Practical Task 5.2

(Pass Task)

General Instructions

This practical task asks you to further extend the banking system by adding transaction classes to perform deposit and withdraw operations on a given bank account. In addition, your task is to complement the system with a new operation required to transfer money between two accounts. This all needs development of three new classes: `WithdrawTransaction`, `DepositTransaction`, and `TransferTransaction`. Objects of these classes can then be used to retain a history of the transactions that have occurred. You will implement the mechanism to keep transaction history in the next task. For now, you will only need to create and properly test the three classes. You may use your solution for Task 3.2 as you will mainly add new functionality to the existing project.

1. For the first part of this task, focus on the implementation of the `WithdrawTransaction` class. To let the system to withdraw money from an account, this class employs a set of instance variables listed below.
   - **`_account`** of `Account` data type
     Refers to the `Account` object that the system is to withdraw from.
   - **`_amount`** of decimal data type
     Specifies how much money the user wants to withdraw.
   - **`_executed`** of Boolean data type
     Indicates whether the withdraw operation has been attempted. It ensures that the transaction is only performed once.
   - **`_success`** of Boolean data type
     Indicates whether the withdraw operation has been successfully completed.
   - **`_reversed`** of Boolean data type
     Indicates whether the withdraw operation has been successfully reversed.

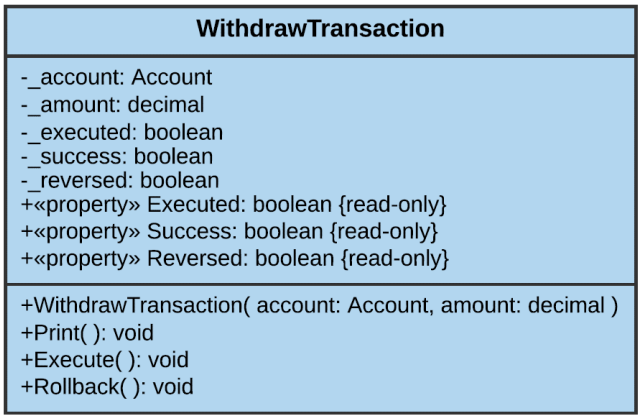
   The following list of public methods forms an interface that the user can use to interact with an instance of the class.

   - **WithdrawTransaction(Account account, decimal amount)**
     Constructor. Initializes a new instance of the `WithdrawTransaction` class. It sets the associated account and the amount to withdraw.
   - **void Print( )**
     Prints the withdraw transaction's details. This method reports the status of the transaction and the amount that was withdrawn from the associated account if that was successfully finished.
   - **void Execute( )**
     Executes the withdraw transaction and its underlying tasks. This method debits the associated `_account` deducting the specified `_amount`. It throws an `InvalidOperationException` when the transaction has been already attempted and specifies the reason for this exception. It also throws this type of exception when the associated account has insufficient funds. When the operation succeeds, it updates the `_success` field of the class setting its value to true. Furthermore, it changes the `_executed` status to true.

- **void Rollback( )**
  Performs a reversal of the preceding withdraw transaction. This method deposits the funds back into the associated `_account` if the withdraw transaction was successful and has not been reversed yet. When the rollback operation is successful, it updates the `_reversed` field of the class. This method must throw an `InvalidOperationException` indicating the particular error when the original transaction has not been finalized or when it has been already reversed.
- **bool Executed**
  Property. Gets the `_executed` status of the `WithdrawTransaction`.
- **bool Success**
  Property. Gets the `_success` status of the `WithdrawTransaction`.
- **bool Reversed**
  Property. Gets the `_reversed` status of the `WithdrawTransaction`.

The `InvalidOperationException` that the both methods may throw is used in cases when the failure to invoke a method is caused by reasons other than invalid arguments. Typically, it is thrown when the state of an object cannot support the method call.

The following UML diagram should help you to understand the content of the `WithdrawTransaction` class.

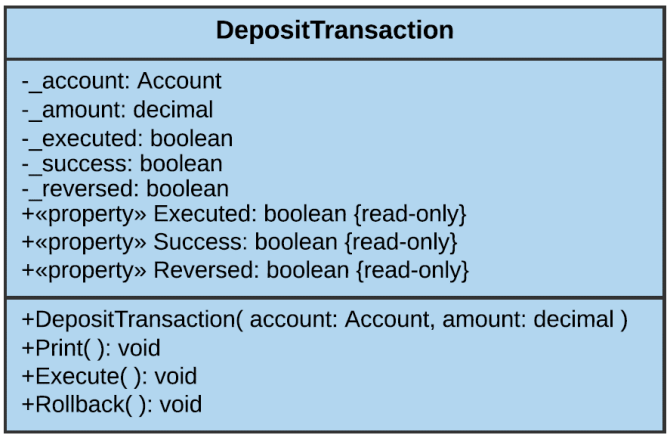


Now, when the `WithdrawTransaction` class is finished, switch to the `BankSystem` class of the project and modify the DoWithdraw method so that it uses the `WithdrawTransaction`. It will need to perform the following steps.

- Ask the user how much to withdraw.
- Create a new `WithdrawTransaction` class object, for the specified account and amount.
- Ask the transaction to `Execute`.
- Ask the transaction to `Print`.

Compile and run the program to make sure that it works correctly. Elaborate on testing and debugging of the new class. Think about how the user can potentially act to crash your program. Focus on possible scenarios.

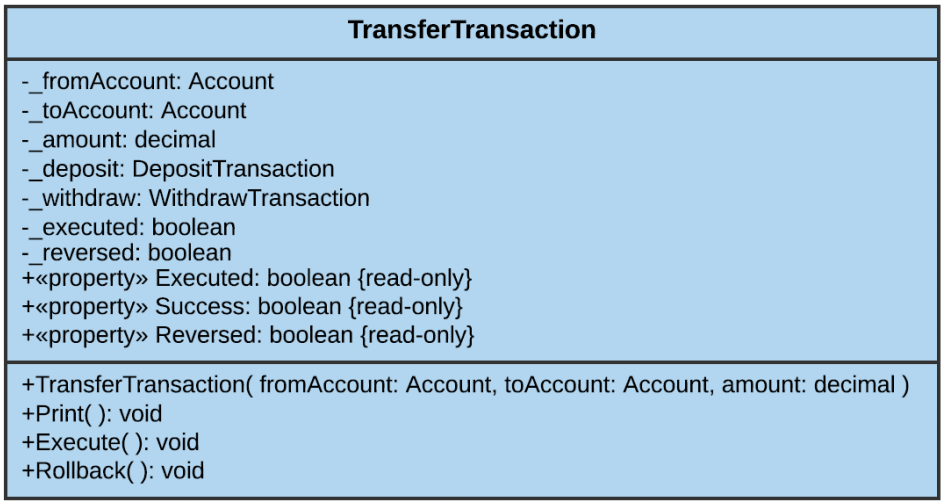
2. For the second part of this task, proceed with the implementation of the `DepositTransaction` class. It should be modelled off the `WithdrawTransaction` class you have just created. The following UML diagram is to detail the content of the `DepositTransaction` class.



Though the `DepositTransaction` is a sort of mirroring of the `WithdrawTransaction` class, some aspects are different. You will need to think about possible exceptions that may occur for this transaction.

Similarly, change the `DoDeposit` method in the `BankSystem` class to make use of the `DepositTransaction` class. It should run a similar sequence of actions as `DoWithdraw`. When you have the code, compile and run to make sure that everything still works.

3. Finally, proceed with the `TransferTransaction` class, which is to be similar to both the `WithdrawTransaction` and `DepositTransaction` class. The only difference is that this transaction involves two accounts: One account represents a source of funds and the other is the account you want to credit. In fact, a transfer can be defined as a compound transaction consisting of two parts: deposit and withdraw. This therefore will need two objects: one of the `DepositTransaction` and the other of the `WithdrawTransaction` class. This can be seen from the following UML diagram:



Complete the `TransferTransaction` class with regard to the UML diagram. Take into account the following facts about the class and how it operates.

- The constructor of the `TransferTransaction` class must create both the `DepositTransaction` and the `WithdrawTransaction` class objects and store them in their respective fields. Indeed, the `TransferTransaction` will simply delegate necessary tasks to these two objects. This structure avoids code duplication which is undesirable in object-oriented programming.
- The `Success` property of the `TransferTransaction` returns true if only both its `_deposit` and `_withdraw` transactions have previously succeeded. Otherwise, it returns false. Therefore, there is no need for a `_success` field like that provided for the other types of transactions.

- The `Print` method must show the account names derived from the respective `Account` class objects. For example, it may output "Transferred $xxx from 04563199454 Andrew Dalwood's account to 54983140452 Michael Klein's account". Then both the `_deposit` and `_withdraw` objects should print their details.
- The `Execute` method involves the `_deposit` and `_withdraw` objects. It must throw an `InvalidOperationException` if the transfer transaction has been already executed. The same exception is valid in case when `_fromAccount` does not have enough funds to proceed. Note that this operation is only valid when the `_withdraw` ends successfully before the `_deposit` starts. If the `_deposit` in the second part of the `Execute` fails, the method must call the `Rollback` on the `_withdraw` object.
- The `Rollback` method will reverse the use of the `_withdraw` and `_deposit` objects. If the transfer transaction succeeded, then this method calls the `Rollback` method on the `_withdraw` and subsequently on the `_deposit` objects. This method must throw an `InvalidOperationException` if the original transfer transaction has not been successfully completed or if the transaction has been already reversed. The same exception must be thrown when the account to debit within the `Rollback` method, i.e. `_toAccount`, has insufficient funds to complete the operation. This, for example, can be the situation when a transfer transaction from `_fromAccount` to `_toAccount` was followed by a withdraw transaction on `_toAccount` that left insufficient funds to call a rollback operation for the former transaction.

Once the `TransferTransaction` class is finished, switch to the `BankSystem` class of the project and include the option to transfer between two accounts. This will need you to extend the UI and implement a `DoTransfer` method, which is similar to the current version of `DoWithdraw` or `DoDeposit` though it operates on two accounts. You therefore will likely need to create a couple of accounts in the `Main` method of the `BankSystem` class before you invoke your banking system.

Run the program and deposit, withdraw, transfer and print. Complete and test your program for potential logical issues and runtime errors.

4. Prepare to discuss the following with your tutor:
   - 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?

Further Notes

- Study the way to program and use classes and objects in C# by reading Sections 2.3-2.6 of SIT232 Workbook available in CloudDeakin → Resources.
- Explore Sections 4.2-4.6 of the Workbook to get explanation of how to implement object/class relationships that you will need to code and integrate the three transaction classes.
- If you seriously struggle with the remaining concepts used in this practical task, we recommend you to focus on reading the entire Sections 2 and 4 of the Workbook.
- The following link will give you more insights on the InvalidOperationException and its application:
  - https://docs.microsoft.com/en-us/dotnet/api/system.invalidoperationexception

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements the required functionality. It must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.