CS 319 - 2022/2023 Spring
Design Patterns Homework
10.05.2023

Due: 23:55, 17.05.2023

Version 4, Last Updated: 16.05.2023 21.15

Important Notes:

- In this lab, you are expected to use design patterns that you learned in class and in the tutorial.
- You will submit the following:
    - Source code which is written in Java.
        - For each different class, you should create a new java file.
        - There should be a main file where you test your implementation in detail. You must test each **part** of the question and separate them via comments.
    - A report in pdf named report which will contain:
        - **Class Diagram** of your implementation.
        - Another **diagram** of cases and case transitions of your *StockMarket* implementation. Also, don't forget to clearly indicate what type the diagram is.
        - Explanation of design patterns that you used. You should explicitly say **on which object(s), how (e.g. which properties did you change)**, and why you used that design pattern.
- While submitting your work, put your source code and report in a folder named "**<Student_ID>_<Surname>_<Name>_DesignPattern**". Finally, zip that file and upload it to Moodle. (Example: 22003850_Ozgun_Tolga_DesignPattern.zip)
- The students who do not follow the submission guidelines will lose some points!
- You can refer to any kind of resource while preparing your solution. However your submitted content must be original, i.e any detection of AI generated or similar content will be investigated under plagiarism. Please check out the following link for more information: http://www.cs.bilkent.edu.tr/~david/plagiarism
- **NO LATE SUBMISSIONS ALLOWED!**
- If you have any questions, feel free to ask in the Slack channel or send a mail to tolga.ozgun@ug.bilkent.edu.tr

The problem starts on the next page.

You have recently graduated from Bilkent University. An investment company in Wall Street has hired you as a Software Engineer. This company is a newly-founded company, and they want to build their infrastructure from scratch.

You are tasked with creating a stock market simulation software so that the company can simulate their actions. The company wants to expand this software in the future, and is willing to give you enough time to make it right. In other words, they want you to write good code and use proper software development methods.

In summary, the simulation software should have the following features:

- The software should contain various users, stocks and stock markets.
- The users should be able to operate on stocks, and get updates from the stock markets and stocks.

PART 1: Class Implementations

As your first step, your team lead has tasked you to implement a *Stock* class. This object will have following properties: *name* and *symbol* as type String, where name represents the stock name and the symbol represents the abbreviation, a short name for the stock (i.e. Alphabet Inc. has GOOGL as its symbol). *Stock* objects will also have *price*, *percentChange, volume* and *marketCap* properties as type double. You can implement however many properties and functions you want, as long as they are needed. However, besides the constructor, a *Stock* object must have a custom implementation of *toString()*. When *toString()* is called, it should return an informative message about the *Stock* object. The format of this message is important and should be as following:

Example 1:

For a *Stock* object with

- name = “Apple”
- symbol = “APPL”
- price = 100.0
- percentChange = -3.28
- volume = 1001513.0
- marketCap = 1024150.0

Output of *toString()* will be as following:

```
--- Stock Information ---
Name: Apple (AAPL)
Price: 100.0 TRY
Percent Change: -3.28
Volume: 1001513.0
Market Cap: 1024150.0
```

After turning your work in, your team lead also asked you to implement a *StockMarket* class. This object must have *name* and *symbol* properties as type String. It should also have a *stocks* property as type LinkedList<Stock>. Similar to a *Stock* object, a *StockMarket* object should have a custom *toString()* function, where it should return information about the current object. Besides these properties and methods, you can implement any properties or methods of your own as long as they are needed.

Example 2

For a *StockMarket* object with:

- name = "New York Stock Exchange"
- symbol = "NYSE"
- stocks = appleStock, microsoftStock

Where appleStock is a *Stock* object with

- name = "Apple"
- symbol = "APPL"
- price = 100.0,
- percentChange = 1.82,
- volume = 1023113.0
- marketCap = 100350.0

And microsoftStock is a *Stock* object with

- name = "Microsoft"
- symbol = "MSFT"
- price = 100.0,
- percentChange = -3.28,
- volume = 19121513.0
- marketCap = 1131250.0

Output of *toString()* in the current *StockMarket* object will be:

```
--- Stock Market Information ---
Name: New York Stock Exchange (NYSE)
Stocks:

--- Stock Information ---
Name: Apple (AAPL)
Price: 100.0 TRY
Percent Change: 1.82
Volume: 1023113.0
Market Cap: 100350.0

--- Stock Information ---
Name: Microsoft (MSFT)
Price: 100.0 TRY
Percent Change: -3.28
Volume: 19121513.0
Market Cap: 1131250.0
```

Example 3

For a StockMarket object with

- name = “NASDAQ”
- symbol = “NASDAQ”
- and an empty stocks list

Output of *toString()* in the current *StockMarket* object will be:

```
--- STOCK MARKET ---
Name: New York Stock Exchange (NYSE)
Stocks:
No stocks available
```

Now, your team lead wants you to implement the User class and its subclasses. Users can be classified into different types based on their investment strategies. You should implement the following subclasses:

- AggressiveInvestor
- BalancedInvestor
- ConservativeInvestor

All User subclasses should have the following properties: *stockMarket* as StockMarket, *name* as String, *investment_budget* as double and *investment_portfolio* as Map<String, Integer> where *String* contains *Stock*'s name and Integer has the quantity. The constructor method should take *stockMarket* as StockMarket, *name* as String and *investment_budget* as double. You can implement however many properties and functions you want, as long as they are needed.

All of the *User* object subclasses should implement a *shouldBuyStock(Stock)* method where the argument is a *Stock*. This function should return an Integer, indicating how many Stock should the *User* purchase. If they should not buy any stock, the method should return 0. It should not return any negative values. Each subclass has its own implementation.

An *AggressiveInvestor* should only buy a *Stock* if the *Stock* has a negative change. If so, it should buy as many of the current *Stock* as possible without exceeding 10% of their investment budget. A *BalancedInvestor* should only buy a *Stock* if the *Stock* has a negative change, and if their budget is more than 20 times the *Stock* price. If so, they should buy as many of the current *Stock* as possible without exceeding 8% of their investment budget. A *ConservativeInvestor* should only buy a *Stock* if the *Stock* has a negative change and their budget is more than 25 times the *Stock* price. If so, they should buy as many of the current *Stock* as possible without exceeding 5% of their investment budget.

Example 4

An *AggressiveInvestor* object with

- name = "Beff Jezos"
- investment_budget = 990023.21

and a *Stock* object with

- name = "Microsoft"
- symbol = "MSFT"

- price = 123.4,
- percentChange = -3.28,
- volume = 1001513.0
- market cap = 1024150.0

should output when *shouldBuyStock(Stock)* is called:

```
Beff Jezos calculated that they should buy 802 shares of
Microsoft (MSFT).
```

User classes should also implement a *shouldSellStock(Stock)* method similar to the previous *shouldBuyStock(Stock)* method. This function takes a *Stock* object and returns an integer value representing the number of *Stock* that should be sold. If no *Stock* should be sold, the function should return 0. It should not return any negative values.

An *AggressiveInvestor* should sell a *Stock* if the *Stock* has a positive change. If so, it should sell as many of the current *Stock* as possible without exceeding 12% of their current investment in that *Stock*. Also, if a Stock has a negative change below -2.00%, an *AggressiveInvestor* should instantly sell **all of** that stock. A *BalancedInvestor* should sell a *Stock* if the *Stock* has a positive change, and if their budget is less than 1000 times the *Stock* price. If so, they should sell as many of the current *Stock* as possible without exceeding 8% of their investment budget. A *ConservativeInvestor* should sell a *Stock* if the *Stock* has a positive change and their budget is more than 75 times the *Stock* price. If so, they should sell as many of the current *Stock* as possible.
UPDATE: There has been a calculation mistake. Although it has been said that
If so, it should sell as many of the current Stock as possible without exceeding 12% of their current investment in that Stock.
The calculation in Example 5 was done considering 12% of total budget. Therefore answers of either 12% of total budget or 12% of current investment will be accepted. For simplicity, current investment can be calculated by current stock price * quantity.

Example 5
An *AggressiveInvestor* object with

- name = “Beff Jezos”
- investment_budget = 990023.21
- stocks = <”Microsoft”, 1731>

and a *Stock* object with

- name = “Microsoft”
- symbol = “MSFT”
- price = 123.4,
- percentChange = 3.28,
- volume = 1001513.0
- market cap = 1024150.0

should output when *shouldSellStock(Stock)* is called:

```
Beff Jezos calculated that they should sell 962 shares of
Microsoft (MSFT).
```

PART 2: Extend Functionalities

Now that you have implemented the *Stock* and *StockMarket* classes, your team lead has asked you to extend a functionality. You will need to create *open()*, *close()*, *buyStock(Stock, int, User)*, and *sellStock(Stock, int, User)* methods inside the *StockMarket* class. *open()* and *close()* methods will return a boolean to indicate if the corresponding action was completed, i.e once *StockMarket* is closed after calling *close()*. However, *buyStock()* and *sellStock()* will return a number.

HINT: You can use Java Interfaces.

- *open()*: This method should be called when the stock market is opened. Should return true if the stock market was closed before, and opened after calling this method.
- *close():* This method should be called when the stock market is closed. Should return true only if the stock market was open before, and closed after calling this method.
- *buyStock(Stock, int, User)*: This method should be called when a user wants to buy a stock. It takes a *Stock* object, an integer and a *User* object as input parameters. The integer indicates the amount of Stock to be bought. Should return the total cost of transaction after fees, if successful. Should return a negative value, if unsuccessful.
- *sellStock(Stock, int, User)*: This method should be called when a user wants to sell a stock. It takes a *Stock* object and a *User* object as input parameters. The integer indicates the amount of Stock to be bought. Should return true if the sell action is successful. Should return the total value of transaction after fees, if successful. Should return a negative value, if unsuccessful.

These methods should have different implementations for different cases in the *StockMarket*. We will have four different situations in *StockMarket*.

Firstly, *StockMarket* can be open, and during this time users are allowed to make transactions, whether it is buying or selling *Stock*s. In this state, the fee per transaction is 1.5% of the total transaction amount.
When the StockMarket is closed, no transactions can be made. In other words, no *Stocks* can be sold or bought.

In addition, the *StockMarket* can be in a *'High Volatile'* situation. StockMarket can be *High Volatile* once the daily cost of transactions exceeds 1,000,000 TRY. In this case, the fee per transaction is 3% of the total transaction amount.

Lastly, the StockMarket can be in a *'Low Volatile'* situation. StockMarket can be *High Volatile* once the daily number of transactions are higher than 10 but the total cost of transactions is less than 500,000 TRY. In this state, the fee per transaction is 0.5% of the total transaction amount.

Fee is calculated after calculating the *Stock* cost. If the User does not have enough money to complete the transaction after fees, the function should not make the transaction and should return a negative value.

HINT: The day consists of transactions between the *StockMarket* being opened and closed. In other words, each cycle from *StockMarket*'s opening and closing constitutes a day.

You should make sure that *StockMarket* can switch the functionality of the *open()*, *close()*, *buyStock(Stock, int, User)*, and *sellStock(Stock, int, User)* methods correctly according to its current situation.

Example 6
Once the *buyStock(Stock, int, User)* is called while the *StockMarket* is open on the *Stock* object with

- name = "Microsoft"
- symbol = "MSFT"
- price = 123.4,
- percentChange = -3.28,
- volume = 1001513.0
- market cap = 1024150.0

Also with the integer (amount) given to *buyStock* function equals to 802, and the *User* object with

- name = “Beff Jezos”
- investment_budget = 990023.21

It should output the following:

```
Beff Jezos has made a transaction to buy 802 shares of
Microsoft (MSFT) while the Stock Market is open.
Original cost: 98966.8 TRY
Fee: 1.5%
Total cost: 100451.302 TRY
Beff Jezos has 889571.908 TRY left.
```

If the *StockMarket* is low volatile or high volatile, it should display that instead of declaring it open.

Example 7

Once the *buyStock(Stock, int, User)* is called while the *StockMarket* is closed on the *Stock* object with

- name = “Microsoft”
- symbol = “MSFT”
- price = 123.4,
- percentChange = -3.28,
- volume = 1001513.0
- market cap = 1024150.0

Also with the integer (amount) equals to 962, and the *User* object with

- name = “Beff Jezos”
- investment_budget = 990023.21

It should output the following:

```
Beff Jezos has made a transaction to buy 802 shares of
Microsoft (MSFT) while the Stock Market is closed.
Unable to process the transaction.
```

The outputs provided in *buyStock(Stock, int, User)* should also carry to *sellStock(Stock, int, User)*, but instead of outputting

`'has made a transaction to buy'`,

it should output `has made a transaction to sell'`

After that, you should make sure that *User* objects are notified of the changes in the *StockMarket* situation. Extend the User functionality to include the following:

An *AggressiveInvestor* should be allowed to make up to 10 transactions per day,
A *BalancedInvestor* should be allowed to make up to 7 transactions per day, and
A *ConservativeInvestor* should be allowed to make up to 5 transactions per day.

Example 8

If an *AggressiveInvestor* with

- name = "Barren Wuffet"

tries to make the 11th transaction of the day your implementation should output:

```
Barren Wuffet tried to make their 11th transaction of the day,
however as an AggressiveInvestor they are only allowed up to
10.
```

Moreover, the *Users* should be notified of change in all Stocks in their *StockMarket*. And after they are notified, they should output a message and they should call previously implemented *shouldBuyStock(Stock)* and *shouldSellStock(Stock)* methods, and buy or sell stocks based on their response.

Example 9

If a *User* with

- name = "Eray Tüzün"

is alerted of a change in the *Stock* with properties:

- name = "Microsoft"
- symbol = "MSFT"

```
Eray Tüzün is alerted of change in Microsoft(MSFT).
```

PART 3: Final Touch

Now that everything is done, you are expected to integrate the functionality of updating the *Stock* objects. You should implement an update() method which will take *price*, *percentChange*, *volume* and *marketCap* as type double. This should update the *price* and *percentChange*, *volume* and *marketCap* properties of the *Stock* object.

Example 9

A *Stock* object with

- name = “Microsoft”
- symbol = “MSFT”
- price = 123.4,
- percentChange = -3.28,
- volume = 9742423.0
- marketCap = 24824213.0

is used to call the update(89.12, -3.12, 3774231.0, 32834141.0) method. The output is as following:

```
Stock Microsoft (MSFT) is updated to price: 89.12 TRY, percent
change: -3.12, volume: 3774231.0 and market cap: 32834141.0
```

If a function definition is not given to have a console output, you can declare your own console output. This applies to the functions of your own declaration. However, try to limit this to minimum, and only to the functions with a core use. For example, adding a *Stock* to a *StockMarket*.

Example 10

An example Tester program:

```
public class Tester {
   public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket("New York Stock Exchange", "NYSE");
        Stock stock = new Stock("Apple", "AAPL", 100.00, 3.2, 1000000, 10000);
        stockMarket.addStock(stock);
        User balancedInvestor = new BalancedInvestor(stockMarket, "Cool Person",
1000000);
        stock.update(97.7, 2.3, 274242, 252343224);
   }
}
```

Example output:

```
Stock Market with name "New York Stock Exchange (NYSE)"
created.
Apple(APPL) stock with price 100.00 TRY, 3.2 percent change,
1000000.0 volume and 10000.0 market cap is added to New York
Stock Exchange (NYSE).
Cool Person is now trading on New York Stack Exchange (NYSE).
Stock Apple(APPL) is updated to price: 102.3 TRY, percent
change: 2.3, volume: 274242.0 and market cap: 252343224.0
Cool Person is alerted of change in Apple (APPL).
Cool Person calculated that they should buy 0 shares of Apple
(APPL).
Cool Person does not own any shares so cannot sell Apple
(APPL).
```

---

end of problem