Course Programming Project

DePaul Stock Exchange (DSX)

Phase 5

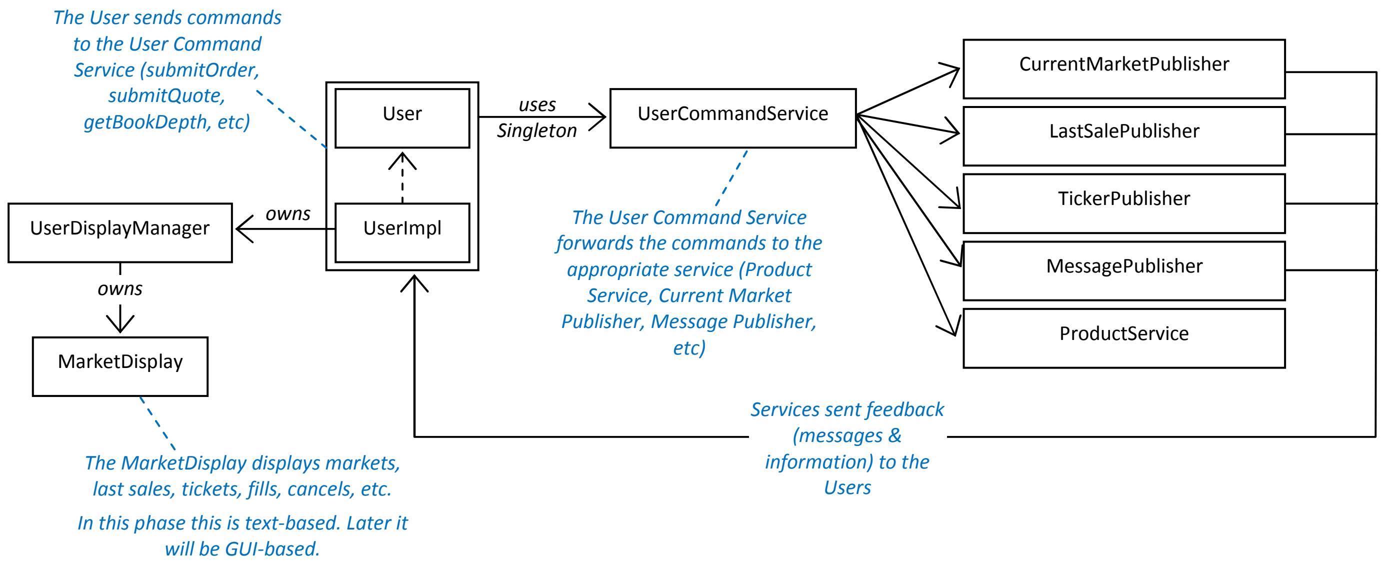
User (interface update), UserImpl (implementation), TradableUserData, Position, and UserCommandService classes

Plus – addition of 2 provided classes (UserDisplayManager & MarketDisplay);

Phase 5 of the Course Programming Project involves the creation of the classes that will handle the User to Trading System interaction.

- New/Updated Classes:
  - The **User** interface adds new methods that add user inputs and queries
  - The **UserImpl** class is a full implementation of the User interface.
  - The **TradableUserData** class will be used by Users to hold the details of Orders and Quotes they have entered into the Trading System.
  - The **Position** class holds the gains, losses, and costs that user has incurred while trading.
  - The **UserCommandService** class is the User's façade to the Trading System.

- Provided Classes to Integrate into your Project ("gui" package)
  - The **MarketDisplay** class is a text-based interface to the Trading System.
  - The **UserDisplayManager** class is a façade to the Market Display.


The User sends commands to the User Command Service (submitOrder, submitQuote, getBookDepth, etc)
User
uses Singleton
UserCommandService
CurrentMarketPublisher
LastSalePublisher
TickerPublisher
MessagePublisher
ProductService
UserDisplayManager
owns
UserImpl
owns
MarketDisplay
The User Command Service forwards the commands to the appropriate service (Product Service, Current Market Publisher, Message Publisher, etc)
Services sent feedback (messages & information) to the Users
The MarketDisplay displays markets, last sales, tickets, fills, cancels, etc.
In this phase this is text-based. Later it will be GUI-based.

Phase 5 Class Detail

1. User *interface*

The User interface (from the "client" package) will be expanded to include messages that send requests into the trading system. The existing methods on the User interface (shown below) were designed to accept data from the trading system. The new (additional) methods are designed to allow a user to submit requests into the trading system

Existing *(these are already in your User interface):*

1. public String getUserName()
2. public void acceptLastSale(String product, Price p, int v)
3. public void acceptMessage(FillMessage fm)
4. public void acceptMessage(CancelMessage cm)
5. public void acceptMarketMessage(String message)
6. public void acceptTicker(String product, Price p, char direction)
7. public void acceptCurrentMarket(String product, Price bp, int bv, Price sp, int sv)

New (to be added to the existing interface):

1. void connect() – *Instructs a User object to connect to the trading system.*
2. void disConnect()– *Instructs a User object to disconnect from the trading system.*
3. void showMarketDisplay() – *Requests the opening of the market display if the user is connected.*
4. String submitOrder(String product, Price price, int volume, BookSide side) – *Allows the User object to submit a new Order request*
5. void submitOrderCancel(String product, BookSide side, String orderId) – *Allows the User object to submit a new Order Cancel request*
6. void submitQuote(String product, Price buyPrice, int buyVolume, Price sellPrice, int sellVolume) – *Allows the User object to submit a new Quote request*
7. void submitQuoteCancel(String product) – *Allows the User object to submit a new Quote Cancel request*
8. void subscribeCurrentMarket(String product) – *Allows the User object to subscribe for Current Market for the specified Stock.*
9. void subscribeLastSale(String product) – *Allows the User object to subscribe for Last Sale for the specified Stock.*
10. void subscribeMessages(String product) – *Allows the User object to subscribe for Messages for the specified Stock.*
11. void subscribeTicker(String product) – *Allows the User object to subscribe for Ticker for the specified Stock.*
12. Price getAllStockValue() – *Returns the value of the all Sock the User owns (has bought but not sold)*
13. Price getAccountCosts() – *Returns the difference between cost of all stock purchases and stock sales*
14. Price getNetAccountValue() – *Returns the difference between current value of all stocks owned and the account costs*
15. String[][] getBookDepth(String product) – *Allows the User object to submit a Book Depth request for the specified stock.*
16. String getMarketState() – *Allows the User object to query the market state (OPEN, PREOPEN, CLOSED).*
17. ArrayList<TradableUserData> getOrderIds() – *Returns a list of order id's for the orders this user has submitted.*
18. ArrayList<String> getProductList() – *Returns a list of the stock products available in the trading system.*
19. Price getStockPositionValue(String sym) – *Returns the value of the specified stock that this user owns*
20. int getStockPositionVolume(String product) – *Returns the volume of the specified stock that this user owns*
21. ArrayList<String> getHoldings()– *Returns a list of all the Stocks the user owns*
22. ArrayList<TradableDTO> getOrdersWithRemainingQty(String product) – *Gets a list of DTO's containing information on all Orders for this user for the specified product with remaining volume.*

2. TradableUserData *class*

The TradableUserData class (for the "client" package) will hold selected data elements related to the Tradables a user has submitted to the system. (Currently it will only be used for Orders). A list of these objects will help the user know what orders they have in the system, and each object contains just enough information to use to generate a cancel requires if needed.
This class needs 4 data members: a String for the user name, a String for the stock symbol, a data member for the "side", and a String to hold an order id. All data members should have public accessors and private modifiers. There should be a constructor that accepts 4 parameters that will be used to set each of the 4 data members (be sure to call your modifiers in the constructor).

3. UserImpl *class*

The UserImpl class (for the "client" package) is our application's implementation of the "User" interface. This represents a "real" user in the trading system; many of these objects can be active in our system.

Data

The UserImpl class will need the following data members to support the various behaviors required by a user of the trading system:

1. A String to hold their user name
2. A long value to hold their "connection id" – provided to them when they connect to the system.
3. A String list of the stocks available in the trading system. The user fills this list once connected based upon data received from the trading system.
4. A list of TradableUserData objects that contains information on the orders this user has submitted (needed for cancelling).
5. A reference to a Position object (part of this assignment) which holds the values of the users stocks, costs, etc.
6. A reference to a UserDisplayManager object (part of this assignment) that acts as a façade between the user and the market display.

Constructor

- The UserImpl constructor should accept a String user name that should be user to set the user name data member. In addition this constructor should create a new Position object and use that to set the Position data member.

Methods (implementations from the User interface)

1. public String getUserName() – This method should return the String username of this user.

2. public void acceptLastSale(String product, Price price, int volume) – This method should call the user display manager's updateLastSale method, passing the same 3 parameters that were passed in. Then, call the Position object's updateLastSale method passing the product and price passed in. *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

3. public void acceptMessage(FillMessage fm) – This method will display the Fill Message in the market display and will forward the data to the Position object:
   - o Create a "Timestamp" object (import java.sql.Timestamp) and set it to: new Timestamp(System.currentTimeMillis())
   - o Create String summary of the Fill Message passed in (for the Market Display), in the format:
     {2012-10-25 09:27:36.243} Fill Message: SELL 150 IBM at $10.00 leaving 0 [Tradable Id: USR2IBM$10.00710011959469681]
   - o Call the user display manager's updateMarketActivity method, passing String summary you just created.

- Then, call the Position object's updatePosition method passing the stock symbol, fill price, side, and fill volume from the FillMessage passed in.
- *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

4. public void acceptMessage(CancelMessage cm) – This method will display the Cancel Message in the market display:
    - Create a "Timestamp" object (import java.sql.Timestamp) and set it to: new Timestamp(System.currentTimeMillis())
    - Create String summary of the Fill Message passed in (for the Market Display), in the format:

    {2012-10-25 09:37:28.238} Cancel Message: BUY 100 GE at $40.00 BUY Order Cancelled [Tradable Id: USR0GE$40.00710603957963725]
    - Call the user display manager's updateMarketActivity method, passing String summary you just created.
    - *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

5. public void acceptMarketMessage(String message) – This method will display the Market Message in the market display:
    - Call the user display manager's updateMarketState method, passing String message passed into this method.
    - *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

6. public void acceptTicker(String product, Price price, char direction) – This method will display the Ticker data in the market display:
    - Call the user display manager's updateTicker method, passing the same 3 parameters passed into this method.
    - *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

7. public void acceptCurrentMarket(String product, Price bPrice, int bVolume, Price sPrice, int sVolume) – This method will display the Current Market data in the market display:
    - Call the user display manager's updateMarketData method, passing the same 5 parameters passed into this method.
    - *If any method call throws an exception, you should catch it, print the error message to the screen, and then continue. Do not propagate the exception.*

8. void connect() – This method will connect the user to the trading system. To do this:
    - Call the user command service's "connect" method passing this user (i.e., "this") as the parameter, and user the "long" connection id data member to save the return value from that call.
    - Set the 'list of the stocks" data member to the results of a call to the user command service's "getProducts" method, passing this user's user name and connection id as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

9. void disConnect() – This method will disconnect the user to the trading system. To do this:
    - Call the user command service's "disConnect" method passing this user's user name and connection id as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

10. void showMarketDisplay() - This method qwill activate the market display. For now this is a text based user interface, but in future it will be a full GUI. This should do the following:
    - If the "list of stocks" data member is null, then this user has not connected so the display should not be activated. Throw (and propagate) an exception (i.e., UserNotConnectedException).
    - If the user display manager data member is null then this is the first time we've opened a display – set the user display manager data member to a new UserDisplayManager object.
    - Then call the user display manager's "showMarketDisplay()" method.

11. String submitOrder(String product, Price price, int volume, BookSide side) - This method forwards the new order request to the user command service and saves the resulting order id. This is done as follows:
    - Call the user command service's "submitOrder" method passing the same 4 parameters passed to this method as the parameters to that method. Save the String id that is returned from that call.
    - Create a new TradableUserData object using the username, product, side and the id you just saved as the parameters to its constructor.
    - Add that TradableUserData object to the list of TradableUserData objects data member.
    - *Propagate any exceptions these calls might generate.*

12. void submitOrderCancel(String product, BookSide side, String orderId) - This method forwards the order cancel request to the user command service as follows:
    - Call the user command service's "submitOrderCancel" method passing this user's user name, connection id, and the product, side, and order id (passed into this method) as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

13. void submitQuote(String product, Price bPrice, int bVolume, Price sPrice, int sVolume) - This method forwards the new quote request to the user command service as follows:
    - Call the user command service's "submitQuote" method passing this user's user name, connection id, and the 5 parameters passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

14. void submitQuoteCancel(String product) - This method forwards the quote cancel request to the user command service as follows:
    - Call the user command service's "submitQuoteCancel" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

15. void subscribeCurrentMarket(String product) - This method forwards the current market subscription to the user command service as follows:
    - Call the user command service's "subscribeCurrentMarket" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

16. void subscribeLastSale(String product) - This method forwards the last sale subscription to the user command service as follows:
    - Call the user command service's "subscribeLastSale" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

17. void subscribeMessages(String product) - This method forwards the message subscription to the user command service as follows:
    - Call the user command service's "subscribeMessages" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

18. void subscribeTicker(String product) - This method forwards the ticker subscription to the user command service as follows:
    - Call the user command service's "subscribeTicker" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method.
    - *Propagate any exceptions these calls might generate.*

19. Price getAllStockValue() - Returns the value of the all Sock the User owns (has bought but not sold). To do this, simply return the results of a call to this user's Position object's "getAllStockValue()" method.

20. Price getAccountCosts() – Returns the difference between cost of all stock purchases and stock sales. To do this, simply return the results of a call to this user's Position object's "getAccountCosts ()" method.

21. Price getNetAccountValue() – Returns the difference between current value of all stocks owned and the account costs. To do this, simply return the results of a call to this user's Position object's "getNetAccountValue ()" method.

22. String[][] getBookDepth(String product) - Allows the User object to submit a Book Depth request for the specified stock. To do this, return the results of a call to the user command service's "getBookDepth" method passing this user's user name, connection id, and the String product passed into this method as the parameters to that method. *Propagate any exceptions these calls might generate.*

23. String getMarketState() - Allows the User object to query the market state (OPEN, PREOPEN, CLOSED). To do this, return the results of a call to the user command service's "getMarketState" method passing this user's user name, connection id as the parameters to that method. *Propagate any exceptions these calls might generate.*

24. ArrayList< TradableUserData> getOrderIds() - Returns a list of order id's (a data member) for the orders this user has submitted.

25. ArrayList<String> getProductList() - Returns a list of stocks (a data member) available in the trading system.

26. Price getStockPositionValue(String product) – Returns the value of the specified stock that this user owns. To do this, simply return the results of a call to this user's Position object's "getStockPositionValue (String product)" method.

27. int getStockPositionVolume(String product) – Returns the value of the specified stock that this user owns. To do this, simply return the results of a call to this user's Position object's "getStockPositionVolume (String product)" method.

28. ArrayList<String> getHoldings() - Returns a list of all the Stocks the user owns. To do this, simply return the results of a call to this user's Position object's "getHoldings ()" method.

29. ArrayList<TradableDTO> getOrdersWithRemainingQty(String product) - Gets a list of DTO's containing information on all Orders for this user for the specified product with remaining volume. To do this, return the results of a call to the user command service's "getOrdersWithRemainingQty" method, passing the user name, connection id, and the String product symbol as parameters.

4. Position *class*

The Position class (for the "client" package) is used to hold an individual users profit and loss information, including how much they have spent buying stock, how much they gained or lost selling stock, and the value of the stock they currently own. To maintain this information this class needs the following (private) data members:

- A HashMap<String, Integer> to store the "holdings" of the user. The "key" is the stock, the "value" is the number of shares they own. Initially this will be empty.
- A Price object to hold the "account costs" for this user. This will keep a running balance between the "money out" for stock purchases, and the "money in" for stock sales. This should be initialized to a (value) price object with a value of "$0.00".
- A HashMap<String, Price> to store the "last sales" of the stocks this user owns. Last sales indicate the current value of the stocks they own. Initially this will be empty.

Constructor

- No parameters needed, nothing to initialize.

Methods

- public void updatePosition(String product, Price price, BookSide side, int volume) – This method will update the holdings list and the account costs when some market activity occurs. To do this:
    - Calculate the “adjusted volume” to be either the same as the volume passed in (if the “side” passed in is BUY), or the negative value of the volume passed in (if the “side” passed in is SELL).
    - If the holdings HashMap does not contain the product passed in as a key, then add an entry to the HashMap using the product as the key and the adjusted volume as the value
    - Else (HashMap does contain the product passed in as a key), get the current holding volume for the specified product from the HashMap, add the adjusted volume to that volume, and put the results back in the HashMap using the product as the key. *However, of the resulting volume is Zero, remove the entry from the holding HashMap for that product. Zero means the user no longer owns any of that stock.*
    - After that if/else – Multiply the price passed in by the volume passed in and save that total price.
    - If the “side” passed in is BUY, set the account costs data member to the results of a call to the account costs data member’s “subtract” method passing the total price from the current account costs value. (i.e., accountCosts = accountCosts.subtract(totalPrice);).
    - Else (it is SELL), set the account costs data member to the results of a call to the account costs data member’s “add” method passing the total price from the current account costs value. (i.e., accountCosts = accountCosts.add(totalPrice);).

- public void updateLastSale(String product, Price price) – This method should insert the last sale for the specified stock into the “last sales” HashMap (product parameter is the key, Price parameter is the value). It does not matter if there is already an entry for that product, the “put” into the HashMap will overwrite any existing entry.

- public int getStockPositionVolume(String product) – This method will return the volume of the specified stock this user owns. If the holdings HashMap does not have the product passed in as a key, then the user does not own this stock, so return zero (0). Otherwise, return the volume from the holdings Hashmap using the product passed in as the key.

- public ArrayList<String> getHoldings() – This method will return a sorted ArrayList of Strings containing the stock symbols this user owns.
    - To create an ArrayList from the keys in the holding HashMap, do the following:

      ```
      ArrayList<String> h = new ArrayList<>(holdings.keySet());
      ```
    - Then sort the ArrayList (using Collections.sort(…)) and return the sorted list.

- public Price getStockPositionValue(String product) – This method will return the current value of the stock symbol passed in that is owned by the user. To calculate this:
    - If the holdings HashMap does not contain an entry for the product passed in, return a (value) Price object of “$0.00”
    - Otherwise, the user does own the specified stock so get and save the last sale Price object from the “last sales” HashMap using the product passed in as the key. If that returns null (no current last sale for that stock) use a (value) Price object of “$0.00” instead. This is the last price for that stock.
    - Calculate the position value as the last sale price for that stock multiplied by the volume that oser owns (found in the holdings HashMap using the product passed in as the key). Return that calculated Price.

- public Price getAccountCosts() – This method simply returns the “account costs” data member.

- public Price getAllStockValue() – This method should return the total current value of all stocks this user owns. You should make use of the “getStockPositionValue” method – call it for each stock in the holding HashMap and sum up the total of all the position values. That is the value to be returned.

- public Price getNetAccountValue() - This method should return the total current value of all stocks this user owns PLUS the account costs.

5. UserCommandService *class*

The **UserCommand** class (for the "client" package) acts as a façade between a user and the trading system. This class should be a Singleton, as there is only one User Command Service that all users will work with. The user command service will need the following (private) data members to properly perform its tasks:

- A HashMap<String, Long> to hold user name and connection id pairs. Initially this "connected user ids" HashMap should be empty.
- A HashMap<String, User> to hold user name and user object pairs. Initially this"connected users" HashMap should be empty.
- A HashMap<String, Long> to hold user name and connection-time pairs (connection time is stored as a long). Initially this"connected time" HashMap should be empty.
-

Constructor

- Private constructor (Singleton) – no other functionality needed.

Methods

1. private void verifyUser(String userName, long connId) – This is a utility method that will be used by many of the methods in this class to verify the integrity of the user name and connection id passed in with many of the method calls found here. This method should do the following checks:
   - If the "connected user ids" HashMap does not contain the user name passed in as a key, then this user is not actually connected – throw an exception (i.e., "UserNotConnectedException").
   - If the long connId passed in does not equal the long value from the "connected user ids" HashMap (using the user name as the key), then the connection id passed in is invalie – throw an exception (i.e., "InvalidConnectionIdException").

2. public synchronized long connect(User user) – This method will connect the user to the trading system. This is done as follows:
   - If the "connected user ids" HashMap already contains the user's username (passed in) as a key, then this user already connected – throw an exception (i.e., "AlreadyConnectedException").
   - Add (put) an entry in the "connected user ids" HashMap using the user's username (passed in) as the key and the current time (in nanoseconds - System.nanoTime()) as the value.
   - Add (put) an entry in the "connected users" HashMap using the user's username (passed in) as the key and the User object passed in as value.
   - Add (put) an entry in the "connected time" using the user name as the key and the current time (in milliseconds - System.currentTimeMillis()) as the value.
   - Return the nanotime value you added to the "connected user ids" (in the second step) for this user's username

3. public synchronized void disConnect(String userName, long connId) – This method will disconnect the user from the trading system. This is done as follows:
   - Call "verifyUser" (in this class) passing the userName and connId parameters.
   - Remove the user's entry from the "connected user ids" HashMap (using the userName as the key to do the remove).
   - Remove the user's entry from the "connected users" HashMap (using the userName as the key to do the remove).
   - Remove the user's entry from the "connected time" HashMap (using the userName as the key to do the remove).

4. public String[][] getBookDepth(String userName, long connId, String product) – Forwards the call of "getBookDepth" to the ProductService.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Return the results of a call to the ProductService's "getBookDepth(product)" method.

5. public String getMarketState(String userName, long connId) – Forwards the call of "getMarketState" to the ProductService.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Return the results of a call to the ProductService's "getMarketState().toString()" method.

6. public synchronized ArrayList<TradableDTO> getOrdersWithRemainingQty(String userName, long connId, String product)– Forwards the call of "getOrdersWithRemainingQty" to the ProductService.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Return the results of a call to the ProductService's "getOrdersWithRemainingQty (userName, product)" method.

7. public ArrayList<String> getProducts(String userName, long connId) – This method should return a sorted list of the available stocks on this system, received from the ProductService.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Call the ProductService's "getProductList()" method and save the resulting ArrayList<String>.
    - Sort that ArrayList<String> using Collections.sort(…) and return the sorted list.

8. public String submitOrder(String userName, long connId, String product, Price price, int volume, GlobalConstants.BookSide side) - This method will create an order object using the data passed in, and will forward the order to the ProductService's "submitOrder" method.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Create a new Order object using the name, product, price, volume, and side data passd in.
    - Call the ProductService's "submitOrder" method passing that new order as the parameter, and save the returned String order id.
    - Return that order id.

9. public void submitOrderCancel(String userName, long connId, String product, BookSide side, String orderId) - This method will forward the provided information to the ProductService's "submitOrderCancel" method.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Call the ProductService's "submitOrderCancel" method passing the product, side, and order id (passed into this method).

10. public void submitQuote(String userName, long connId, String product, Price bPrice, int bVolume, Price sPrice, int sVolume) - This method will create a quote object using the data passed in, and will forward the quote to the ProductService's "submitQuote" method.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Create a new Quote object using the name, product, buy price, buy volume, sell price and sell volume data passed in.
    - Call the ProductService's "submitQuote" method passing that new quote as the parameter

11. public void submitQuoteCancel(String userName, long connId, String product) - This method will forward the provided data to the ProductService's "submitQuoteCancel" method.
    - Call "verifyUser" (in this class) passing the userName and connId parameters.
    - Call the ProductService's "submitQuoteCancel" method passing the userName and product (passed into this method).

12. public void subscribeCurrentMarket(String userName, long connId, String product) - This method will forward the subscription request to the CurrentMarketPublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the CurrentMarketPublisher’s “subscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

13. public void subscribeLastSale(String userName, long connId, String product) - This method will forward the subscription request to the LastSalePublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the LastSalePublisher’s “subscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

14. public void subscribeMessages(String userName, long conn, String product) - This method will forward the subscription request to the MessagePublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the MessagePublisher’s “subscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

15. public void subscribeTicker(String userName, long conn, String product) - This method will forward the subscription request to the TickerPublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the TickerPublisher’s “subscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

16. public void unSubscribeCurrentMarket(String userName, long conn, String product) - This method will forward the un-subscribe request to the CurrentMarketPublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the CurrentMarketPublisher’s “unSubscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

17. public void unSubscribeLastSale(String userName, long conn, String product) - This method will forward the un-subscribe request to the LastSalePublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the LastSalePublisher’s “unSubscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

18. public void unSubscribeTicker(String userName, long conn, String product) - This method will forward the un-subscribe request to the TickerPublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the TickerPublisher’s “unSubscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

19. public void unSubscribeMessages(String userName, long conn, String product) - This method will forward the un-subscribe request to the MessagePublisher.
    - Call “verifyUser” (in this class) passing the userName and connId parameters.
    - Call the MessagePublisher’s “unSubscribe” method passing the User object (from the “connected users” HashMap using the userName as the key) and product (passed into this method).

Provided Classes to Integrate into your Project ("gui" package)

- **gui** <package>
  - **UserDisplayManager** (class) – This class is a façade to the MarketDisplay. The User (UserImpl) will communicate with the UserDisplayManager, not the actual MarketDisplay. This way, the MarketDisplay can be changed or replaced without affecting the User (UserImpl).
  - **MarketDisplay** (class) – A stand in for a GUI; displays text-based feedback showing information received from the trading system

These classes will be provided to you on the COL site (Documents section) in a ZIP file that contains the "gui" folder that holds the UserDisplayManager and MarketDisplay classes. You need to put that "gui" folder in your project folder, in the same place (at the same level) as the other packages (price, tradable, client, etc.).

You might need to make slight changes of your class names or exceptions vary from what these classes were designed with. As in other phases, this kind of alteration is ok to do.

Testing Phase 5

A test "driver" class with a "main" method will be provided (*soon*) that will exercise the functionality of your Phase 5 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 and related Classes**
- **User Command Service**

Phase 6 (1 Week) 11/5 – 11/12:

- User Interface GUI
- Simulated Traders

[Final Exam 11/19]