SIT771 Object-Oriented Development

Pass Task 7.1: Abstract Transactions

Focus

Make the most of this task by focusing on the following:

- Concept

  Focus on learning diverse Object-Oriented Programming (OOP) principles, including inheritance, abstraction, and polymorphism, understanding their applications, methodologies, and contextual relevance in software development.

- Process

  Focus on instilling crucial skills like modular and efficient code design, a solid understanding of object-oriented principles, and the ability to minimize code duplication preparing yourself for collaborative, maintainable, and scalable software development practices.

Overview

For this task, we're going to take a look at creating an abstract Transaction class, which our Withdraw, Deposit and Transfer will inherit from. Doing this will reduce code duplication and allow us to simplify code in certain areas of our program!

Submission Details

Submit the following files to OnTrack.

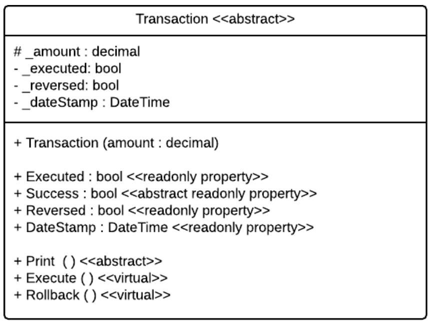
- Your program code (*Program.cs*, *Transaction.cs*, *Bank.cs*, *Account.cs*, *WithdrawTransaction.cs*, *DepositTransaction.cs*, *TransferTransaction.cs*)
- A screen shot of your program running.

Instructions

In this task, you’re going to be creating 1 new abstract class: **Transaction.cs**

To start off, let’s add a Transaction class:

1. Create a new C# file named `Transaction.cs` , in it, add the Transaction class. Here is the UML diagram for the transaction class:



```
Transaction <<abstract>>
---------------------------------
# _amount : decimal
- _executed: bool
- _reversed: bool
- _dateStamp : DateTime
---------------------------------
+ Transaction (amount : decimal)

+ Executed : bool <<readonly property>>
+ Success : bool <<abstract readonly property>>
+ Reversed : bool <<readonly property>>
+ DateStamp : DateTime <<readonly property>>

+ Print ( ) <<abstract>>
+ Execute ( ) <<virtual>>
+ Rollback ( ) <<virtual>>
```

Figure: UML class diagram of Transaction class

   The `#` symbol represents `protected` : a new scope modifier that means this class member is available within this class and any child classes, but not other classes.

2. Implement the abstract Transaction class.

   - Add the protected `decimal amount` field, and the private `DateTime _dateStamp` , `bool _executed` , and `bool _reversed` fields.
   - Implement the abstract readonly property for `Success` and the other properties.
   - Add a constructor that sets the value of the `_amount` field.

   - Add the **abstract** `Print()` method. This will be a placeholder that the different transactions will override and provide details for.
   - Add the virtual `Execute()` method. It should perform the following steps:
      - Throw an exception if the transaction has already been executed.
      - Set `_executed` to true
      - Assign `_dateStamp` the current time. You can get the current time using the `DateTime.Now` property from C#.
   - Add the virtual `Rollback()` method. This should operate in a similar way to `Execute`

3. Modify `DepositTransaction` , `WithdrawTransaction` and `TransferTransaction` to inherit from `Transaction` .

   The main details of these classes will not change, so you only need to set these up to workwith the new Transaction class. There should not be large changes needed.

   - The constructor's should call the base class constructor, passing up the amount, and then in its body it will assign the `_account` (s) as necessary.
      - Transfer will need two private account fields, the `_toAccount` and the `_fromAccount` which should be assigned in the Transfer constructor.

   For example:

   ```
   public WithdrawTransaction(Account account, decimal amount) : base(amount
   {
       _account = account;
   }
   ```

   - Execute should **override** the method from the base class:
      - call `base.Execute();` to set the timestamp and executed.
      - perform the transaction on the account.
      - it must be marked as `override` to indicate that this method is changing how. `Execute` works for this class of `Transaction` .
   - Rollback will be similar to execute but will reverse the operations.
   - Implement `Print` (overriding the abstract method from the Transaction class).

Now that our three types of transactions are all inherited from the `Transaction` class, we can store a list of transactions in the accounts and in the bank!

1. Modify the `Bank` class to have a private field `private List<Transaction> _transactions` .
2. Replace the three ExecuteTransaction methods with a single `public void ExecuteTransaction(Transaction transaction)` method! This will accept a `transaction` argument. It will perform the following steps:
   - Add the passed in Transaction to the list of transactions

        - Tell the `transaction` to Execute.

3. Finally, create a `public void PrintTranscationHistory()` method, which iterates through the `_transactions` , asking each one to `Print` .

4. Check your Program class and make sure that the Transactions (DepositTransaction, WithdrawTransaction, and TransferTransaction) are all being called on your Bank object.

5. Add in a menu option to print the transaction history.

6. Run your program and upload a screenshot to OnTrack alongside the program files.

Task Discussion

For this task, you need to discuss the use of inheritance and polymorphism in object-oriented solutions:

- Explain how inheritance and polymorphism are used in the solution.
- How can the one method perform any kind of transaction?
- What changes would you need to make to the Bank to include a new transaction type?
- What are the advantages we get through inheritance?
- What advantages does polymorphism provide?