Senior Developer Test

Maximum Time for This Test: 3 hours (180 mins)

Scenario

You are writing a small part of a banking system for a company called Acme Bank:

1. Acme Bank runs only 2 types of accounts, namely, a savings account, and a current account.

2. Most of the behaviour in these 2 types of accounts is very similar eg. deposit, withdraw, calculate interest, transfer, etc.

   Note, however, that the actual implementation of some of this functionality ***varies significantly*** between these 2 types of Accounts.

   As such, the functionality for these 2 types of accounts must be implemented in 2 different classes: SavingsAccount and CurrentAccount.

   Note that despite the similarity in some of their functionality, in the real world, there is also lot of functionality between these 2 accounts that are different.

3. For the purposes of this exercise, we will not look at all the functionality, **we will only implement the "withdraw" and "deposit" functionality.**

4. For a Savings Account, withdraw works as follows:

   a. A Savings account must have a minimum balance of R1000.00 at all times
   b. A Savings account's balance is decreased by the amount withdrawn

5. For a Savings Account, deposit works as follows:

   a. A Savings account can only be opened if you deposit at least R1000.00 in it
   b. A Savings account's balance is increased by the amount deposited

6. For a Current Account, withdraw works as follows:

   a. Current accounts can have an overdraft limit (the maximum overdraft limit allowed on a Current Account by Acme Bank is R100 000.00).

   b. This means that a current account can have both a positive or negative balance.

   c. One cannot withdraw more than the (balance + overdraft limit) on a current account.

      Thus, as an example, if one has a current account with a positive balance of R5

000.00, and an overdraft limit of R25 000.00, then the maximum one can withdraw is R30 000.00.

As another example, if one has a current account with a negative balance of R3000.00, and an overdraft limit of R20 000.00, then the maximum one can withdraw is R17 000.00.

7. For a Current Account, deposit works as follows:

    a. A Current account has no minimum deposit requirement
    b. A Current account's balance is increased by the amount deposited

Todo

Write a basic implementation of the withdraw functionality (as described above) using the following interface:

```
public interface AccountService {
        public void openSavingsAccount(Long accountId, Long amountToDeposit);

        public void openCurrentAccount(Long accountId);

        public void withdraw(Long accountId, int amountToWithdraw)
                throws AccountNotFoundException, WithdrawalAmountTooLargeException;

        public void deposit(Long accountId, int amountToDeposit)
                throws AccountNotFoundException;
}
```

Things to note:

1. All code should be written in a package called com.acme.test01.*yournameandsurname*

2. Each account has a unique attribute called "id" that identifies it – this is a technical id, and has no business value. Accounts are considered equal if they have the same technical id.

3. Customer need not be modeled as a class in this exercise. You may simply use an attribute such as "customerNumber" (with type String) on the account to model the customer that owns that account.

4. Please model and implement SavingsAccount and CurrentAccount in the way you consider to be the best way from an Object Orientation perspective.

5. Note that to simplify working with numbers (for the purposes of this evaluation) you may implement all money related figures as integers. Of course you may choose to before more ambitious.

6. As we will not be accessing a real database as part of this exercise, please implement an **in memory map or set of objects** as your database. This should be done in a **singleton** class called SystemDB. This singleton should be pre-populated with a few hardcoded Accounts (both Savings Accounts and Current Accounts) – do this in the constructor of SystemDB. Note that the SystemDB class should only manage the prepopulated objects - there is not need to write an add, delete, or any update logic. Obviously, withdrawals should only be allowed against accounts that actually exist in your database.

   Please prepopulate the database with at least the following accounts (if you want, you can add more accounts):
   a. Savings account (customerNum=1, balance = R2000)
   b. Savings account (customerNum=2, balance = R5000)
   c. Current account (customerNum=3, positive balance = R1000, overdraft = R10000):
   d. Current account (customerNum=4, negative balance = R5000, overdraft = R20000)

7. Note that AccountNotFoundException is a runtime exception and WithdrawalAmountTooLargeException is a checked exception, and must be implemented by you as such. Hopefully, the purpose of these exceptions is obvious, and hopefully it is obvious as to when these exceptions should be thrown. If not, please consult the person who handed you this test.

8. If you are familiar with the principles of layered architectures, then please try and implement the above functionality using these principles. If not, please implement it as you see fit. Note that if you use a layered architecture in your solution, please the SystemDB as a real DB, and design/implement accordingly.

9. If possible, write a few tests (in code) that tests SavingsAccount, CurrentAccount and AccountService to make sure that it is working correctly. If you are not sure how to do

this, then please test your code as you would normally do so. If you do write tests in code, feel free to use your normal approach to this.

10. The code needs to work correct at a basic level. It does not have to be absolutely 100% perfect in terms of catering for a live environment. Please feel free to comment areas that you may implement in more detail in a live environment.

11. In implementing the service/s and domain objects (SavingsAccount and CurrentAccount), please try to use simple Java Objects – DO NOT use EJB's.

12. The use of any of the extended java features introduced in later versions of java, such as generic types, streams, closures, etc. may add additional strength to your evaluation.

13. No User Interface is required to be written, as one should write basic tests (in Java) to ensure your code is working – if you are not use to writing tests in code, then please write a Main class to check that your code is running reasonably ok.

14. As no logging library is provided (such as log4j), you can either use Java Logging or simply use System.out (if you need to do any logging at all).

15. Feel free to use a build solution such as ant, maven or gradle.

16. Please feel free to include any additional commentary on your solution and ideas you may have for extending the solution.

17. You may undergo a subsequent interview regarding your solution, so please be prepared to discuss any design decisions you may have taken / assumed.