Zehui Jiang | U68975915 | zhjiang@bu.edu

Online Fancy Bank ATM
DESIGN DOC

All classes are listed below:

- ATM
  - Bank.java
  - User.java
  - Account.java
  - CheckingAccount.java
  - SavingAccount.java
  - Currency.java
- GUI
  - MainMenu.java
  - ReportWindow.java
  - LookUpWindow.java
  - IDLookUp.java
  - NameLookUp.java
  - README.java
  - TestAccounts.java
  - AccountMenu.java
  - RegisterWindow.java
  - DepositWindow.java
  - WithdrawWindow.java
  - ExchangeWindow.java
  - TransferWindow.java
  - TransferCurrencyWindow.java
  - LoanWindow.java
  - RepaymentWindow.java

For ATM classes, they are designed in a hierarchical manner.

Bank.java

This class represent the bank that contains all the information about users and account details. To be specific, this class has two main variables: a list of User objects and a list of id strings. This design keeps track of both Users objects and unique account ids, which is an injective reference to Account object. Thus, this is a generic class and no changes are needed even if the specific requirements about the account or user is changed. Besides, there is one another variable: a list of transaction history. It is used to implement the report function for manager. It's stored in this class because only this class has higher hierarchy that all other classes and can contain transactions from all accounts.

User.java

This class represent the users. There are two variables in this class, a list of Accounts that belongs to this user since each user can have unlimited accounts, and a String name representing the user name. This design is out of the same purpose with Bank.java that when requirements are changed, we want code changes to be as less as possible. Hence, with a nested list of Account, it would be a generic class that can handle all kinds of accounts.

Zehui Jiang | U68975915 | zhjiang@bu.edu

Account.java

The class with most lines of codes in ATM classes. This class contains all the variables that an account would have: a list of currencies, user name, account id, all kinds of fee and account type, etc. Please note that here, the three currencies are hard coded into the list. Actually, the optimal and regular solution would be storing all currency information in database, then we use a dataDao class to import all the currency information and write it into the list of currencies. Since we are not at that phase yet, hardcoding it is the only way to do it. This class also contains all the method that an account is capable of: deposit, transfer, withdraw, exchange and loan/repay. Each one of those actions has one corresponding method in this class. This design is very natural and implemented all the necessary functions of an ATM machine. Another good thing is that when account requirement is changed, all we need is to modify this class.

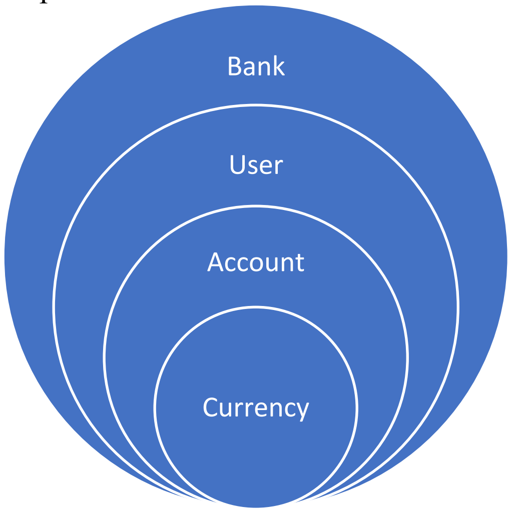
CheckingAccount.java & SavingAccount.java

These two classes are subclasses of Account.java. However, both of them doesn't contain any unique variable or methods. Because the two types of account have exactly the same variables and methods. The only difference is that in the constructor, they pass different parameter to the super class constructor. This is actually meaningful because we won't be confused about different settings of variables with this design. Moreover, in the future when there are more types of account is added, it's easy to add new classes with different variable settings and new methods.

Currency.java

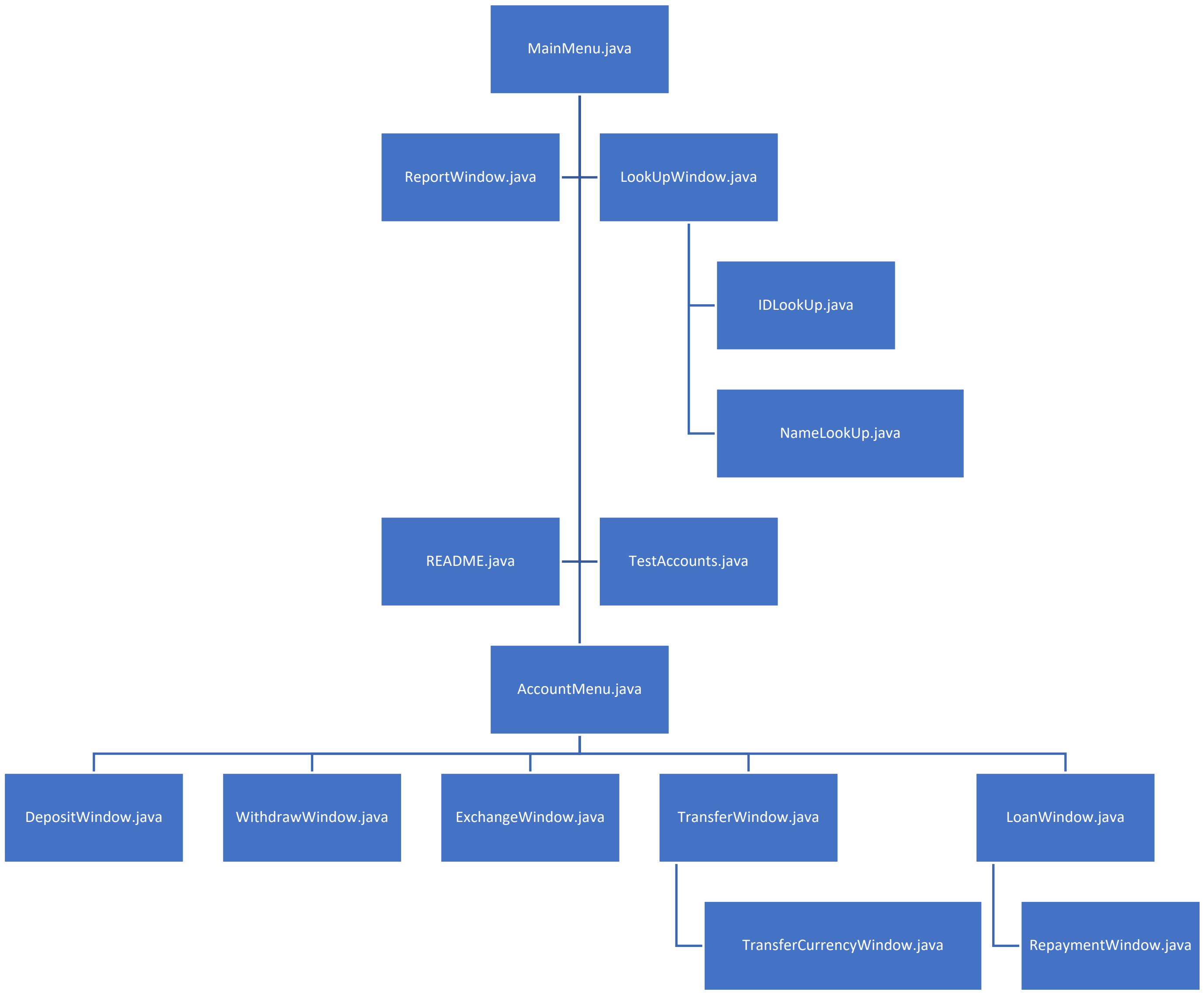
This class represent a kind of currency in accounts. Each Account object can contain unlimited numbers of different Currency objects. This is a class contains every variable that a currency would have: exchange rate and currency name. Notice that all exchange rate here is written in exchange rate to US Dollar. However, this class is only half-generic since it contains variables like balance amount and loan amount because it's more like a currency sub-account instead of currency in real life. In this way, by taking more responsibility than it should, it makes all the upper classes like Account, User and Bank more generic.

To use a graph to represent these classes:



Zehui Jiang | U68975915 | zhjiang@bu.edu

For GUI classes, to be honest, this is the first time I try to write them in Java. And I'm not really sure about principles and rules I should obey when writing them. But I tried my best to make all these classes as clear as possible.
I can use a graph to present the structure of this part:



Zehui Jiang | U68975915 | zhjiang@bu.edu

I will not go into detail for all these classes because most of them is method-oriented instead of object-oriented. For method-oriented classes, they are more like GUI wrapped methods. Here I will focus on some design that I applied to make the GUI better.

1. Set the child frame location relative to its parent frame. In this case, even if the user moved the window, the new window will pop up right above the old window instead of pop up from one corner of the screen.
2. Set the parent frame disabled while child frame is enabled. This can prevent from user opening multiple duplicate windows and causing problems. As soon as the child window is disposed, the parent window will be enabled again. This design makes the system more robust.
3. Add strict restriction when it comes to taking parameter from input. For each text box input, I added very strict restrictions to make sure that no error would be caused by input error. And when it comes to selection from a list/array, use a combo box instead of textbox. It would eliminate any input risks. Without input errors, once the code is well-debugged and inner errors are eliminated, the program will be very robust and even errorless.

Thank you!
Zehui