Course Programming Project

DePaul Stock Exchange (DSX)

Phase 3

Messages, User (interface), Publishers, & DTOs

Phase 3 of the Course Programming Project involves the creation of several classes that will deal with the transfer of information between the trading system and the users. Several "publisher" classes will be added that will publish messages related to the stock market to connected users. These messages will be contained in a set of "message" classes that will bundle up related information for delivery to the users. Additionally, a new DTO class will be created to package up a number of stock market data elements for easy transfer between parts of the system. Finally, a new interface will be created that defines the functions that any "User" of the system must implement (that implementation will happen in a later phase).

Note, that many of the entities in this project need to maintain an indicator of which "side" they represent: BUY or SELL. How you represent this is up to you – enumerated type, constants, etc. In this and other handouts, I will refer to this "side" attribute with the generic type name of "BookSide". Wherever you see "BookSide", you just need to keep in mind that this refers to the type you have decided to use to represent the BUY/SELL side.

1. Messages

The message classes created as a part of Phase 3 include the CancelMessage, FillMessage, and MarketMessage. These classes will work closely with the Publishers described later in this handout, so they should all be put into the same package (i.e., "publishers"). These message classes are relatively simple classes whose purpose is to encapsulate a set of related data items for easy transfer as a return value or a method parameter within the trading system. Note - there are common data and behaviors among some of the message classes so you might want to consider using inheritance or using an interface and delegation.

1A CancelMessage

The CancelMessage class encapsulates data related to the cancellation of an order or quote-side by a user, or by the trading system. CancelMessage data elements should be as follows:

- private String user – The String username of the user whose order or quote-side is being cancelled. Cannot be null or empty.
- private String product – The string stock symbol that the cancelled order or quote-side was submitted for ("IBM", "GE", etc.). Cannot be null or empty.
- private Price price – The price specified in the cancelled order or quote-side. Cannot be null.

- private int volume – The quantity of the order or quote-side that was cancelled. Cannot be negative.
- private String details – A text description of the cancellation. Cannot be null.
- private *BookSide* side – The side (BUY/SELL) of the cancelled order or quote-side. Must be a valid side.
- public String id – The String identifier of the cancelled order or quote-side. Cannot be null.

➢ The CancelMessage class must allow the public "get" (access) of each of these data elements. Modifiers (sets) should be created as private and should be used internally within the class. These should throw an appropriate exception if the data passed into them is invalid.

➢ Additionally, the CancelMessage should implement the **Comparable<CancelMessage>** interface. For example: `public class CancelMessage implements Comparable<CancelMessage>`. The CancelMessage must then implement a "compareTo(CancelMessage cm)" method to satisfy that interface. All this needs to do is return the result of calling the "compareTo" of the Price object from the current CancelMessage, using the Price of the CancelMessage passed into this method as the parameter to "compareTo".

NOTE: You should be sure to add a "toString" to the CancelMessage that displays the information content of the object. For example:

User: REX, Product: SBUX, Price: $52.00, Volume: 140, Details: Cancelled By User, Side: BUY, Id: ABC123XYZ

1B FillMessage

The FillMessage class encapsulates data related to the fill (trade) of an order or quote-side. FillMessage data elements are the same as the CancelMessage, and are should be as follows:

- private String user – The String username of the user whose order or quote-side was filled. Cannot be null or empty.
- private String product – The string stock symbol that the filled order or quote-side was submitted for ("IBM", "GE", etc.). Cannot be null or empty.
- private Price price – The price that the order or quote-side was filled at. Cannot be null.
- private int volume – The quantity of the order or quote-side that was filled. Cannot be negative.
- private String details – A text description of the fill (trade). Cannot be null.
- private *BookSide* side – The side (BUY/SELL) of the filled order or quote-side. Must be a valid side.
- public String id – The String identifier of the filled order or quote-side. Cannot be null.

➢ The FillMessage class must allow the public "get" (access) of each of these data elements. Modifiers (sets) should be created as private and should be used internally within the class. These should throw an appropriate exception if the data passed into them is invalid.

Additionally, the FillMessage should implement the **Comparable<FillMessage>** interface. The FillMessage must then implement a "compareTo(FillMessage fm)" method to satisfy that interface. All this needs to do is return the result of calling the "compareTo" of the Price object from the current FillMessage, using the Price of the FillMessage passed into this method as the parameter to "compareTo".

NOTE: You should be sure to add a "toString" to the FillMessage that displays the information content of the object. For example:

User: REX, Product: SBUX, Price: $52.00, Volume: 140, Details: Cancelled By User, Side: BUY

1C MarketMessage

The MarketMessage class encapsulates data related to the "state" of the market. The possible market states are: **CLOSED**, **PREOPEN**, and **OPEN**. *Only* these values are legal market states. MarketMessage data (only one) should be as follows:

- private *[MARKET STATE]* state – This data element should hold a market state value (one of CLOSED, PREOPEN, and OPEN). *How you represent [MARKET STATE] is up to you (constants, enumerated type, etc).*

➢ The MarketMessage must allow the public "get" (access) of its data element. A modifier (set) method should be created as private and should be used internally within the class. This should throw an exception if the data is invalid (is it is even possible to have a bad value).

➢ The MarketMessage class *does not* need to implement the Comparable interface. It does not need a toString method buy you may add one if you want to.

2. User (interface only)

The "User" interface contains the method declarations that any class that wishes to be a "User" of the DSX trading system must implement. Though we will use this User *type* for data members and method parameters in Phase 3, we will not actually have any implementers. Implementers will come in a later phase. To keep this interface separate from your other classes, this interface should be put into its own package (i.e., "client"). More classes will be added to that package later in the project.

Required User Interface Behaviors:

- String getUserName() – This will return the String username of this user

- void acceptLastSale(String product, Price p, int v) – This will accept a String stock symbol ("IBM, "GE", etc), a Price object holding the value of the last sale (trade) of that stock, and the quantity (volume) of that last sale. This info is used by "Users" to track stock sales and volumes and is sometimes displayed in a GUI.

- void acceptMessage(FillMessage fm) – This will accept a FillMessage object which contains information related to an order or quote trade. This is like a receipt sent to the user to document the details when an order or quote-side of theirs trades.

- void acceptMessage(CancelMessage cm) – This will accept a CancelMessage object which contains information related to an order or quote cancel. This is like a receipt sent to the user to document the details when an order or quote-side of theirs is canceled.

- void acceptMarketMessage(String message) – This will accept a String which contains market information related to a Stock Symbol they are interested in.

- void acceptTicker(String product, Price p, char direction) – This will accept a stock symbol ("IBM, "GE", etc), a Price object holding the value of the last sale (trade) of that stock, and a "char" indicator of whether the "ticker" price represents an increase or decrease in the Stock's price. This info is used by "users" to track stock price movement, and is sometimes displayed in a GUI.

- void acceptCurrentMarket(String product, Price bp, int bv, Price sp, int sv) – This will accept a String stock symbol ("IBM, "GE", etc.), a Price object holding the current BUY side price for that stock, an int holding the current BUY side volume (quantity), a Price object holding the current SELL side price for that stock, and an int holding the current SELL side volume (quantity). These values as a group tell the user the "current market" for a stock. For example:

  ```
  AMZN:  BUY 220@12.80 and SELL 100@12.85.
  ```

  This info is used by "Users" to update their market display screen so that they are always looking at the most current market data.

3. MarketDataDTO

The MarketDataDTO class is based upon the "Data Transfer Object" pattern. This DTO will be used to encapsulate a set of data elements that detail the values that make up the current market. This information is needed by many components of our trading system so rather then pass "real" market data objects from the trading system, or pass each of the 5 data elements individually, this DTO will be used to facilitate the data transfer. The MarketDataDTO should contain the following information:

- public String product - The stock product (i.e., IBM, GOOG, AAPL, etc.) that these market data elements describe.
- public Price buyPrice – The current BUY side price of the Stock
- public int buyVolume – The current BUY side volume (quantity) of the Stock
- public Price sellPrice – The current SELL side price of the Stock
- public int sellVolume – The current SELL side volume (quantity) of the Stock

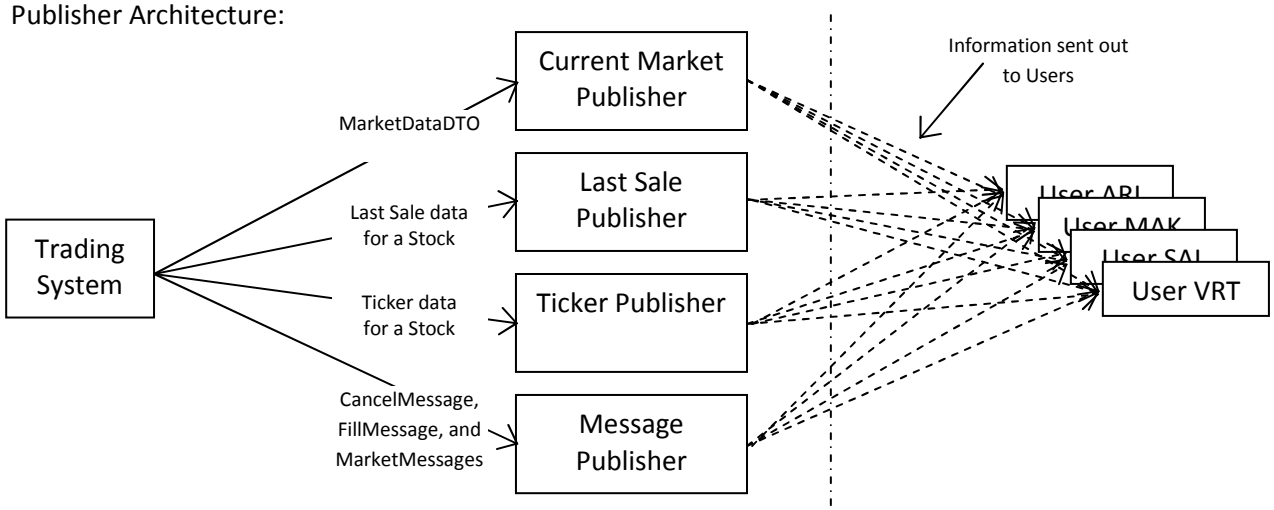
NOTE: It is advisable to add a "toString" to the MarketDataDTO for debugging purposes.

4. Publishers

The publishers created as a part of Phase 3 include the CurrentMarketPublisher , LastSalePublisher, TickerPublisher & MessagePublisher. These publishers are used by the DSX trading system to send various messages out to the users. Thesee classes will work closely with the messages described earlier in this handout, so they should all be put into the same package (i.e., "publishers").

The publishers use the "Observer" design pattern so they follow a publish-subscribe paradigm. Users will "subscribe" for current market messages, for last sale messages, for ticker messages, and for cancel, fill, and market messages. When information in the trading system changes (the current market for a stock changes, a trade occurs, an order is cancelled, etc) these publishers will "publish" messages to their subscribers if the message is relevant to them.

Publisher Architecture:

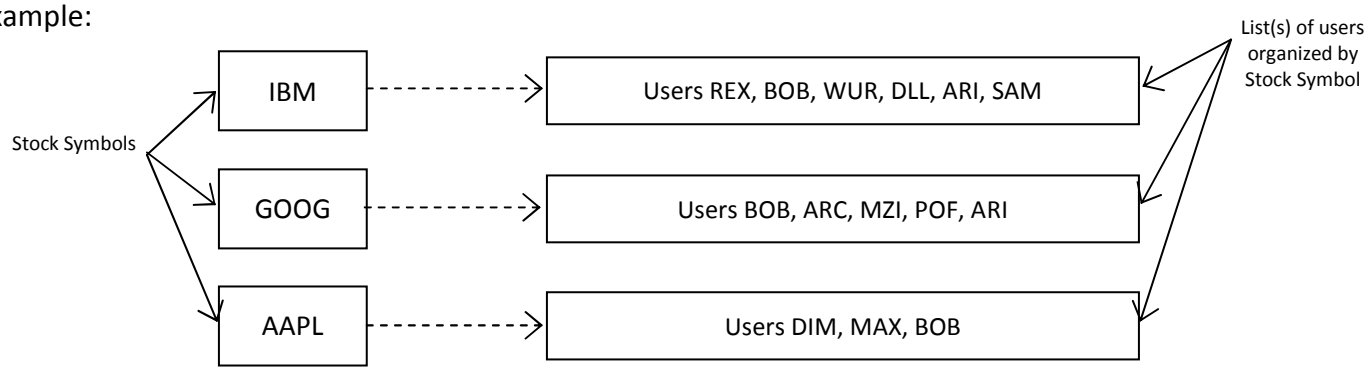


The Observer Implementation

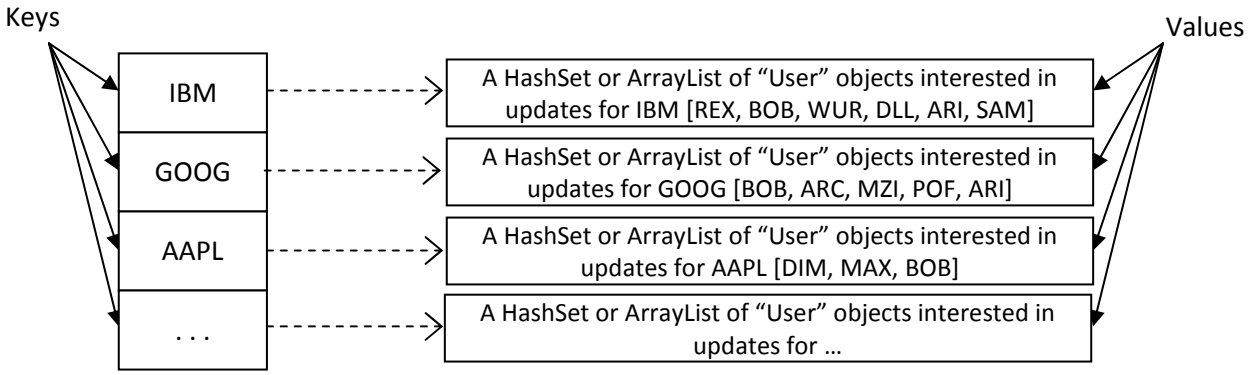
All publishers need a way to keep track of "subscribed" users. Users will "subscribe" for the data the publisher generates, so the publisher needs to know who is subscribed to that data, so it can be sent out to them.

Each of our publishers will need a similar structure to maintain subscriptions/subscribers. Subscribers will "subscribe" for data (current market, last sale, ticker, and market messages) related to a stock. Your publishers must maintain a list of subscribers organized by the stock symbol in which they are interested. For example, the CurrentMarketPublisher will need to store the subscribers by the stock for which they want current market updates.

Example:



One suggested data structure to deal with this is a Java HashMap (key/value structure) that uses the String stock symbol as the "key", and uses a HashSet or ArrayList of Users as the "value" (be SURE to look up HashMap and HashSet (or ArrayList) if you are not familiar with these classes):



Common Observer Functionality

All of our publishers will need a "subscribe" method for users to call so they can subscribe for data. When calling "subscribe", users will provide a "User" reference (a reference to themselves), and the String stock symbol they are interested. If a user wants data on multiple stocks, they will need to make multiple subscriptions. The subscribe method needs to be "synchronized" (covered later in the course), as multiple user threads will be attempting to subscribe and unsubscribe for data:

public **synchronized** void subscribe(User u, String product) { … }

This "subscribe" method should do the following:

- If the user is already subscribed for the stock, throw an exception (i.e., "AlreadySubscribedException").
- Otherwise, add the User to the HashSet or ArrayList related to the provided stock symbol.

Additionally, all of our publishers will need an "unSubscribe" method for users to call to un-subscribe for data as well. When calling "unSubscribe", users will provide a "User" reference (a reference to themselves), and the String stock symbol they wish to un-subscribe from. If they have multiple subscriptions, they will need to make multiple unSubscribe calls. The unSubscribe method needs to be "synchronized" (covered later in the course), as multiple user threads will be attempting to subscribe and unsubscribe for data:

public **synchronized** void unSubscribe(User u, String product) { … }

The "unSubscribe" method should do the following:

- If the user is not subscribed for the stock, throw an exception (i.e., "NotSubscribedException").
- Otherwise, remove the User from the HashSet or ArrayList related to the provided stock symbol.

Publisher-Specific Functionality

4A CurrentMarketPublisher

The CurrentMarketPublisher should implement the "Singleton" design pattern, as we only want to have a single instance of the CurrentMarketPublisher. Besides the subscribe/unsubscribe functionality common to all publishers, the CurrentMarketPublisher will need a method that the components of the trading system will call to send the market updates out – "publishCurrentMarket". The "publishCurrentMarket" callers will provide a "MarketDataDTO" reference as a parameter to the method that contains the market update data. The "publishCurrentMarket" method needs to be "synchronized" (covered later in the course), as it makes use of the subscriptions HashMap - so that should be temporarily locked during use:

public **synchronized** void publishCurrentMarket(MarketDataDTO md)

The "publishCurrentMarket" method should do the following:

- For each User object in the HashSet or ArrayList *for the specified stock symbol* (the stock symbol can be found in the MarketDataDTO), do the following:
    - Call the User object's "acceptCurrentMarket" method passing the following 5 parameters:
        - The String stock symbol taken from the MarketDataDTO passed in.
        - The BUY side price taken from the MarketDataDTO passed in. (If this is null, a Price object representing $0.00 should be used, do NOT send null to the users)
        - The BUY side volume taken from the MarketDataDTO passed in.
        - The SELL side price taken from the MarketDataDTO passed in. (If this is null, a Price object representing $0.00 should be used, do NOT send null to the users)
        - The SELL side volume taken from the MarketDataDTO passed in.

4B LastSalePublisher

The LastSalePublisher should implement the "Singleton" design pattern, as we only want to have a single instance of the LastSalePublisher. Besides the subscribe/unsubscribe functionality common to all publishers, the LastSalePublisher will need a method that the components of the trading system will call to send the last sales out – "publishLastSale". The "publishLastSale" callers will provide a String stock symbol, a Price holding the last sale price, and an "int" holding the last sale volume a parameter to the method. The "publishLastSale" method needs to be "synchronized" (covered later in the course), as it makes use of the subscriptions HashMap - so that should be temporarily locked during use:

public **synchronized** void publishLastSale(String product, Price p, int v)

The "publishLastSale" method should do the following:

- For each User object in the HashSet or ArrayList *for the specified stock symbol* (i.e., the product), do the following:
  - Call the User object's "acceptLastSale" method passing the following 3 parameters:
    - The String stock symbol passed into "publishLastSale" (i.e., the product).
    - The last sale price passed into "publishLastSale" (If this is null, a Price object representing $0.00 should be used, do NOT send null to the users)
    - The last sale volume passed into "publishLastSale".
- Then, after the "for" loop, make a call to the TickerPublisher's "publishLastSale" method, passing it the "product" symbol String, and the Price "p" that were passed into this method.

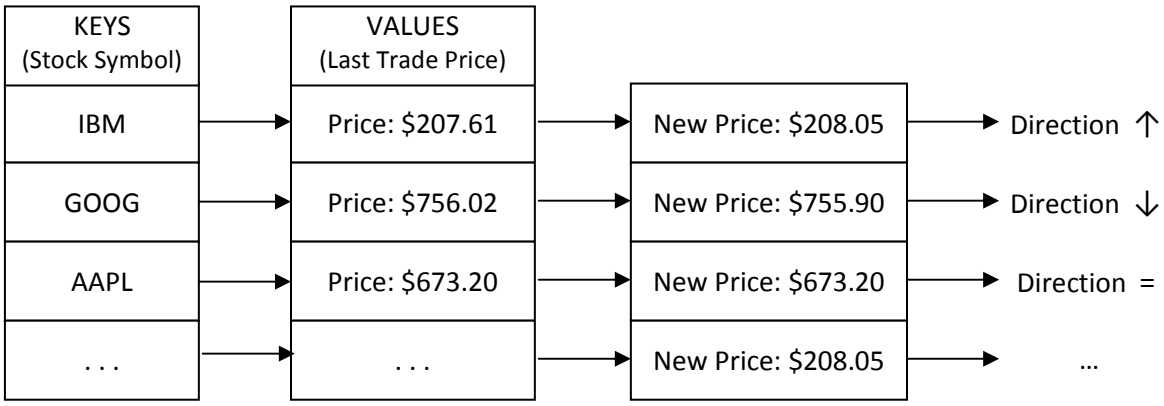
4C TickerPublisher

The TickerPublisher should implement the "Singleton" design pattern, as we only want to have a single instance of the TickerPublisher. Besides the subscribe/unsubscribe functionality common to all publishers, the TickerPublisher will need a method that the components of the trading system will call to send the last sales out – "publishTicker". The "publishTicker" callers will provide a String stock symbol, and a Price holding the last sale price a parameter to the method. The "publishTicker" method needs to be "synchronized" (covered later in the course), as it makes use of the subscriptions HashMap - so that should be temporarily locked during use:

public **synchronized** void publishTicker(String product, Price p)

NOTE that the TickerPublisher differs from the previous publishers in that it will need to save the most recent ticker value for each stock symbol so it can determine if the stock price have moved up, down, or stayed the same. You might consider using a HashMap to do this, using the String stock symbol as the key, and the last trade Price object as the value.

Example:



The "publishTicker" method should do the following:

- Determine if the new trade Price for the provided stock symbol (product) is greater than (up), less than (down), or the same (equal) as the previous trade price seen for that stock. If the price has moved up, the up-arrow character '↑' should be sent to the user. If the price has moved down, the down-arrow character '↓' should be sent to the user. If the price is the same as the previous price, the equals character '=' should be sent to the user. If there is no previous price for the specified stock, the space character ' ' should be sent to the user. These symbols will be used in the next step.

- Then, for each User object in the HashSet or ArrayList *for the specified stock symbol* (i.e., the product), do the following:

    - Call the User object's "acceptTicker" method passing the following 3 parameters:
        - The String stock symbol passed into "publishTicker" (i.e., the product).
        - The price passed into "publishTicker" (If this is null, a Price object representing $0.00 should be used, do NOT send null to the users)
        - The characted previously determined that represents the movement of the stock (↑,↓,=,<space>).

4D MessagePublisher

The MessagePublisher should implement the "Singleton" design pattern. Besides the subscribe/unsubscribe functionality common to all publishers, the MessagePublisher will need several methods that the components of the trading system can call to send the messages out – "publishCancel(CancelMessage cm)", "publishFill(FillMessage fm)", and "publishMarketMessage(MarketMessage nn)". Each of these methods needs to be "synchronized"

(covered later in the course), as they make use of the subscriptions HashMap - so that should be temporarily locked during use:

public **synchronized** void publishCancel(CancelMessage cm)

The "publishCancel" method should do the following:

- Find the individual User object in the HashSet or ArrayList *for the specified stock symbol* whose user name matches the user name found in the CancelMessage object passed into "publishCancel". The user name and stock symbol are found in the provided CancelMessage.
- Once found, call the User object's "acceptMessage" passing the CancelMessage object that was passed in.

public **synchronized** void publishFill(FillMessage fm)

The "publishFill" method should do the following:

- Find the individual User object in the HashSet or ArrayList *for the specified stock symbol* whose user name matches the user name found in the FillMessage object passed into "publishFill". The user name and stock symbol are found in the provided FillMessage.
- Once found, call the User object's "acceptMessage" passing the FillMessage object that was passed into "publishFill"

- public **synchronized** void publishMarketMessage(MarketMessage mm)

The "publishMarketMessage" method should do the following:

- For *all* subscribed Users *regardless of the stock symbol they are interested in*, do the following:
  - Call the User object's "acceptMarketMessage" passing the String generated by calling the "toString" method of the MarketMessage object passed in.

Testing Phase 3

A test "driver" class with a "main" method will be provided that will exercise the functionality of your Phase 3 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]