project will be implemented in phases, each with a specific duration and due date as is listed below. Detailed documents on each phase will be provided at the beginning of the phase.

Phase 1 (1 Week) 9/17 – 9/24:

- Price & Price Factory

Phase 2 (1 Week) 9/24 – 10/1:

- Tradable & Tradable DTO
- Order
- Quote & Quote Side

Phase 3 (2 Weeks) 10/1 – 10/15: [Midterm 10/8]

- Current Market Publisher
- Last Sale Publisher
- Ticker Publisher
- Message Publisher
- Fill Message
- Cancel Message
- Market Message
- User (interface)

Phase 4 (2 Weeks) 10/15 – 10/29:

- **Product Service**
- **Book & Book Side**
- **Trade Processor**

Phase 5 (1 Week) 10/29 – 11/5:

- User Implementation
- User Command Service

Phase 6 (1 Week) 11/5 – 11/12:

- User Interface GUI
- Simulated Traders

[Final Exam 11/19]