Stoica Razvan Cosmin

Group 30424

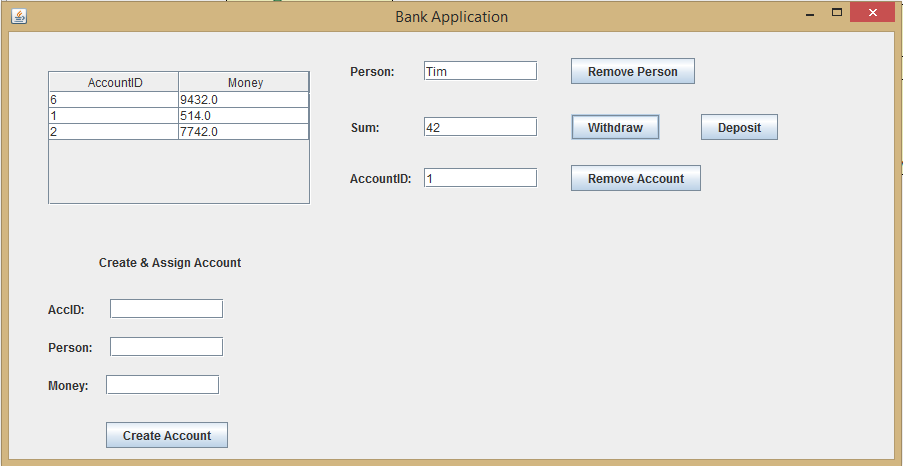
**Bank Application**

1. **Homework objective**

Design and implement a bank application implementing the observer design pattern. The application must manage operations on persons accounts (deposit, withdraw, add account, remove account).

1. **Problem analysis**

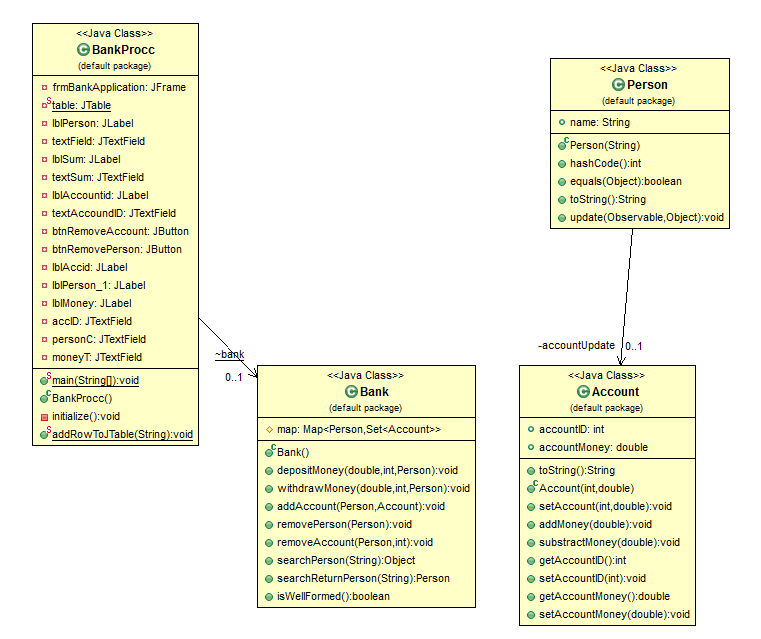
The program should be simple to use and have clear input/output display. It will load a number of accounts and people when is launched, but we can also create them from the user interface. We have a JTable in which we will display all the accounts to the main holder. We can withdraw and deposit very easy by introducing the values in a text layout.



1. **Design**

For the design of the project I’ve to tried settle on the given templated in the class. So I have 4 classes: Person, Account, Bank, Server and BankProcc.

.



UML DIAGRAM

The Person class implements the Observer, it has a string for the name with getters and setters, and a private instance of the Account which will be used in order to get the notification if an account is modified. Person class being the Observer. The constructor gets the name. We also override the hashCode(), equals and toString() methods. We also override the update method which will have as arguments an Observable object and an object, and here we will use the instance of the account (called accountUpdate) to get information if the account has been updated.

The Account class extends Observable. It has an accountID and accountMoney, both having getters and setters which call the methods setChanged() and notifyObservers() in order to tell the observer if there has been a change. The constructor initialize the accountID and the accountMoney. We also have the methods addMoney and subtractMoney which will do add/subtract money from the account, also both notify the observer. We also override the toString() method.

The Bank class has enabled assertions. So we have invariants and pre and post conditions for every method. We also have a Map<Person,Set<Account>>, which is a hash map with the key Person and the value a Set of Accounts. We also have the depositMoney merhod which has as arguments the sum of the money, the account ID and the person which we need to put the money. We also assert this method with the method isWellFormed() which is used to check the hash map for any errors. The withdrawMoney() method is the same as depositMoney but this time we subtract money. The addAccount method has as arguments a person an account and adds an account to the person. The removePerson() method removes a person from the hash map. The removeAccount() has as arguments a person and the accountID of that person, the method removes the specified account. The method searchPerson() has as argument a string which gets the name of a person and searches for that person in the hash map and returns the key if it find that person. The method searchReturnPerson() is the ame as searchPerson() but it returns a Person, not a key.

In the BankProcc class we have a static method addRowToJTable() which is used to populate the JTable with the accounts of a person. Here we have labels for person, sum, accountID etc. And textFields for everytime we need to call a function from the Bank class. We have the buttons Remove Person, Withdraw, Deposit, Remove Account. And a JTable included in a JScrollPane so we can scroll if there are more accounts than the table space in the user interface.

As for packages we have in the Account class java.util.ArrayList, java.util.Observable. In the Person class java.util.ArrayList, java.util.Observable, java.util.Observer. In the Bank class we have java.awt.List, java.io.Serializable, java.util.ArrayList, java.util.Collection, java.util.HashMap, java.util.HashSet, java.util.Iterator, java.util.Map, java.util.Set. And in the BankProcc class we have the usual swing and awt packages.

1. **Implementation and testing**

The operations work correctly if they are written in the expected format, if not the program will throw an exception (the user will see a pop-up message that will warn him/her if he did something wrong). I tested each class before using the user interface, and I tried to print every possible outcome to make sure that the application doesn’t break or throws unexpected exceptions.

1. **Results**

All the given operations work correctly and the application displays error messages in case the user inputs something wrong.

1. **Conclusion and further developments**

As a conclusion the project needs a lot of improvements in order to be optimal and bug free, and as for further developments a test class and the possibility of saving and loading information for the Bank would be a huge improvement.