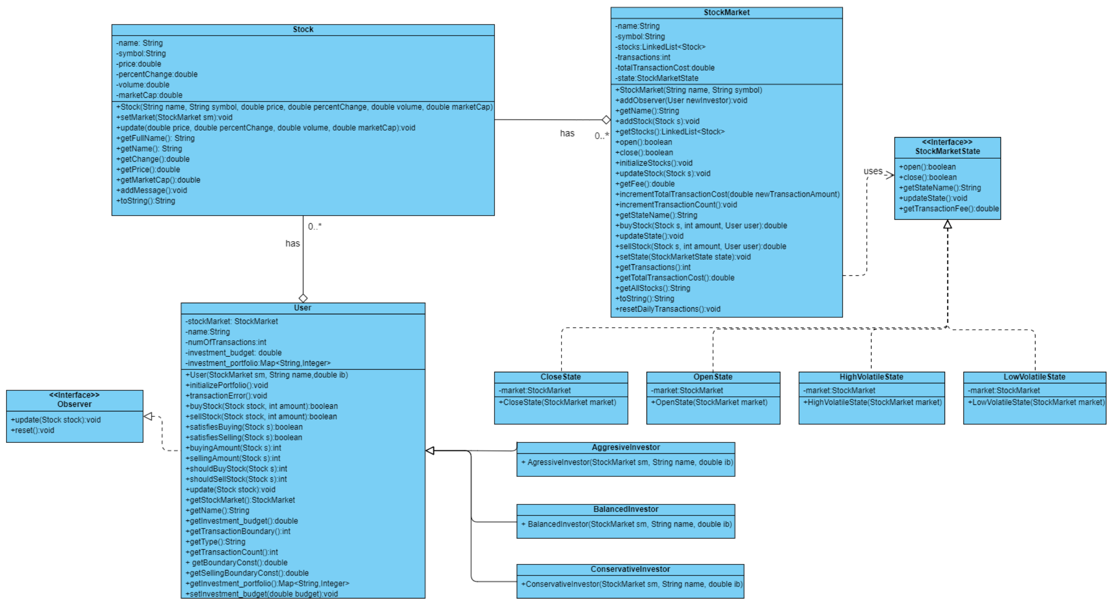
CS319 OBJECT-ORIENTED SOFTWARE PROGRAMMING

Design Pattern Homework

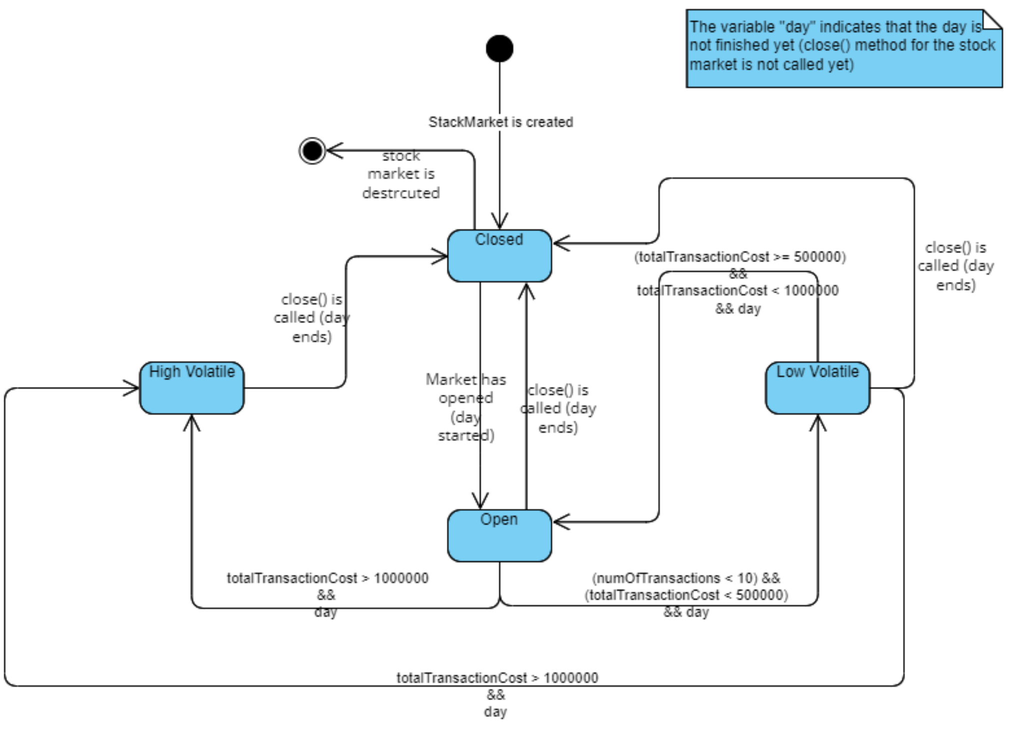
SPRING 2023

Name: İdil Atmaca
ID: 22002491
Instructor: Eray Tüzün
Teaching Assistant: Tolga Özgün
Date: 17/05/2023

CLASS DIAGRAM


Stock
-name: String
-symbol:String
-price:double
-percentChange:double
-volume:double
-marketCap:double
+Stock(String name, String symbol, double price, double percentChange, double volume, double marketCap)
+setMarket(StockMarket sm):void
+update(double price, double percentChange, double volume, double marketCap):void
+getFullName(): String
+getName(): String
+getChange():double
+getPrice():double
+getMarketCap():double
+addMessage():void
+toString():String
has
0..*
StockMarket
-name:String
-symbol:String
-stocks:LinkedList<Stock>
-transactions:int
-totalTransactionCost:double
-state:StockMarketState
+StockMarket(String name, String symbol)
+addObserver(User newInvestor):void
+getName():String
+addStock(Stock s):void
+getStocks():LinkedList<Stock>
+open():boolean
+close():boolean
+initializeStocks():void
+updateStock(Stock s):void
+getFee():double
+incrementTotalTransactionCost(double newTransactionAmount)
+incrementTransactionCount():void
+getStateName():String
+buyStock(Stock s, int amount, User user):double
+updateState():void
+sellStock(Stock s, int amount, User user):double
+setState(StockMarketState state):void
+getTransactions():int
+getTotalTransactionCost():double
+getAllStocks():String
+toString():String
+resetDailyTransactions():void
uses
<<Interface>>
StockMarketState
+open():boolean
+close():boolean
+getStateName():String
+updateState():void
+getTransactionFee():double
0..*
has
User
-stockMarket: StockMarket
-name:String
-numOfTransactions:int
-investment_budget: double
-investment_portfolio:Map<String,Integer>
+User(StockMarket sm, String name,double ib)
+initializePortfolio():void
+transactionError():void
+buyStock(Stock stock, int amount):boolean
+sellStock(Stock stock, int amount):boolean
+satisfiesBuying(Stock s):boolean
+satisfiesSelling(Stock s):boolean
+buyingAmount(Stock s):int
+sellingAmount(Stock s):int
+shouldBuyStock(Stock s):int
+shouldSellStock(Stock s):int
+update(Stock stock):void
+getStockMarket():StockMarket
+getName():String
+getInvestment_budget():double
+getTransactionBoundary():int
+getType():String
+getTransactionCount():int
+ getBoundaryConst():double
+getSellingBoundaryConst():double
+getInvestment_portfolio():Map<String,Integer>
+setInvestment_budget(double budget):void
<<Interface>>
Observer
+update(Stock stock):void
+reset():void
CloseState
-market:StockMarket
+CloseState(StockMarket market)
OpenState
-market:StockMarket
+OpenState(StockMarket market)
HighVolatileState
-market:StockMarket
+HighVolatileState(StockMarket market)
LowVolatileState
-market:StockMarket
+LowVolatileState(StockMarket market)
AggresiveInvestor
+ AgressiveInvestor(StockMarket sm, String name, double ib)
BalancedInvestor
+ BalancedInvestor(StockMarket sm, String name, double ib)
ConservativeInvestor
+ConservativeInvestor(StockMarket sm, String name, double ib)

STATE DIAGRAM


The variable "day" indicates that the day is not finished yet (close() method for the stock market is not called yet)
StackMarket is created
stock market is destrcuted
Closed
close() is called (day ends)
Market has opened (day started)
close() is called (day ends)
(totalTransactionCost >= 500000) && totalTransactionCost < 1000000 && day
close() is called (day ends)
High Volatile
Low Volatile
Open
totalTransactionCost > 1000000 && day
(numOfTransactions < 10) && (totalTransactionCost < 500000) && day
totalTransactionCost > 1000000 && day

Part 1
Template Design Pattern

There are three kinds of investors: Aggressive, Balanced, and Conservative. These types determine the number of transactions allowed for a day, constants used for buying and selling conditions, and actions. Since all the selling and buying amounts are calculated in the same way, and the only thing different in each user type is the constants, the template design pattern was used. In the user class, we are calculating the amounts of selling and buying stocks, but we are getting the constants from the individual class the user is in. However, we must implement one more step to check if the user meets the particular conditions. For example, the change rate should be negative to buy stock. However, for balanced users, their investment budget must be more than 20 times the total shares they buy. A similar case applies to Conservative users as well. Methods like satisfiesBuying() and satisfiesSelling() were used to check this condition. These methods are first introduced inside the User class and overridden, if necessary, inside any of the children's classes. Moreover, inside Conservative and Balanced users, we check if the operation could be done and return a boolean to indicate it. Furthermore, we are checking the particular condition that Aggressive investors have for selling stocks. If the change rate is less than -2, we are selling all our shares of that stock. Otherwise, we are implementing the template we used for other classes. Moreover, we are overriding the shouldSellStock() method inside the Aggressive Investor to achieve that.

Part 2
State Design Pattern

The stock market has four situations: closed, open, high-volatile, and low-volatile. Different conditions enable going from one specific situation to another. Moreover, each specific situation will affect the transaction fee. To achieve these, a state design pattern was implemented. An interface called StockMarketState was used, and each of these different states was created as concrete classes that implemented this interface. Using these classes, operations like open() and close() for StockMarket were implemented. Moreover, each state class includes a getTransactionFee() method to get the special transaction fee from each state. To connect the states and the stock market, an instance called stockMarketState was created inside the StockMarket. This attribute is initialized as "closed" inside the constructor. In other words, once the stock market is created, it goes to the closed state. Once the open() method is called, it goes to the open state. Afterward, according to the number of transactions and total transaction amount, it can either go to the low or high volatile state.

Moreover, we can go from low to high or open state when the conditions are met. Lastly, whenever the close() method is called, the stock market returns to the closed state.

Part 3
Observer Design Pattern

Whenever we have an update on stock, every investor that is in the stock market, which consists of the updated stock, must be notified. To achieve this, the observer design pattern was implemented. An Observer interface was created with the update(Stock stock) and reset() methods inside. This interface was later implemented inside the user class, and the update() and reset() methods were overridden. Whenever the day ends, the stock market closes, and we reset the number of daily transactions for that user. Furthermore, when the update() method is called, a message that says the user is alerted is printed. Moreover, shouldSellStock() and shouldBuyStock() methods were called inside it. If these methods return a positive integer value, buyStock() and sellStock() methods are called. Users will successfully buy or sell stocks if all conditions are met. The update() method is crucial in achieving the most important functionalities. So, we have to properly call the update() method inside the user class whenever there is an update in the current stock market. Consequently, a list of Observer objects was created inside the stock market class. Whenever a new investor is added to the stock market, we also add that user to our observer list. Whenever a specific stock is updated, inside the Stock class's update() method, the updateStockMarket() method is called. Inside the updateStockMarket() method, we are iterating over all the observers in the list and calling the update method for each.