Solid Research - Task 3

The Problem

We have several merchants from where we can purchase items (such as apples). For simplicity let us assume that we have one type of item, whose price and quantity (number of items in stock) varies over time. Therefore, both the price and the quantity of this item can go down as well as up. Also note that each merchant can sell its goods at a different price from the other merchants and one merchant does not affect the other merchants.

The following table shows three merchants together with their prices and quantities available.

Table 1: Merchants Prices and Quantity

Merchant	Price	Quantity
Merchant A	Euro 1.78	3
Merchant B	Euro 1.82	2
Merchant C	Euro 1.84	6

You need to write a program that allows us to purchase a number of items from one or more merchants. This program needs to be clever enough to purchase the cheapest items as quickly as possible. For example, if this item is sold at Euro 1.78 from *Merchant A*, while the other merchants are selling this at Euro 1.82, then the program should purchase this item from *Merchant A*. In the event that the cheapest merchant does not have enough items to fulfil our order, then we need to purchase the remaining items from the next cheapest merchant as shown in the following table.

Table 2: Example of optimal allocation of 4 different orders

	Number of Items we are willing to Purchase	Merchant A	Merchant B	Merchant C
Scenario 1	6	3	2	1
Scenario 2	2	2		
Scenario 3	4	3	1	
Scenario 4	20	3	2	6

In Table 2 we assumed that the price and quantity of each merchant are as those shown in Table 1. In Scenario 4, where we want to purchase 20 (twenty) items in total, we only managed to purchase 11 (eleven) items.

It is important to note that the prices may change quickly. Thus you need to obtain the values (prices and quantities) **from all merchants at the same time, otherwise you may be using stale values**.

Interfaces

Your program will interact with the merchants using the following interfaces. The merchant is represented by the following `Merchant` interface and defines only two methods.

```
public interface Merchant {

  Quote quote() throws Exception;

  OrderResponse order(Order order) throws Exception;
}
```

Before we can purchase any items from a merchant we need to obtain a quotation using the `quote()` method as defined above. This method returns an object of type `Quote`, which is described as follows:

```
public interface Quote {

  int getQuantity();

  BigDecimal getPrice();
}
```

This object provides two properties which can be used to determine the available quantity and the price of the item from this merchant at a given point in time. With reference to Table 1, *Merchant A* will reply with a quotation of 3 (three) as its quantity and 1.78 (one Euro and seventy eight Euro cents) as its price.

The goods can be purchased through the `order()` method which takes an object of type `Order`, described next.

```
public interface Order {

  int getQuantity();

  Quote getQuote();
}
```

The `Order` interface defines two properties: the quotation that was obtained from the same merchant when invoking the `quote()` method; and the number of items to be purchased from this merchant. With reference to Scenario 2 in Table 2, where we want to purchase only 2 (two) items, the order in this example will have a quantity of 2 (two) together with the quotation that was retrieved from *Merchant A*. This quotation has a quantity of 3 (three) and a price of 1.78 (one Euro and seventy eight Euro cents).

The merchant will respond with the `OrderResponse`, described next

```
public interface OrderResponse {
  int getQuantity();
}
```

The quantity that is found in this object provides the number of items that you were able to purchase at the requested price from this merchant.

Note that between the time you requested a quote and the time you made the order, the quantity (and the price) of the item may have changed and the merchant may not be able to provide the quantity you need. Thus, through the `OrderResponse` you can determine how many items you managed to purchase at the prices indicated by the quotation. 0 (zero) quantity means that you did not purchase anything, while a quantity of 10 (ten), means that you were able to purchase 10 (ten) items from this merchant (at the prices indicated by the quotation).

Let assume that *Merchant A* has a quantity of 3 (three) and you want to purchase 3 (three) items from this merchant, as it is the cheapest merchant. Also, let assume that between when the quotation was obtained and when the order was made the quantity for this item drops to 2 (two). In this case the merchant can only sell you 2 (two) items and thus it will return an instance of order response with a quantity of 2 (two). The following table shows more examples of this.

Table 3: Quantity changes between quotation and order

	Quantity to purchase	Quantity at quote time	Quantity at order time	Order response quantity
Scenario 1	3	3	2	2
Scenario 2	5	3	4	4
Scenario 3	2	3	3	2

It is important to note that both the `quote()` and the `order()` methods are executed over the wire and thus may take some time to complete, such as a couple of seconds. These methods may also fail, in which case you can assume that the request did not go through. Anything that causes an exception is considered as a failure.

If a merchant fails to return the quotations (that is, the `quote()` method throws an exception), simply ignore this merchant. In this case only consider the merchants that returned a quote. If the process fails while placing the order (that is, the `order()` method throws an exception), simply consider this order as failed and continue as if you purchased nothing from this merchant.

The Solution

The program that you have to develop should provide an algorithm that is able to obtain a quotation from a set of merchants and make a decision based on the information obtained. You do not have to write a simulator for the merchant's price and quantity changes over time, but simply focus on the algorithm that obtains quotations from all available merchants and makes the required orders.

Furthermore, it is important to note that you need to obtain the prices and quantities **from all merchants at the same time, otherwise you run the risk of using stale (out-dated) values**.

An example of the algorithm's method signature is as follows.

```
public int purchase(int quantity){
}
```

The method takes in the quantity that needs to be purchased, requests a quotation from the available merchants and then purchases the items from the cheapest merchant. Finally it returns the total number of items that it was able to purchase from the available merchants.

What to turn in

1. A Java program that solves the problem described above. The program should work with Java 1.7 (http://www.oracle.com/technetwork/java/javase/downloads/index.html) and can make use of third party libraries.
2. The program needs to be Maven (http://maven.apache.org/) compliant and all dependencies must be managed through Maven. Furthermore, the build process should also invoke all tests.
3. There is no need to have a `main()` method and should not expect any user interaction. Only the JUnit (http://junit.org/) tests will be executed. Any required inputs should be provided by the test cases.
4. While you are free to use any third party libraries, please make this application as light as possible avoiding any heavy frameworks that require setup such as JEE. You are more than welcome to use Spring (http://www.springsource.org/spring-framework) or Google Guice (https://code.google.com/p/google-guice/) frameworks to manage your object dependencies.
5. This program does not communicate with a DB
6. This program does not need to create real merchants or simulate the price and quantity changes over time. It should make use of simple mocks that return the required values.
7. This program does not have to create a client/server application. It should simply use mocks to simulate the required scenario as described above. There is no need to create a full-fletched client/server application. Only implement an algorithm that solved the purchasing problem described above using the mocks.
8. Make sure you include test cases (using JUnit) that cover the following scenarios:

   a) Purchase all from one merchant
   b) Purchase from two merchants (the one with the best price will not have enough quantity)
   c) Simulate failure at both the `quote()` and the `order()` methods and proceed with the next merchants. In this test we assume that one of the merchants fails and the algorithm needs to carry on and purchase from the other merchants that did not fail.
   d) Purchase from no merchants as all fail