SIT771 Object Oriented Development

Pass Task 4.1: Bank Transactions

Overview

For this task, you will pick up from the Bank Account program and add transaction classes which will perform the deposit, withdraw and transfer operations on the bank account.

Submission Details

Submit the following files to OnTrack.

- Your program code (*Program.cs*, *Account.cs*, *WithdrawTransaction.cs*, *DepositTransaction.cs*, *TransferTransaction.cs*)
- A screen shot of your program running

Instructions

In this task, you're going to be created 3 new classes: WithdrawTransaction, DepositTransaction, and TransferTransaction. Objects of these classes can then be used to retain a history of the transactions that have occurred. We will add this transaction history in the next task, for the moment we need to create these classes and get them working.

To start off, let's add a WithdrawTransaction class, which will be able to withdraw money from an account.

1. Create a new C# file named `WithdrawTransaction.cs`, in it, add the WithdrawTransaction class which meets the following UML diagram.

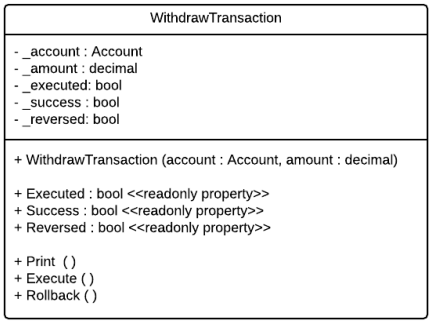

Figure: Withdraw transaction class uml

   - The `_account` field stores the Account object that we are going to withdraw from.
   - The `_amount` field stores how much we want to withdraw.
   - The `_executed` field remembers if the withdraw has been executed. This can be used in `Execute` to ensure that the transaction is only performed once. Externally this is accessible via the `Executed` property.

- The `_success` field remembers if the withdraw was successful, which can be accessed from the read only `Success` property.
- The `_reversed` field remembers if the withdraw was reversed (via the `Rollback` method). This is accessed from the read only `Reversed` property.
- `Print` is used to output the transaction's details.
- `Execute` is used to get the transaction to perform its task. In this case, it will do the withdrawal and remember if this succeeded.
- `Rollback` performs a reversal of the execute if the transaction was successful and has not already been reversed. In this case it deposits the funds back into the account.

The code for part of this is below...

```
public class WithdrawTransaction
{
    private Account _account;
    private decimal _amount;
    private bool _executed = false;
    private bool _success = false;
    //TODO: add reversed

    public bool Success
    {
        get
        {
            return _success;
        }
    }

    //TODO: Need reversed and executed property

    public WithdrawTransaction(Account account, decimal amount)
    {
        _account = account;
        _amount = amount;
    }

    public void Execute()
    {
        if ( _executed )
        {
            throw new Exception("Cannot execute this transaction as it has al
        }

        _executed = true;
        _success = _account.Withdraw(_amount);
    }

    public void Rollback()
    {
        //TODO: Throw an exception if the transaction has not been executed
        //TODO: Throw an exception if the transaction has been reversed
        //TODO: Remember that we have reversed
        //TODO: Then rollback by reversing actions from execute
    }

    public void Print()
    {
        //TODO: print here
    }
}
```

2. Implement the `Rollback` method using the notes in the above code. It will reverse the action performed in `Execute`, but should only run if the transaction has been executed and has not been reversed already.

3. Have a go at implementing the `Print` functionality - it should print to the terminal whether or not

the `WithdrawTransaction` was successful, how much was withdrawn from the account, and a note if it was reversed.

You should be able to get some of this code from the `DoWithdraw` method of your *Program*.

4. Switch back to **Program.cs** and change the `DoWithdraw` method to use your new class. It will need to perform the following steps:
    - Ask the user how much to withdraw
    - Create a new `WithdrawTransaction` object, for the indicated account and amount.
    - Ask the transaction to `Execute`
    - Ask the transaction to `Print`
5. Compile and run the program to make sure that it works correctly.
6. Add a DepositTransaction file and class yourself. It should be modeled off the `WithdrawTransaction` class you've just created, as shown in the following UML diagram.

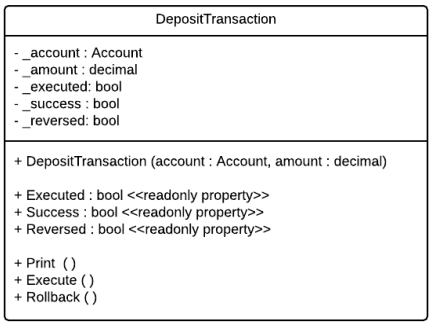

Figure: Deposit Transaction class UML

7. Also change `DoDeposit` in the *Program* to make use of the `DepositTransaction` class.
8. When you have the code, compile and run to make sure everything still works.
9. Add a **TransferTransaction.cs** file and class. The `TransferTransaction` class is a lot like the `WithdrawTransaction` and `DepositTransaction`, however it will require two accounts (a from account and a to account), and internally it can create both a `DepositTransaction` and a `WithdrawTransaction` object to do the work on its behalf. Here is the UML diagram for it:

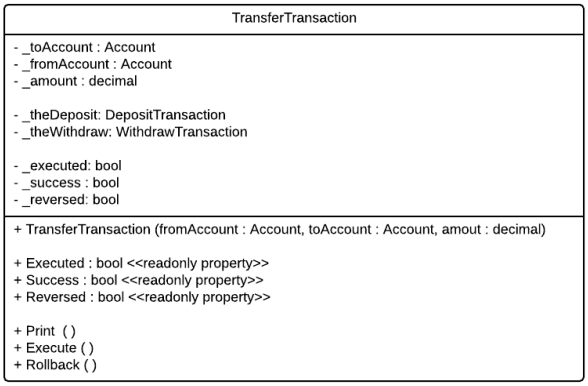

Figure: account transfer class UML

In the constructor, create both the `DepositTransaction` and `WithdrawTransaction` objects, and store them in their respective fields. The `TransferTransaction` object will get these objects to do the basic work for it. This avoids code duplication, which is a good thing.

The `Success` property can be entirely **calculated** for the `TransferTransaction` : it only returns true when both its `DepositTransaction` **and** its `WithdrawTransaction` objects

succeeded (otherwise it returns false). So no need for a `_success` field as with the other transactions.

When Printed, print a summary line eg "Transferred $x from Jake's Account to My Account" (where the account name's come from the respective Account objects). Then get both the withdraw and deposit transactions to print their details. Indent these by asking `Console` to `Write` some spaces before you ask each transaction to print.

The `Execute` method will use the `Withdraw` and `Deposit` objects created in the constructor to withdraw and deposit from the two accounts - however you will need to ensure that the withdraw was successful before you can deposit into the other account. Here is some pseudocode for this method:

- Throw an exception if the transaction have been executed already.
- `Execute` the WithdrawTransaction
- if the WithdrawTransaction was successful
  - `Execute` the DepositTransaction
  - if the DepositTransactoin was **not** successful
    - `Rollback` the WithdrawTransaction

The `Rollback` method will reverse the use of the `Withdraw` and `Deposit` objects. Here is some pseudocode for these methods:

- Throw an exception if the transaction has not been executed.
- Throw an exception if the transaction has been reversed.
- if `theWithdraw` was successful then
  - `Rollback` the WithdrawTransaction
- if the DepositTransaction was successful
  - `Rollback` the DepositTransaction

10. Switch back to **Program.cs** and update to include options to transfer between accounts.

    Have your program transfer from Jake's account to your account.

11. Run the program and deposit, withdraw, transfer and print, create a screenshot and upload alongside code files to OnTrack!

Remember to backup your work, and keep copies of everything you submit to OnTrack.

Task Discussion

For this task you need to discuss the use of classes to create a number of roles that work together to perform required functionality. Here are some guides on what to prepare for:

- Explain how classes can define different roles for objects to play. How does this relate to the core principle of abstraction?
- How does encapsulation relate to the TransferTransaction class? From an external perspective do we know/care that it uses other objects internally to perform tasks on its behalf?