CompSys202/MechEng270
Object-oriented design and programming
Assignment #2 (24% of final grade)
Due: 1:00pm, 17 September 2019

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- Gain confidence modeling a problem.
- Gain confidence programming in C++.
- Apply OOP concepts such as inheritance and encapsulation.
- Apply good coding practices such as naming conventions and code reuse.

1 The Financial Management System

In this assignment you will simulate the management of a financial system. Your program will allow this system to manage customers which store money in different types of accounts by providing functionality to represent users of the system (customers), represent accounts of different types and the transactions between these accounts. You have been given headers with class and method declarations for the classes you must implement in parts of this assignment. You are encouraged to add helper methods or data members to these classes where necessary to structure and simplify code, but **you should not change any existing method declarations** (except where specified).

2 Tasks

2.1 Private copy constructors and copy-assignment operators

You will notice in most of the classes we have declared the copy constructors and copy-assignment constructors as `private`. This means you are not allowed to pass these types around by value (review the lecture about copying objects). Instead, you must use pointers to those types. While the declarations are provided inside the respective header files, you only need to provide simple implementations in the corresponding files to stop compiler warnings (if any).

2.2 *Refactor* the Money class and *Implement* the Customer class *[Task 1]*

You will need to complete the implementation of the following classes: Customer and Money. The very first test only requires you to return your username (formerly known as UPI) inside `FinancialManagementSystem::author()`.

2.2.1 Refactor the Money class

You will find the Money.cpp and Money.hpp files in the Financial/ directory. These are the example solutions shared with you in the lecture after Assignment 1. The Money class will be used in Assignment 2, to represent the money being passed between different accounts. There will be changes that you need to make to Money to accomodate this.

- Refactor the Money class to allow a Money object to represent a negative value.
- You will now be able to construct, add or subtract so that the resulting value of the Money object is negative.
- *However*, you are not able to add a negative amount or subtract a negative amount. e.g. addDollars(-4) or subtractCents(-4) should have no effect.
- This will be important for implementing the logic described later in the assignment.

Below are additional details about the methods you will refactor:

- `Money(int dollars, int cents)`

It should now be possible for negative dollars or cents to be passed in as parameters when constructing a Money object.

- `getCents()`, `getDollars()`, `asCents()`

It would now be possible for the above methods to return a negative value

- `addDollars(int dollars)`, `addCents(int cents)`, `subtractDollars(int dollars)`, `subtractCents(int cents)`

The above methods should have no effect if a value less than 0 is passed in as a parameter. e.g. addDollars(-4) should have no effect on the object it is called on.

- `add(const Money &other)`, `subtract(const Money &other)`

The above methods should have no effect if the Money object passed in as a parameter represents a current value of less than 0.

2.2.2 Implement the Customer class

You may add more code to the header, but don't change the existing method declarations. The `Customer` class holds the name of the customer and a unique customer ID that must be different for every Customer instance. You need to declare member variables and implement the constructor, getName and getID methods.

- `Customer(const string &name)`

Implement the constructor of the `Customer` class. The parameter passed into this constructor refers to the name of the customer. You need to set the instance variable name from the parameter, and also ensure a unique ID is assigned to the customer.

- `getName()`, `getID()`

These methods should return the respective instance variable; the customer's name and customer's ID (all set in the constructor).

2.2.3 Testing Task 1

Testing code in this assignment works in a similar way to Assignment 1. When you are ready to test your code, you can compile and run the tests by executing the command `make test` from the command line console. Alternatively you can use Eclipse or another IDE.

Making sure that your program passes all of the tests you have been given will not guarantee that you will get 100% of the marks for this assignment, but it will mean that you get at least some marks for the parts that work according to those tests. Writing your own tests or extending the ones you have been given is strongly recommended, as there are many requirements in this assignment that the existing tests do not cover. Look at the assignment specifications, and think carefully about which specifications were explicitly mentioned but there aren't test cases provided to you.

2.3 Implement the Accounts *[Task 2]*

You will complete the `Account`, `SavingsAccount`, `ChequeAccount`, `CreditAccount` classes defined in the respective header files within the Financial folder (e.g. Financial/Account.hpp). The implementation files for most of these classes have also already been created for you with blank implementations for the methods. The aim of this task is to create objects which will be used to represent accounts that store money. An Account is identified by its unique account ID and created with the customer ID of the customer it belongs to. Moreover, each account has a balance that can be changed by depositing or withdrawing Money. A Customer can never have a base Account be created, instead it must be either a Savings, Cheque or Credit account. The SavingsAccount will have a bonus value associated with it and the ChequeAccount will have a transaction fee on withdrawals. Each of these three account types have different requirements when withdrawing or depositing Money into the account. These requirements are described below:

- *The Account class:*

The account class contains information about the account ID and the customer ID of the customer it belongs to. The account ID for each account must be unique. Different accounts may share the same customer ID, as one customer may have multiple accounts. The Account class also has a balance represented by a Money object, all Accounts should begin with a balance of 0.

`virtual bool depositMoney(Money amount)` The logic for depositing money should be overriden by its subclass. Note that **only amounts of zero or more may be deposited to any account.**

`virtual bool withdrawMoney(Money amount)` The logic for withdrawing money should be overriden by its subclass. Note that **only amounts of zero or more may be withdrawn from any account.**

`int getCustomerID() const` Returns the customer ID of the Customer the account belongs to.

`int getAccountID() const` Returns the account ID of the Account.

`Money getBalance() const` Returns the amount of the current balance as a Money object.

- *The SavingsAccount class:*

A SavingsAccount also includes a bonus for multiple deposits and unique requirements for withdrawing or depositing Money. A SavingsAccount will **have a bonus value that is added to the account on every second deposit**. The **bonus value begins at 10 dollars and cannot fall below 0 dollars**, but will be reduced when withdrawing money from the account (see below). Additionally, a SavingsAccount cannot have a balance of less than 0 dollars.

`bool depositMoney(Money amount)` Add the given amount of money to the balance. Every **second successful deposit**, add an additional amount — equal to the current bonus value — to the balance of the account. Return true if the deposit was successful and false if the deposit was unsuccessful.

`bool withdrawMoney(Money amount)` Subtract the given amount of money from the balance. The bonus value will be **decreased by two dollars** for each *successful* withdrawal. Return true if the withdrawal was successful and false if the withdrawal was unsuccessful.

`Money getBonusValue() const` Returns the current bonus value of the SavingsAccount.

- *The ChequeAccount class:*

A ChequeAccount also includes a transaction fee on **withdrawals** and unique requirements for withdrawing or depositing Money. A ChequeAccount will **begin with a transaction fee of 1 dollar**. Additionally, a ChequeAccount can not have a balance of less than 0 dollars.

`bool depositMoney(Money amount)` Add the given amount of money to the balance. The amount deposited must be less than or equal to 3000 dollars. Return true if the deposit was successful and false if the deposit was unsuccessful.

`bool withdrawMoney(Money amount)` Subtract the given amount of money from the balance. Also subtract an amount equal to the current transaction fee, then after this increase the transaction fee by 1 dollar. Return true if the withdrawal was successful and false if the withdrawal was unsuccessful.

`Money getTransactionFee() const` Returns the current transaction fee of the ChequeAccount.

- *The CreditAccount class:*

A CreditAccount is different than the other account classes as it has a balance that is permitted to be below 0. The positive balance is not allowed to be greater than 5000 dollars. You will need to make sure that you have adjusted the Money class to allow transactions like this to occur.

`bool depositMoney(Money amount)` Add the given amount of money to the balance. The **balance** of the CreditAccount cannot be greater than 5000 dollars. Return true if the deposit was successful and false if the deposit was unsuccessful.

`bool withdrawMoney(Money amount)` Subtract the given amount of money from the balance, allowing the balance to be below 0. Return true if the withdrawal was successful and false if the withdrawal was unsuccessful.

As you can see some of the features are repeated several times. For example all accounts have an account ID and customer ID. Some things are unique, such as only SavingsAccounts have interest rates and only ChequeAccounts have transaction fees. You therefore need to consider your design and where you place members/methods. This decreases the repetition by using inheritance where appropriate. If you implement methods in a base class, you will have to delete the code for the declarations and implementations of those methods in the derived classes if you don't want them overriden.

- Testing task 2

The tests for these additional `Account` subclasses can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T2_TESTS
```

Once the Task 2 tests are enabled, run `make test` to execute all of the currently enabled tests. Remember that not all aspects of these classes are tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

2.4 Implementing the Transaction class *[Task 3]*

In this section you need to implement the Transaction class. A transaction represents the transfer of money between two accounts, withdrawing an amount from one of them and depositing the amount to another account. You will need to complete the constructor and *getter* methods. Each Transaction should be given a unique ID which will be one higher for each Transaction constructed. A Transaction can have three different states, represented by the enum TransactionState. These are FAILED, PENDING and COMPLETED. Transactions should be constructed in the PENDING state. After attempting to perform the transaction, the state will change to FAILED if the transaction was unsuccessful or COMPLETED if successful. A Transaction will be unsuccessful if either the 'from' account is unable to withdraw the specified amount, or the 'to' account is unable to have the specified amount deposited in it.

2.4.1 *bool Transaction(Account *fromAccount, Account* toAccount, Money amount)*

This constructor sets the transaction with the specified account the money is being transferred from, the account the money is being transferred to, and the amount of money being transferred. If the amount is below 0, the transaction amount should instead be set as 0. This should also set the unique ID of the transaction. The first Transaction created will have a unique ID of 1. Every transaction created after this will have a unique ID higher than the previous transaction. e.g The first Transaction created will have unique ID of 1, the second Transaction created could have a unique ID of 2 etc. All transactions should be constructed in the PENDING state.

2.4.2 *getState(), getAmount(), getID(), getFromAccount() and getToAccount()*

These methods should return the respective members.

2.4.3 *bool performTransaction()*

This method is responsible for executing the transaction after it has been created. This method will ensure the *amount* is withdrawn from the *fromAccount* and deposited into the *toAccount*. If this amount is successfully withdrawn and deposited into the other account, the state of the transaction should be updated to COMPLETED and return true. If this failed (either an illegal withdrawal or illegal deposit, see the Account conditions) the state will instead be updated to FAILED and return false. HINT: Remember that to withdraw and deposit between accounts, you must ensure that *both* the withdraw and deposit will be successful before any amounts are added or removed from accounts!

2.4.4 Testing the `Transaction`

Tests for the `Transaction` class have been included in the test.cpp file provided to you. To enable them, uncomment the following line at the top of test.cpp:

```
// #define ENABLE_T3_TESTS
```

This will enable the tests for the `Transaction` class. Save the test.cpp file and run `make test` to execute all of the currently enabled tests.

2.5 Implementing the Financial Management System [Task 4]

The `FinancialManagementSystem` class implements the core functionality of the system. It ties together customers and accounts and controls interaction between them. You need to implement all of the methods that are declared in *FinancialManagementSystem.hpp* file. You can see the description for methods below:

2.5.1 bool addCustomer(Customer *customer) and std::vector<Customer *> getCustomers() const

The addCustomer() method adds the customer to the list of customers (create a vector) of the respective FinancialManagementSystem class. Each customer has a unique ID. This means that the method should return false if the customer is already included in the system (based on ID). Otherwise, it adds the customer to the customer vector and returns true. The getCustomers() method returns the vector of all customers currently stored.

2.5.2 bool verifyCustomer(int customerID) const

The verifyCustomer() method checks that the customer is in the financial management system. This means that the method should return false if the customer is not already included in the system (based on customer ID). Otherwise, if the customer is stored in the financial management system, this returns true.

2.5.3 bool addAccount(Account *account) and std::vector<Account *> getAccounts() const

The addAccount() method adds the account to the list of account (create a vector) of the respective FinancialManagementSystem class. Each account has a unique ID. This means that the method should return false if the account is already included in the system (based on ID). This will also return false if the account owner (by customer ID) is not already stored in the financial management system. Otherwise, it adds the account to the account vector and returns true. The getAccounts() method returns the vector of all accounts currently stored.

2.5.4 bool verifyAccount(int accountID) const

The verifyAccount() method checks that the account is in the financial management system. This means that the method should return false if the account is not already included in the system (based on account ID). Otherwise, if the account is stored in the financial management system, this returns true.

2.5.5 bool addTransaction(Transaction *transaction)

The addTransaction() method adds a transaction in the PENDING state to the list of transactions (create a vector) of the respective FinancialManagementSystem class. Both accounts involved in the transaction must already be in the financial services system. If these conditions are met it adds the transaction to the transactions vector and returns true, otherwise the transaction is not added and it returns false. The getAccounts() method returns the vector of all transactions (of any state) currently stored.

2.5.6 bool verifyTransaction(int transactionID) const

The verifyTransaction() method checks that the transaction is in the financial management system. This means that the method should return false if the transaction is not already included in the system (based on transaction ID). Otherwise, if the transaction is stored in the financial management system, this returns true.

2.5.7 std::vector<Transactions *> getTransactions() const and std::vector<Transactions*> getTransactions(TransactionState state) const

getTransactions() should return the vector of all transactions currently stored. If a TransactionState (FAILED, PENDING or COMPLETED) is passed in as a parameter, then only stored transactions in that state should be returned.

2.5.8 std::vector<Transaction*> performPendingTransactions()

This method performs all pending transactions in the financial services system. This should ensure that the amount of the transaction is transferred between the accounts in the Transaction if it is possible. These transactions should occur in ascending order of their unique transaction ID (e.g. Transaction with ID of 5 should be performed before Transaction with ID of 8). You should ensure each Transaction should have its status updated to COMPLETED if successful and FAILED if unsuccessful. The transactions that have been successfully executed during this method call should be returned in a vector of Transactions.

2.5.9 Money getCustomerBalance(int customerID) const

This method should get the total combined balance of all accounts owned by the customer with the customerID passed in as a parameter. These balances should be summed and returned as a single Money object.

2.5.10 Testing the `FinancialManagementSystem`

The tests for the `FinancialManagementSystem` class can be enabled by uncommenting the following line at the top of test.cpp:

```
// #define ENABLE_T4_TESTS
```

Once the Task 4 tests are enabled, run `make` to execute all of the currently enabled tests. Remember that not all aspects of the `FinancialManagementSystem` class may be tested in the tests you have been given, so **don't assume the task is complete once all of the tests pass**.

Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run (not including build time) for longer than 10 seconds will be terminated and will be recorded as failed.
- Although you may add more to them (e.g. member variables, `#include` statements, or helper functions), you must not modify the interface of classes defined in the following files (e.g. do not delete the existing functions declared):
  - `Financial/Money.hpp`, `Financial/Customer.hpp`, `Financial/Transaction.hpp`
  - `Financial/Account.hpp`, `Financial/SavingsAccount.hpp`, `Financial/ChequeAccount.hpp`, `Financial/CreditAc`
  - `Financial/FinancialManagementSystem.hpp`
- Do not move any existing code files to a new directory, and make sure all of your new code files are created inside the `Financial` directory.
- You may modify `test.cpp` as you please (for your own testing purposes); these files will not be marked at all. Be aware that your code must still work with the original `test.cpp`.

- Your code will also be inspected for **good programming practices**, particularly using good object-oriented principles. Think about naming conventions for variables and functions you declare. Make sure you comment your code where necessary to help the marker understand **why** you wrote a piece of code a specific way, or **what** the code is supposed to do. Use consistent indentation and brace placement.
- Also complete destructors of the classes where appropriate. This set will be manually marked, so be sure to think about it and use `delete` for objects created on the heap when they are no longer needed.

Submission

You will submit via Canvas. **Make sure you can get your code compiled and running via command line with the test.cpp file provided in the ECSE Department Linux (Ubuntu) computers** (i.e. where your labs are) **or using the VirtualBox image shared on S-drive**. Submit the following, in a single ZIP archive file:

- A **signed and dated Cover Sheet** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can generate and download this in Canvas, see the Cover Sheet entry.
- The **entire** contents of the **src_for_students** folder you were given at the start of the assignment, including the new files and code you have written for this assignment. Ensure you **execute `make clean` before zipping the folder** so your submission does not include any executable files (your code will be re-built for marking).
- Do NOT nest your zip files (i.e. do not put a zip file inside a zip file).

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions. Get into the habit of downloading them again, and double-checking all was submitted correctly.

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help (struggling a little with it will help you learn, especially if you end up solving it). If you still need help, check on Canvas (if it is about interpreting the assignment specifications) or ask in the Lab help clinics (if you would like more personal help with C++). Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using software similarity tools. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Never put your code in a public place (e.g. Reddit, Github, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

Late submissions

Late submissions will incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)