Assignment 3

Object-Oriented Software Development

Adding Users, Book Management, Flyweight (200 pts)

Objectives

Create well-written object-oriented classes that represent trading Users, a UserManager (a Façade to the Users), and a ProductManager (a Façade to the ProductBooks). We will also update our Price/PriceFactory to implement the Flyweight design pattern. A "Main" class with a "main" method will be provided that uses your classes and generates a variety of trading scenarios.

Assignment 3 Changes

Change your PriceFactory so that it implements the Flyweight Design Pattern – only one Price object of a specific value should be created. You will create Price objects for many price values, but only one Price object should exist for any one price value.

Assignment 3 Classes

1) **User** class

The User class represents a trader user. Trader users will enter tradables to trading in our system. The user will also maintain a collection of tradables submitted, as described below:

- The User class should have a public constructor that accepts and sets the 3-character user id (described below).
- A User must maintain certain data values – shown below (remember to make these private to enforce information hiding):
  - String userId: A 3-letter user code – must be 3 letters, no spaces, no numbers, no special characters (i.e., XRF, BBT, KNT, etc.). This should be passed into the constructor. Cannot be changed once set.
  - HashMap<String, TradableDTO> tradables: The user class should maintain this HashMap of tradables (TradableDTOs) they have submitted. The key is the tradable id, the values is a TradableDTO. This should be set to a new HashMap when it is declared.
- A User must be able to perform certain behaviors – shown below. They should be public unless otherwise indicated:
  - Create a *private* modifier for the userId that is called by the User constructor. The modifier should ensure the incoming value meets the field requirements listed above.
  - public void updateTradable (TradableDTO o) ➔ Add/replace the incoming Tradable DTO to the user's tradable HashMap. If the TradableDTO passed in is null, do nothing.
  - Override the toString method to generate a String like this:
    - User id, then the user's tradables.

      Example

```
User Id: CAT

   Product: GOOG, Price: $52.45, OriginalVolume: 220, RemainingVolume: 0, CancelledVolume: 0, FilledVolume: 220, User: CAT, Side: SELL, Id: CATGOOG$52.4557855274865100

   Product: GOOG, Price: $52.40, OriginalVolume: 270, RemainingVolume: 270, CancelledVolume: 0, FilledVolume: 0, User: CAT, Side: SELL, Id: CATGOOG$52.4057855298770300

   Product: GOOG, Price: $52.85, OriginalVolume: 70, RemainingVolume: 40, CancelledVolume: 0, FilledVolume: 30, User: CAT, Side: SELL, Id: CATGOOG$52.8557855267845000
```

2) **UserManager** class

The UserManager class maintains a collection of all users in the system – it acts as a Façade to the Users. Since we only want to have one UserManager object in our application, this class should be a *Singleton* (design pattern).

- The UserManager maintains a collection (i.e., TreeMap) of all users. The key is the userId, the value is the User object. We use a TreeMap here so when we iterate through the keys, they are in sorted order.
- The UserManager must be able to perform certain behaviors – shown below. They should be public unless otherwise indicated:
  - void init(String[] usersIn) ➔ Create a new User object for each userId in the String array passed in. Each User object should be added to the UserManager's TreeMap of users. If the String passed in is null, throw a DataValidationException.
  - void updateTradable(String userId, TradableDTO o) ➔ This method should add/replace the TradableDTO for the specified User by calling the User's "updateTradable" method. If the userId is null, throw a DataValidationException. If the TradableDTO is null, throw a DataValidationException. If the user does not exist, throw a DataValidationException.
  - Override the toString method to generate a String like this:
    - The result of calling each User's toString().

      Example:

```
User Id: ANN

       Product: GOOG, Price: $52.95, OriginalVolume: 270, RemainingVolume: 270, CancelledVolume: 0,
FilledVolume: 0, User: ANN, Side: SELL, Id: ANNGOOG$52.9558518804111600

       Product: WMT, Price: $70.40, OriginalVolume: 305, RemainingVolume: 155, CancelledVolume: 0,
FilledVolume: 150, User: ANN, Side: BUY, Id: ANNWMT$70.4058518805034300

       …

User Id: BOB

       Product: TGT, Price: $88.75, OriginalVolume: 300, RemainingVolume: 300, CancelledVolume: 0,
FilledVolume: 0, User: BOB, Side: BUY, Id: BOBTGT$88.7558518798475100

       Product: WMT, Price: $70.59, OriginalVolume: 210, RemainingVolume: 210, CancelledVolume: 0,
FilledVolume: 0, User: BOB, Side: SELL, Id: BOBWMT$70.5958518807798900

       …
```

3) **ProductManager** class

The ProductManager class maintains a collection of ProductBook objects for all stocks used in the system – it acts as a Façade to the ProductBooks. Since we only want to have one ProductManager object in our application, this class should be a Singleton (design pattern).

- The ProductManager will need the following data elements to properly represent a product book in the application:
  - A collection of all ProductBooks in the application. This is best represented as a HashMap (the key is the String product symbol; the value is a ProductBook object).
- The ProductManager will need the following methods to properly manage the ProductBooks in the application:

- void addProduct(String symbol): This method should create a new ProductBook object for the stock symbol passed in, and add it to the HashMap of all ProductBook objects. If the symbol is null or it does not match symbol requirements (back in Part 2), throw a DataValidationException.
- ProductBook getProductBook(String symbol): Return the ProductBook using the String symbol passed in. If the product does not exist, throw a DataValidationException.
- String getRandomProduct(): Return a randomly selected product symbol from the collection of all ProductBook objects. If no products exist, throw a DataValidationException.
- TradableDTO addTradable(Tradable o): Add the Tradable to the ProductBook (using the String product symbol from the Tradable object to determine which ProductBook it goes to). Then call the UserManager's updateTradable method, passing it the Tradable's use id and a new TradableDTO created using the Tradable passed in. Return the TradableDTO you receive back from the ProductBook. If the Tradable passed in is null, throw a DataValidationException.
- TradableDTO[] addQuote(Quote q): Get the ProductBook (using the symbol in the Quote object) and call removeQuotesForUser(passing the String user from the Quote object). Next, call addTradable passing the BUY ProductBookSide from the Quote passed in (save the TradableDTO returned). Then call addTradable passing the SELL ProductBookSide from the Quote passed in (save the TradableDTO returned). Return the BUY and SELL TradableDTO's in a 2-element array of TradableDTOs. If the quote passed in is null, throw a DataValidationException.
- TradableDTO cancel(TradableDTO o): Using the String product symbol from the TradableDTO passed in, find the ProductBook. Call that ProductBook's "cancel" method passing it the side and tradable id from the TradableDTO passed in. If successful, return the TradableDTO returned from the ProductBook's "cancel" method. If the cancel attempt fails, print a message indicating the failure to cancel, and return a null. If the TradableDRO passed in is null, throw a DataValidationException.
- TradableDTO[] cancelQuote(String symbol, String user): Using the String symbol, get the ProductBook using the String symbol passed in, and call its removeQuotesForUser passing it the String user. Return the TradableDTO array that comes back from removeQuotesForUser. If the symbol passed in is null, throw a DataValidationException. If the user if passed in is null, throw a DataValidationException. If the product does not exist for the specified symbol, throw a DataValidationException.
- public String toString(): Override the toString method to generate a String containing a summary of all ProductBooks as follows (be sure to let the ProductBooks generate their part of the String).

  Example

```
Product Book: TGT
   Side: BUY
        $133.00:
           XEN BUY side quote for TGT: $133.00, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: XENTGT$133.00518653408166800
           YAM BUY side quote for TGT: $133.00, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: YAMTGT$133.00518653408367400
           ZEN BUY order: TGT at $133.00, Orig Vol: 15, Rem Vol: 15, Fill Vol: 0, CXL Vol: 0,
   ID: ZENTGT$133.00518653408560700

   Side: SELL
        $133.20:
           XEN SELL side quote for TGT: $133.20, Orig Vol: 75, Rem Vol: 75, Fill Vol: 0, CXL
   Vol: 0, ID: XENTGT$133.20518653408198200
           YAM SELL side quote for TGT: $133.20, Orig Vol: 100, Rem Vol: 100, Fill Vol: 0, CXL
   Vol: 0, ID: YAMTGT$133.20518653408395500

Product Book: WMT
   Side: BUY
        $134.00:
           AXE BUY side quote for WMT: $134.00, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: AXEWMT$134.00518653398695600
        $133.90:
           BAT BUY side quote for WMT: $133.90, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: BATWMT$133.90518653400537100

   Side: SELL
        $134.20:
           BAT SELL side quote for WMT: $134.20, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: BATWMT$134.20518653400568300
        $134.30:
           AXE SELL side quote for WMT: $134.30, Orig Vol: 50, Rem Vol: 50, Fill Vol: 0, CXL
   Vol: 0, ID: AXEWMT$134.30518653398953500
```

4) Changes to: ProductBookSide:

Whenever a Tradable’s state changes (i.e., when its data changes – fill, cancel, remaining volumes, etc) we need to send an update of that Tradable’s state back to the user. This will make use of the methods previously described in this document. This should be done as follows:

- In ProductBookSide's tradeOut method, you need to update the user's tradables at the end (i.e., the last line) of both the IF and the ELSE blocks of that method by calling the UserManager’s updateTradable method, passing the tradable’s user id and a the current tradable’s TradableDTO as parameters. Example:
  - `UserManager.getInstance().updateTradable(<user id>, <TradableDTO>);`

- In the ProductBookSide’s cancel method, add a call to update the user's tradable at the end of the IF block (just before you return the TradableDTO). Example:
  - `UserManager.getInstance().updateTradable (<user id>, <TradableDTO>);`

- In the ProductBookSide’s add method, add a call to update the user's tradable at the end of the method (just before you return the TradableDTO). Example:
  - `UserManager.getInstance().updateTradable (<user id>, <TradableDTO>);`

- In the ProductBookSide's removeQuotesForUser, you need to update the user's tradables at the end of that method, just before you return the DTO. Example:
  - `UserManager.getInstance().updateTradable (<user id>, <TradableDTO>);`

5) Testing

A “Main” class is provided with this assignment (on D2L) that should be added to your project. The “main” will perform actions that will test your classes. The expected output is also provided on D2L.

6) Project Assistance

If you are stuck on some design or code related problem *that you have exhaustively researched and/or debugged yourself*, you can email me a ZIP file of your entire project so that I can examine the problem.

All emailed assistance requests must include a detailed description of the problem, and the details of what steps you have already taken in trying to determine the source of the problem.

7) Submissions & Grading

When submitting, you should submit a ZIP file of your entire project so that I can compile and execute it on my end. To reduce the size of the zipped file, please remove the “out” folder from your project (if present) before zipping. (This folder is automatically regenerated each time you compile/run so It’s safe to delete)

The following are the key points that will be examined in Project when graded:

- Good object-oriented design & implementation
- Proper use of java language conventions
- Proper object-oriented coding practices
- Correct, accurate application execution

Submissions must reflect the concepts and practices we cover in class, and the requirements specified in this document.

Please review the Academic Integrity and Plagiarism section of the syllabus before, during, and after working on this assignment. All work must be your own.

Late submission up to 4 days late will be accepted, though late submissions will incur a 20% penalty. *No late submissions will be accepted for grading after that 4 day time period has passed.*

If you do not understand anything in this handout, please ask!