Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

2nd Semester 2016-2017

**Programming Techniques**

**Assignment no. 4**

Student: Crișan Dan

Group: 30424

Supervising teacher: Viorica Chifu

**Table of contents**

1. **Objective** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. **Problem analysis, modeling, scenarios and use cases** . . . . . . . . . . . . . 3

2.1. Problem analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3

2.2. Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3

2.3. Scenarios and use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1. **Design** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .6

3.1. Design decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .6

3.2. UML diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

3.3. Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .6

3.4. Class design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .7

3.5. Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .7

3.6. GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .8

1. **Implementation and testing** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2. **Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3. **Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11
   1. Things I have learned . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11
   2. Future improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
4. **Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12
5. **Objective**

The objective of this project is none other than creating a functional bank, of course of a much smaller complexity than a real one.

1. **Problem analysis, modeling, scenarios and use cases**
   1. Problem analysis

The program should provide a user-friendly interface so that a user could do the following operations:

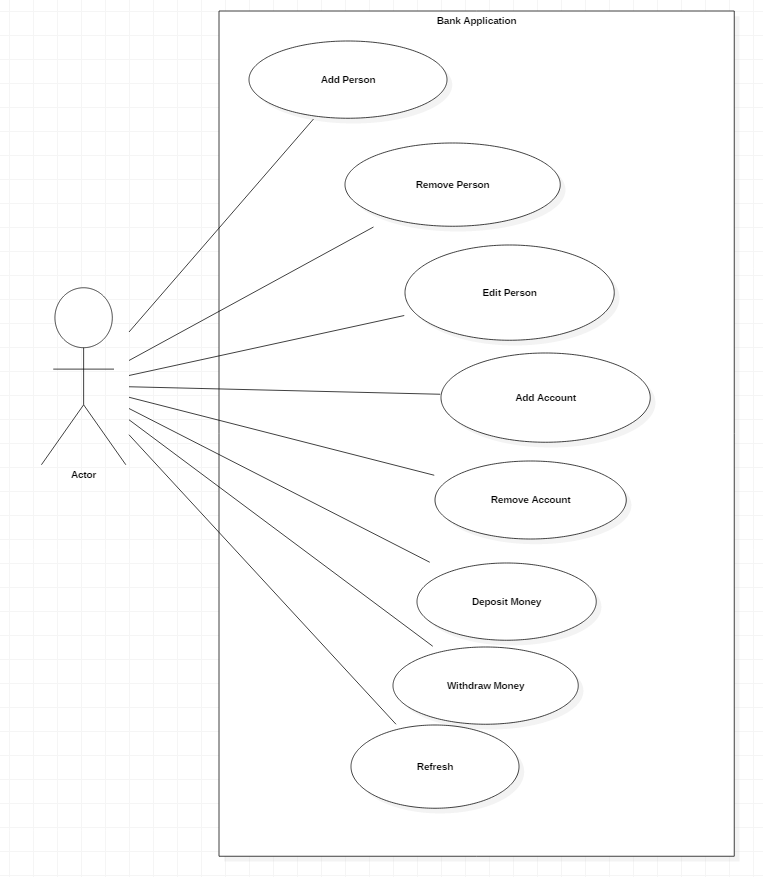
* Add a new person to the bank’s database
* Edit a person’s name or address in the bank’s database
* Remove a person from the bank database
* Add a new account for a selected person
* Delete an account from the bank’s database
* Deposit money in a specified account
* Withdraw money from a specific account

We should not forget that for an account that is a Savings Account, one can make only one deposit or withdrawal during its existence.

* 1. Modeling

The approach I have chosen follows the diagram provided in the laboratory papers. I have a Bank class which implements an interface. A Bank can have multiple persons and each person can have more accounts. The accounts can also be of type Spending and of type Saving. To and from a Savings Account, a person cannot make more than one deposit / withdrawal. Another difference between the account types is that a Savings Account also computes interest over a period of time given by the user.

* 1. Scenarios and use cases



The main (and only) actor in this program is the user. The user can perform any of the actions mentioned above, considering the input is valid. Interaction between the user and the system is done via the GUI, by typing in data for a person or a account, or data required for operations made on an account. Then the user proceeds by pressing the button specific for the desired operation.

There are several use cases we should mention:

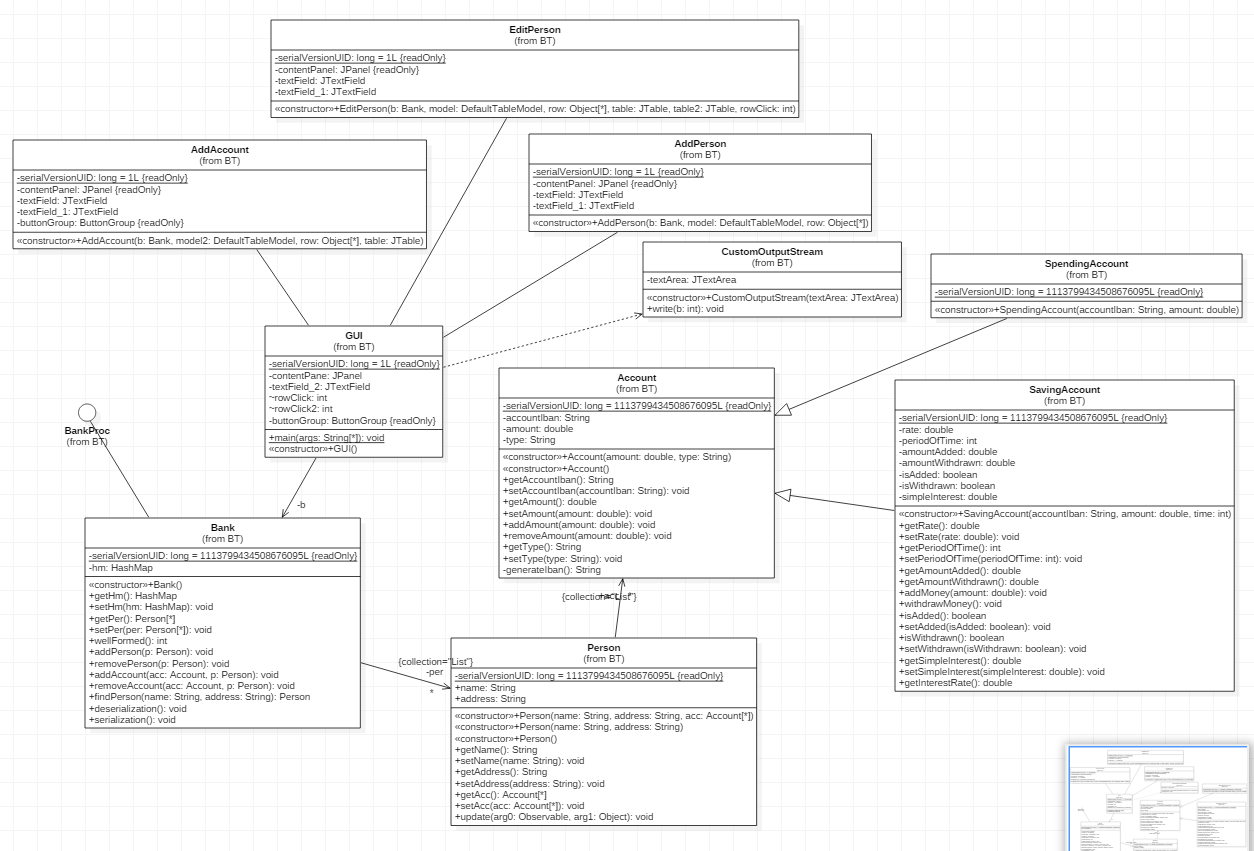
* It is important to specify first that, in order to view data from previous program executions or to be sure that data that the user just changed has taken place in the GUI, the user can use the Refresh button at any time in order to assure everything is up to date.
* Adding a person: After pressing the Add Person button a pop-up window will appear. The user should insert data in the Name: and Address: fields. All other fields can be left empty.
* Editing a person: Just like pressing the Add Person button, after pressing the Edit Person one, a pop-up window will appear. However before pressing the edit button, the user has to click and choose one of the persons from the table which he/she desires to edit. The user should then insert data in the Name: and Address: fields (not necessary both). All other fields can be left empty. The user can see that all accounts have changed their holder according to the newly edited person.
* Removing a person: All the user has to do is select from the table a person by clicking on it and press the Remove Person button. Upon pressing the Refresh button all accounts related to that person will be deleted.
* Adding an account: First, the user should select a person for which the account to be made, by clicking in the persons table. Then upon pressing the Add account button a pop-up window will appear. The user should then insert the desired amount to be deposited into the newly created account and the type of the account (choosing between Savings or Spendings account). If the desired account is a Savings Account, the user should also input a time upon which he/she desires to withdraw the money, so that when the withdrawal is made the withdrawn sum is increased according to the interest set by default in the application to be 5%.
* Removing an account: Just like removing a person, all the user has to do is click on an account and press remove.
* Deposit Money: The user will now type the desired amount to be deposited in the main window. Then after selecting the account in which the sum to be added, the button can be pressed and the action will take place immediately.
* Withdraw Money: There are 2 cases here. If the account is a Spending Account the user should insert the amount desired to be withdrawn from the account, then click on the account from which the sum to be extracted and press the Withdraw Money button. Messages regarding lack of money in the account or account empty will be printed if necessary as well as a message specifying the sum withdrawn each time. If the account is a Savings Account, the amount field can be left empty, since there can be only one withdrawal from a Savings Account. Therefor all money will be extracted together with an interest computed according to the time set at the creation of the account.

1. **Design**
   1. Design decisions

As requested in the assignment, I decided to have the classes Bank, BankProc, Person, Account, SavingsAccount and SpendingAccount as well as other classes I considered necessary such as: GUI for the graphical user interface, AddAccount, AddPerson and EditPerson for the additional pop-up windows and CustomOutputStream, a class which I used in other projects before to redirect the console output to a textArea in the GUI.

* 1. UML class diagram (Unified Modeling Language)

In the following picture, we can see the diagram for the classes found in my program. The diagrams were made using StarUML.



* 1. Data Structures

As data structures I use an ArrayList of persons and a HashMap<String, Person> which has as key a string (the name of a person) and as value the person with the name equal to my string, in my class Bank. Aside from these structures, each person has an ArrayList of accounts, in the class Person. The HashMap is my most important structure, since mainly this is the one that holds all data together and lets me access it in order to print data. Also here is the place all inserted data goes to first, whether I want to add a person, edit one, remove one, add an account or remove one.

* 1. Class design
* Class Bank: All this class has are the data structures mentioned in the section above: a HashMap<String,Person> and an ArrayList<Person>
* Class Account: This class has 2 strings, the IBAN and the type of the account and a double in which the account stores the current amount.
* Class SavingsAccount: Aside from what this class receives from the class Account, here I also have a rate to compute the interest (set at 5%), and int representing the periodOfTime over which the interest is computed, 2 doubles, amountWithdrawn and amountAdded, which keep track of the balance in the account, and 2 booleans which remember whether the savings account has already performed a deposit or a withdrawal respectively.
* Class SpendingAccount: Has nothing more than what it inherits from the Account class.
* Class Person: This class has 2 Strings which hold the name and the address of the person and an ArrayList<Account> which holds a list of the accounts of that person.
* Class GUI and classes related to the interface(AddAccount, AddPerson, EditPerson): aside from the textFields, textArea, buttons and other interface related objects, this time I have implemented here a click listener, accessed through the int variable rowClick.
  1. Algorithms

Since most classes do not have any complicated algorithms, only getters and setters I will not group the algorithms on classes.

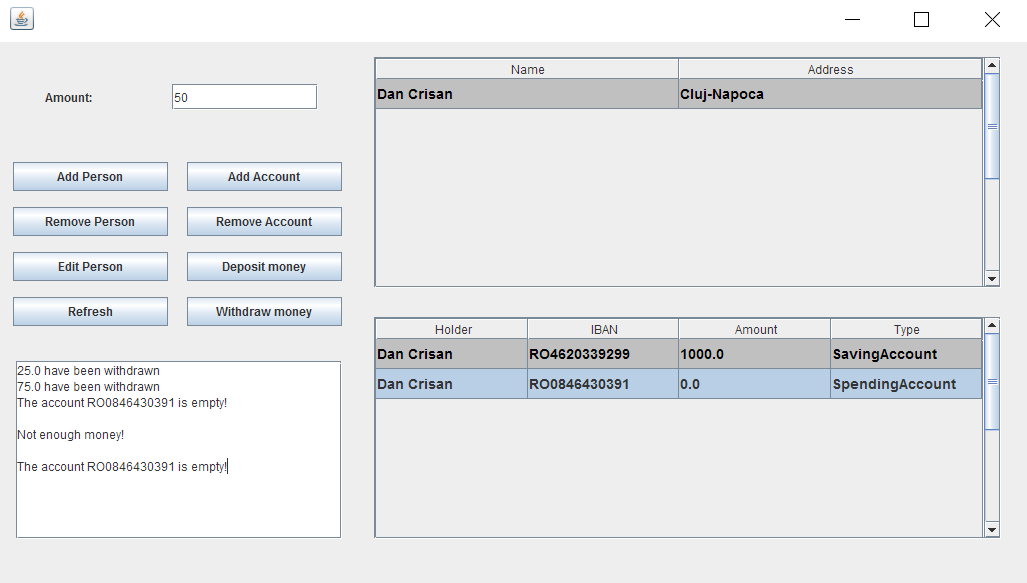
First of the few algorithms I have are in the class Bank. The algorithms I’m talking about are methods which add or remove accounts or persons. They follow mostly the same structure. Using assert I have a few pre-conditions before adding or removing the person/account from the list/HashMap, followed by a few post-conditions. There is also a method called findPerson, which takes a name and an address from my table and searches in the HashMap for that person, in order to say, retrieve its accounts.

* 1. GUI (Graphical User Interface)

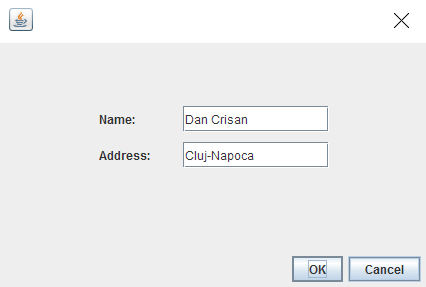
The graphical user interface I have proposed is clearly labeled although not that easy to use if the user does not read the details specified in the use case section. That being said, right off the start, the interface has 2 big table showing the persons in the table above and the accounts in the table below. In the left side of the interface the user can see a text box with the label Amount: which is used to specify the amount of money he or she wants to deposit or withdraw from the account clicked. Below this text box there are several buttons very clearly labeled for adding or removing persons or accounts, for depositing or withdrawing money and for refreshing the tables. Under the buttons there is a text area which prints any information that is account related such as: “Deposit has been made!”.

Pressing the buttons Add Person, Edit Person or Add Account will pop a new dialog box requesting for additional information. Pressing the ok buttons will complete the action, while pressing cancel or closing the dialog box will ignore anything written in the text boxes. The user must remember to click on a person before editing or adding a new account to it.

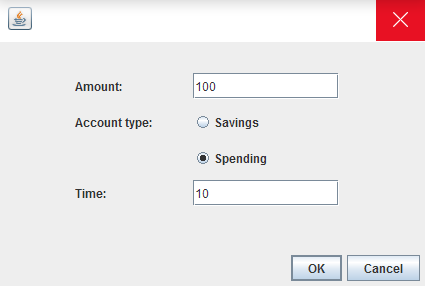
Here is what the interface looks like:



This is the pop-up dialog box for the Add Person and Edit Person buttons:



And here is the pop-up dialog box for the Add Account button:



1. **Implementation and testing**

The main methods which make my program do the specified operations are written as this:

The method for adding a person:

**public** **void** **addPerson**(**Person** p) {

**assert** p!=**null**;

**assert** !hm.containsValue(p) : "Person already exists!";

**assert** !per.contains(p) : "Person already exists!";

hm.put(p.getName(), p);

per.add(p);

**assert** per.contains(p) : "Person hasn't been added!";

**assert** hm.containsValue(p) : "Person hasn't been added!";

}

The method for removing a person:

**public** **void** **removePerson**(**Person** p) {

**assert** p!=**null**;

**assert** hm.containsValue(p) : "Person does not exist!";

hm.remove(p.getName());

per.remove(p);

**assert** !per.contains(p) : "Person hasn't been removed!";

}

The method for adding an account:

**public** **void** **addAccount**(**Account** acc, **Person** p) {

**assert** p!=**null**;

**assert** acc!=**null**;

**assert** hm.containsValue(p) : "Person does not exist!";

**assert** !p.getAcc().contains(acc) : "Account already exists!";

p.getAcc().add(acc);

acc.addObserver(p);

**assert** p.getAcc().contains(acc) : "Account hasn't been added!";

}

The method for removing an account:

**public** **void** **removeAccount**(**Account** acc, **Person** p) {

**assert** p!=**null**;

**assert** acc!=**null**;

**assert** hm.containsValue(p) : "Person does not exist!";

**assert** p.getAcc().contains(acc) : "Account does not exist!";

p.getAcc().remove(acc);

acc.deleteObserver(p);

**assert** !p.getAcc().contains(acc) : "Account hasn't been removed!";

}

The method for finding a person:

**public** **Person** **findPerson**(**String** name, **String** address) {

**for** (**String** **p** : hm.keySet()) {

**if** (p.equals(name) && hm.get(p).getAddress().equals(address))

**return** hm.get(p);

}

**return** **null**;

}

The methods which do the serialization and deserialization:

**public** **void** **deserialization**() {

**try** {

**FileInputStream** **fileIn** = **new** FileInputStream("bankSer.ser");

**ObjectInputStream** **in** = **new** ObjectInputStream(fileIn);

hm = ((**HashMap**<String, Person>) in.readObject());

**this**.setHm(hm);

in.close();

fileIn.close();

} **catch**(**IOException** **i**) {

i.printStackTrace();

} **catch**(**ClassNotFoundException** **c**) {

**System**.***out***.println("Bank class not found");

c.printStackTrace();

**return**;

}

}

**public** **void** **serialization**() {

**try** {

**FileOutputStream** **fileOut** =

**new** FileOutputStream("bankSer.ser");

**ObjectOutputStream** **out** = **new** ObjectOutputStream(fileOut);

out.writeObject(hm);

out.close();

fileOut.close();

} **catch**(**IOException** **i**) {

i.printStackTrace();

}

}

The method for calculating interest in the class SavingAccount:

**public** **double** **getInterestRate**() {

**double** **interestRate** = **this**.getRate()\***this**.periodOfTime\***super**.getAmount();

**this**.setSimpleInterest(interestRate);

**return** interestRate;

}

1. **Results**

The program resulted is a good prototype to start off, for creating a bank with full functionalities. What makes it great in my opinion is mostly the serialization which helps the program remember data through different runs.

1. **Conclusions**
   1. Things I have learned

During this project I got to learn how to work with HashMaps which I think have a great utility and I might use them again in future projects. Also as I look back I realize some other projects I have made throughout the years may have been easier using a HashMap. Moreover, I found serialization quite an interesting and useful thing considering that in most applications users don’t want to lose their data once they close the program. Having said this, I really think this was a project with great applicability in real life, especially thinking that with a little bit of polishing this is in my opinion quite close (considering my current programming knowledge) to what a bank needs as a software for its clients or for its employees.

* 1. Future improvements

One of many improvements this program could benefit would be so that the user could transfer money from one account to another. A more realistic thing to add would be for the time variable which, I used to compute interest to be an actual data and for the user to be “punished” if he/she would withdraw money before the date specified. Spending account should also have a small interest in order for the bank to be more realistic and Saving accounts should let you deposit money more than one time and should let you withdraw money more than one time although with the cost of a decrease in the persons interest on that account.

1. **Bibliography**

In order to redirect the output from the console to my text area in the Graphical User Interface, I used the class CustomOutputStream and a few other commands found on the following website:

<http://www.codejava.net/java-se/swing/redirect-standard-output-streams-to-jtextarea>

For a small insight on what HashMaps presume, I took a look on the link provided by the university:

<http://coned.utcluj.ro/~salomie/PT_Lic/3_Lab/HW4_Tema4/HW4_Indications_Hashing>.pdf