King Mongkuts University of Technology Thonburi
Computer Engineering Department
CPE 343/372 Object Oriented Analysis and Design
Final Examination

20 May 2019 13:30-16:30

ANSWERS

Instructions

This examination consists of 11 pages including this cover page. ***Do all your work in these examination sheets, in the space provided***. You may use the back of the exam as scratch paper or to extend your answers if you need more space.

There are 12 questions for a total of 100 points. The number of points for each question is clearly indicated. Do not spend too much time on any one question. If you get stuck on some question, skip it and come back to it later after you have finished the others.

Read the instructions for each question very carefully. Some questions ask you to provide reasons for your answers ("why?" or "how do you know?") or to give an example. If you do not provide a reason or example as requested, you will not get any credit.

Write your answers clearly in the space provided, in English, using pencil or a blue or black pen.

You will have three hours for this examination.

You may use a hard copy (only) English-Thai/Thai-English dictionary for this examination. No other books or notes are permitted in the examination area. No electronic devices are permitted.

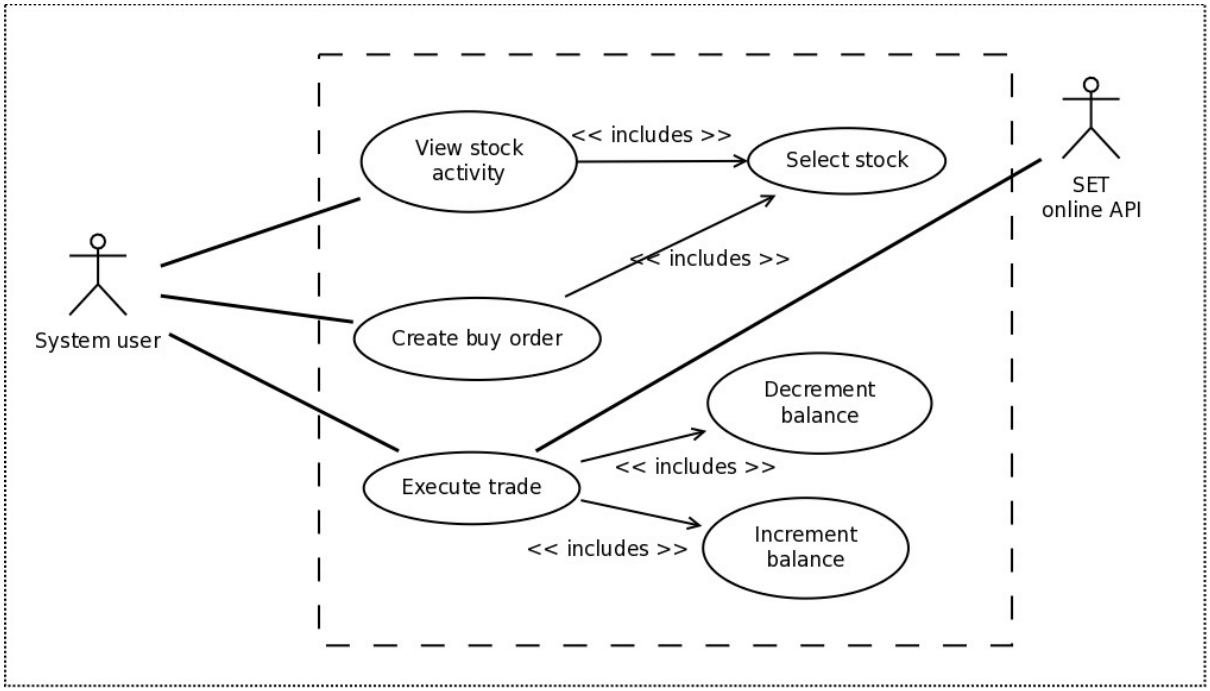
Instructor

Sally E. Goldin

(Dr. Sally E. Goldin)

Suppose we are designing an online stock trading system for a bank or brokerage company. This system will allow registered users to buy and sell stocks as well as to view stock activity and history. It will interact with the Stock Exchange of Thailand (SET) via the public online application programming interface (API) provided by SET.

Below is an initial Use Case Diagram for this system. Study the diagram, then answer the questions after it.



1. Do you think this Use Case Diagram is complete? Are there any use cases missing? If so, write the names of the missing use cases below. For each one, specify the use case goal and the actor(s) involved. If you think the diagram is complete, write “Complete”. (10 points)

In order of importance:

Create sell order – specify the details of a future instruction to sell stock – System user
Cancel order – remove a buy or sell order that has been specified but not yet triggered – System user
Login, Logout, Register - System user

There could be others.

To get full credit you needed to list at least two use cases, one of which was *Create buy order*.

2. This diagram includes four included use cases. Do you think these are correct? If so, explain why each one should be viewed as “included”. If not, which ones should be changed, and why? (10 points)

Increment balance and Decrement balance should be *extends*, not *includes*. Only one of the two will ever occur in a particular case, and if the order is canceled, neither will occur.

This was the main idea I was looking for. In a few cases I gave full credit based on someone making a very good argument for a different approach.

3. Below you will find a main success narrative for the **Create buy order** use case.

Use case name	Create buy order
Actor(s)	Registered system user
Goal	Specify a future transaction for buying some stock
Pre-conditions	User has logged in to the system
Main success narrative	1. User selects the stock of interest. 2. User specifies the number of shares to be bought. 3. User specifies the trigger price. When the stock price reaches this price, the order should be executed. 4. System displays the details of the order. 5. System asks for confirmation. 6. User confirms the order. 7. System records the order for future execution.
Post-conditions	Buy order has been created and will be activated based on the stock price levels.

In the space below, write the details of **one** alternative, extension or exception scenario. Be sure to begin with the number of the first step where the alternative scenario differs from the main success scenario. If the alternative scenario rejoins the main success scenario, please indicate at which step. (10 points)

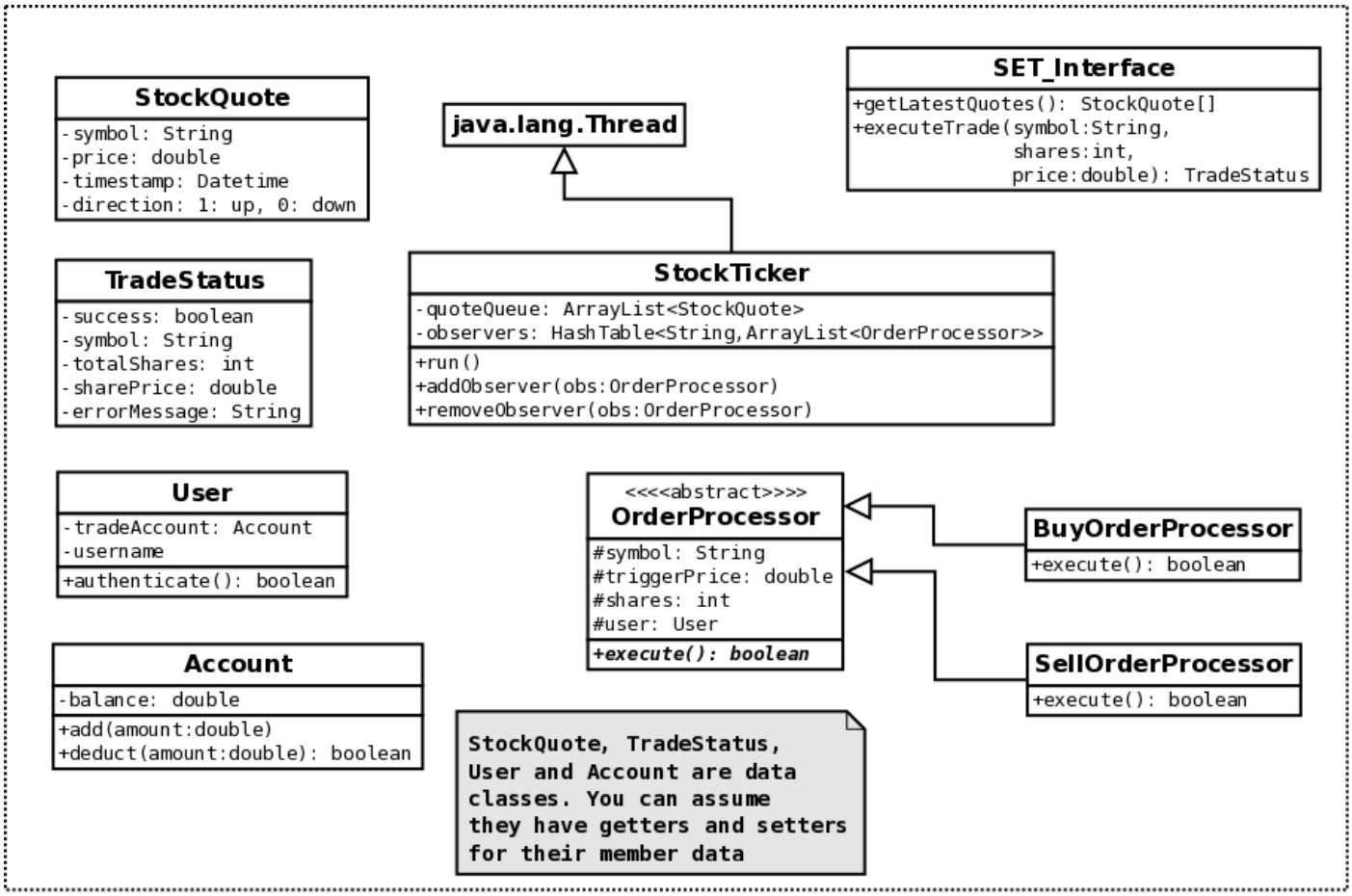
There are a variety of possibilities:

User enters invalid stock symbol (1)
User enters an invalid number of shares (2)
User enters an invalid trigger price (3)
User decides not to confirm order (6)

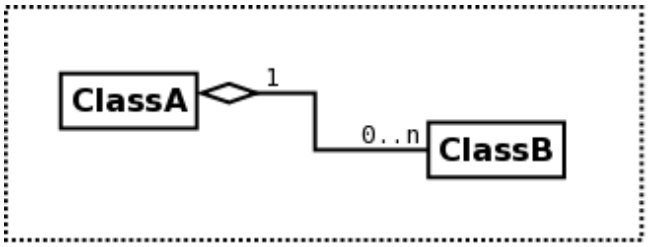
There could be others. The point of this question is to see if the student understand the concept of main success scenario versus alternative scenarios.

A number of people specified a scenario where the user supposedly did not have enough money in his account for the proposed trade. In fact, this would not be checked until the trade was triggered, and thus is not an extension to this particular use case. However, I still gave you full credit.

Below you will find a partial class diagram for this system, focusing on the trading functionality. This diagram is similar to but more detailed than the class diagram we studied in the second design patterns lecture.



4. For simplicity, this diagram does not include aggregation or composition relationships. However, there are several such relationships that are implied by the member data items. In the space below, describe **two** such relationships. Each description should include: 1) the "to" class – the one that would have the diamond pointing to it if we drew the relationship; 2) the "from" class; 3) the multiplicity on the "to" side; 4) the multiplicity on the "from" side. For instance, if we had the diagram below, "to" would be ***ClassA***, "from" would be ***ClassB***, the "to multiplicity" would be ***1*** and the "from multiplicity" would be ***0 to n.***



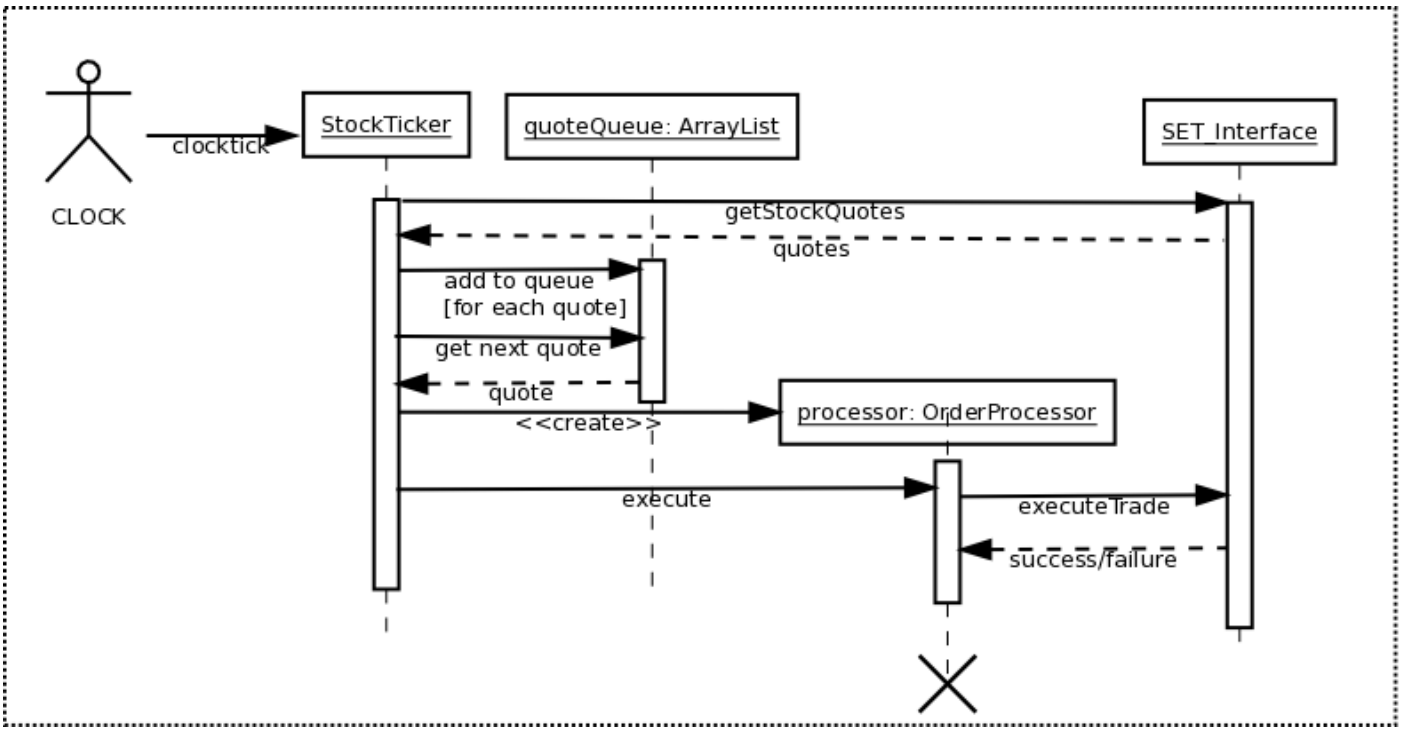
Write your answers below. (5 points)

To User from Account – 1 to 1 – this would actually be a composition relationship, not aggregation
To OrderProcessor from User – 1 to 1
To StockTicker from StockQuote – 1 to 0...n
To StockTicker from OrderProcessor – 1 to 0..n

5. The stock trading system class diagram on page 4 includes an example of one important design pattern that we have studied. What is the design pattern and what classes participate in this pattern? (5 points)

Observer pattern. StockTicker is the observable and OrderProcessor is the observer.

6. Below you will find a sequence diagram for the ***run()*** method on **StockTicker.** This method, which overrides the ***run()*** method from the **Thread** class, can be viewed as an infinite loop that is triggered by the system clock.



This sequence diagram is not logically consistent with the class diagram on page 4. What are the main issues or errors and how would you fix them? You do not need to draw a new diagram, just specify what changes are needed in this diagram. (10 points)

This sequence diagram does not correctly show the way OrderProcessors are handled. The StockTicker does not create OrderProcessors. Instead some other class (not shown) creates an instance whenever the user enters a new order. Then the OrderProcessor is added to the observers hash table in the StockTicker using the addObserver method.

When the StockTicker dequeues a StockQuote, it will use the symbol to look up all the OrderProcessors associated with that symbol, and call the execute method on each one. Then it will remove that observer since orders are only supposed to execute once.

You do not need to describe the problem in this much detail. You can get full credit if you say that the StockTicker does not create OrderProcessors. The fact that I used the "wrong" method name is not important.

7. What is ***polymorphism***? Give a specific example from the lectures, this exam, or your project. (5 points)

Polymorphism occurs when an abstract superclass or interface defines a method, which is implemented differently in different extending/implementing subclasses.

A program can call the polymorphic method on instances without knowing the specific subclass. Nevertheless, the correct method in the specific subclass with be invoked.

Examples include the draw method in the AbstractShape homework exercises, and the execute method in the OrderProcessor class, implemented in BuyOrderProcessor and SellOrderProcessor.

Almost everyone got full credit for this question.

8. What are “code smells” and why are they important? Give an example. (5 points)

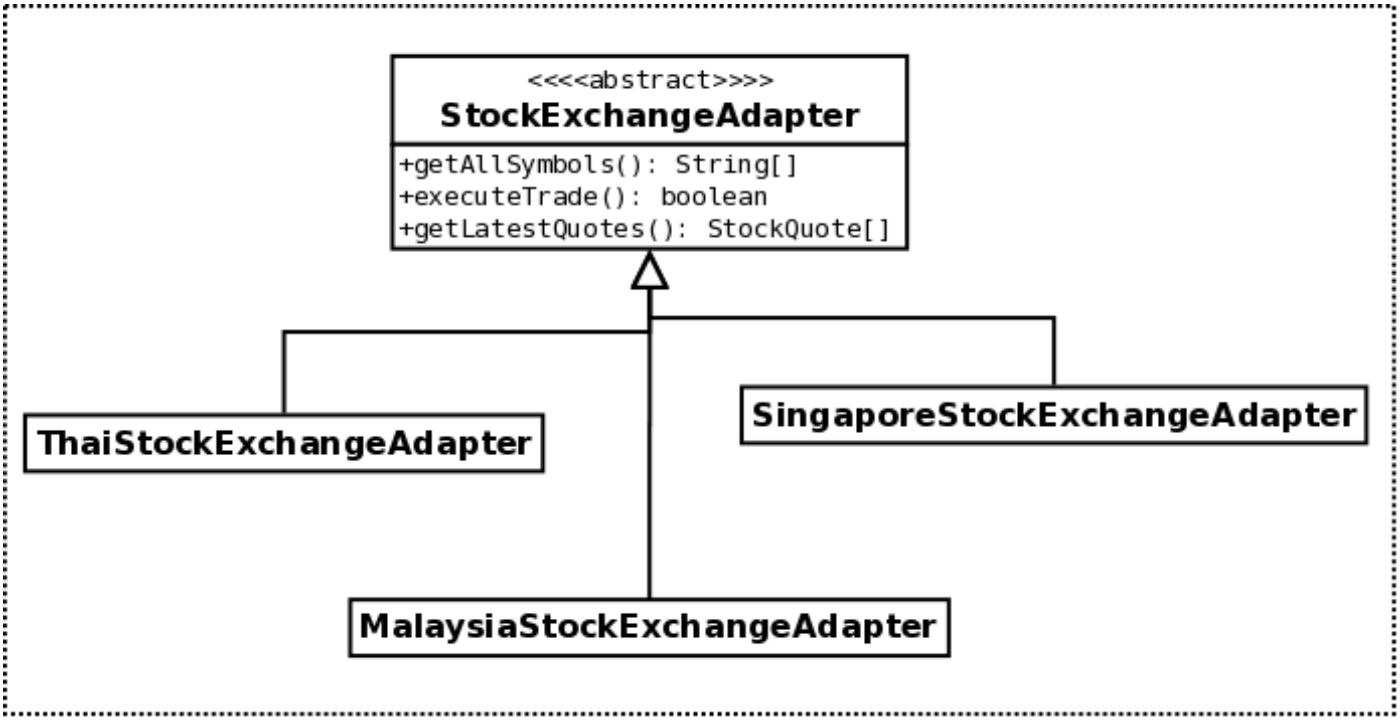
Code smells are characteristics of source code that may indicate that the design is poor and refactoring is needed. Examples include overly long methods, “God” classes, inappropriate intimacy (one class calls methods only in one other class), feature envy (one class consistently calls methods in another class and implements no important behavior of its own), etc. See the Wikipedia article.

9. The class diagram on page 4 assumes that the online stock trading application will be able to trade only with the Stock Exchange of Thailand (SET). Now suppose that the company that owns this application asks us to extend it so that their customers can also view stock prices and do trading on the Malaysian, Singaporean and Vietnamese stock exchanges. Each of these stock exchanges has its own API, which differ from the SET API and from each other. How would you change the class diagram to satisfy this request? What design pattern(s) would you use?

You can answer this question in words, or sketch a modified class diagram below. (10 points)

Revise the class design to use the Adapter pattern. Rather than directly calling the functions in SET_Interface, the system would now have an Adapter class for each stock exchange. All the Adapter classes would have identical methods available. The subclass for each stock exchange would hide the details of differences between them.

A few people suggested using a Factory pattern. This is also a reasonable suggestion, where your factory would instantiate a different interface class depending on which exchange was involved. I gave these people full credit if they could explain how the Factory pattern would work in this situation.



10. In Java, I create the following class to represent the general idea of a bank account:

```
public class BankAccount
{
   /** current amount of money in the account */
   protected double balance = 0;

   /** account number */
   protected int accountNumber;

   /** owner name */
   protected String ownerName;

   /** constructor sets account number and owner */
   public BankAccount(String owner, int acctNumber)
   {
      ownerName = owner;
      accountNumber = acctNumber;
   }

   /** deposit some money */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /** withdraw some money - return amount withdrawn
    *  or -1 if balance is not sufficient.
    */
   public double withdraw(double amount)
   {
      if (balance < amount)
          return -1.0;
      else
      {
          balance = balance - amount;
          return amount;
      }
   }
}
```

Later, I decide that I need to create a subclass of **BankAccount** called **SavingsAccount** for savings accounts. The subclass will have an additional member data item for the ***interest rate*** (a double representing a percentage per month), and an additional method called ***applyInterest()*** to calculate and add the interest to the balance. This method will take take the number of months as an argument. Remember that you need some way to set the interest rate.

On the next page, write the Java code for the **SavingsAccount** class. (10 points)

(Write code for question 10 here.)

```
public class SavingsAccount extends BankAccount
{
   /** interest percent per month */
   protected double interestPercent;

   /** constructor sets account number and owner */
   public SavingsAccount(String owner, int actNumber,double interestRate)
   {
      super(owner,acctNumber);
      interestPercent = interestRate;
   }

   /** increase the balance based on the interest rate
    * This function compounds the interest, that is,
    * it updates the balance for each month.
    */
   public void applyInterest(int numMonths)
   {
      int mon = 0;
      double accumulatedInterest = 0;
      for (mon = 0; mon < numMonths; mon++)
          {
          accumulatedInterest = balance * (interestPercent/100.00);
          balance += accumulatedInterest;
          }
   }
   /* as an alternative to setting the interest rate in the constructor,
    * you could give it a default value and then provide a
    * setter method so the value could be changed.
    */
}
```

To get full credit for this question:

- You must have included a constructor

- You must NOT have included a fixed interest rate with no way to change it.

- The constructor must have arguments for the owner and account number

- Your algorithm for calculating the interest could have been either simple or compound, but needed to add the interest to the current balance.

11. Below you will find definitions for a number of words or concepts important in object oriented analysis and design. Write the correct term in the space below each definition. (10 points – 2 for each definition)

a) A Java construction that defines a set of methods but no data. Individual classes implement these methods so that they will share behaviors. A widely used example is **java.util.Iterator**.

interface

b) A design pattern used to separate the implementation of a user interface from the system state and data upon which the user interface operates.

model-view-controller or MVC

c) A situation where a class has multiple methods with the same method name but different types of arguments.

overloading

d) A situation where a class provides a new implementation of a method that is also implemented in its superclass.

overriding

e) A special method that creates new instances of a class.

constructor

I also accepted “factory method” for this question.

12. Most object-oriented languages allow the programmer to explicitly control the ***visibility*** of both member data and methods. What does this mean? Why is this a good thing? (10 points)

Controlling visibility means that the programmer decides which data can be viewed and modified, and which methods can be called, by other classes, and which data and functions should be hidden.

By restricting visibility to only expose the minimum amount of information, the programmer can reduce dependencies (coupling) between classes. This should reduce bugs and also make future modification of the code easier.