**Technical University of Cluj-Napoca**

**Faculty of Automation and Computer Science**

**Computer Science Department**

**Programming Techniques**

**Fourth Homework: Bank Application**

**Mihoc Nicolae**

**Group: 30425**

Table of Contents

*1. Problem Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3*

*1.1 Problem Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3*

*2. Diagram Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4*

*2.1 Use Case Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4*

*2.2 Class Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5*

*2.3 Sequence Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6*

*2.4 Activity Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7*

*3. Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8*

*3.1 Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8*

*3.2 Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8*

*4. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9*

*5. Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12*

*6. Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12*

*7. Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13*

**1. Problem Specification**

1. Define the interface BankProc (add/remove persons, add/remove holder associated accounts, read/write accounts data, report generators, etc). Specify the pre and post conditions for the interface methods.

2. Define and implement the classes Person, Account, SavingAccount and SpendingAccount. Other classes may be added as needed (give reasons for the new added classes).

3. An Observer DP will be defined and implemented. It will notify the account main holder about any account related operation.

4. Implement the class Bank using a predefined collection which uses a hashtable. The hashtable key will be generated based on the account main holder (ro. titularul contului). A person may act as main holder for many accounts. Use JTable to display Bank related information.

4.1 Define a method of type “well formed” for the class Bank.

4.2 Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions).

5. Implement a test driver for the system. 6. The account data for populating the Bank object will be loaded/saved from/to a file.

**1.1 Analysis of the Problem**

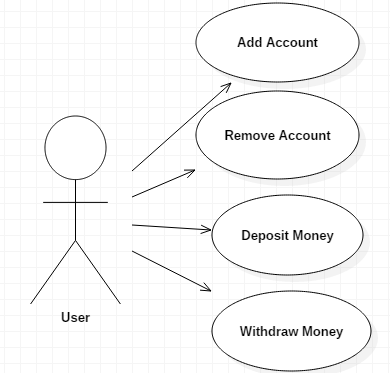
We are required to implement a bank application which resambles the behaviour of a real world bank. In a real world bank the customers can deposit money, can withdraw money and can have 1 or many accounts. In our problem specification each client can have a total of 2 accounts. Another thing that is required is to implement an Observer DP which resambles the real world messages that people get on their phones when they have a bank application which sends them messagges when transactions are done. Each client of the bank will have an id which will be unique and stored in a hashtable.

**2. Diagram Implementation**

For the purpose of better understanding how to resolve this problem first we need to draw the neccessary diagrams. These diagrams will help us to understand how to approach the final solution. We need multiple diagrams because each one shows a different angle related to the problem.

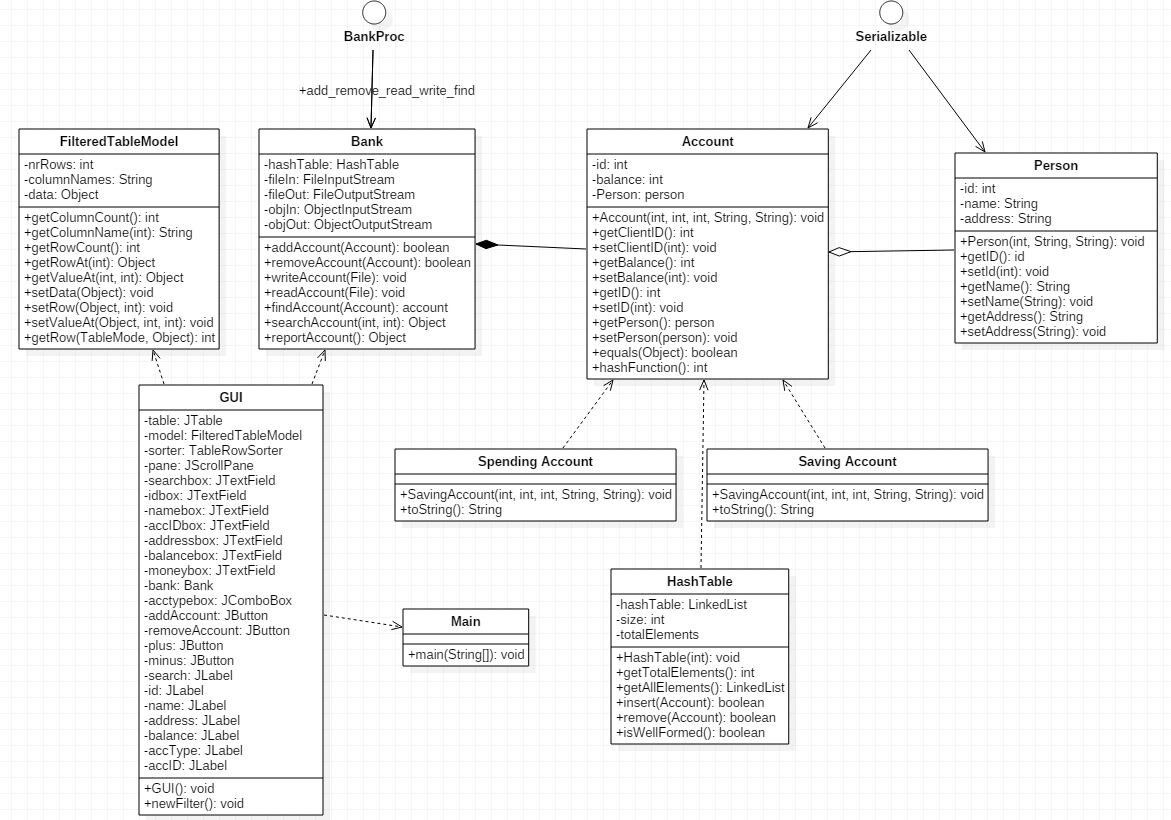
**2.1 Use Case Diagram**

The first diagram is the use case diagram. This diagram is the simplest one because it shows only the available options of the application for the user and it doesn’t approach any details of the implementation. It is useful beacause it shows the final possible actions that can be used by the user.



**2.2 Class Diagram**

The second diagram is the class diagram. It is the most important diagram and the most complex one because it captures the attributes and the methods of every class. Another important thing about this type of diagram is that it shows also the connection made between this classes like aggregation relations, composition relations etc.

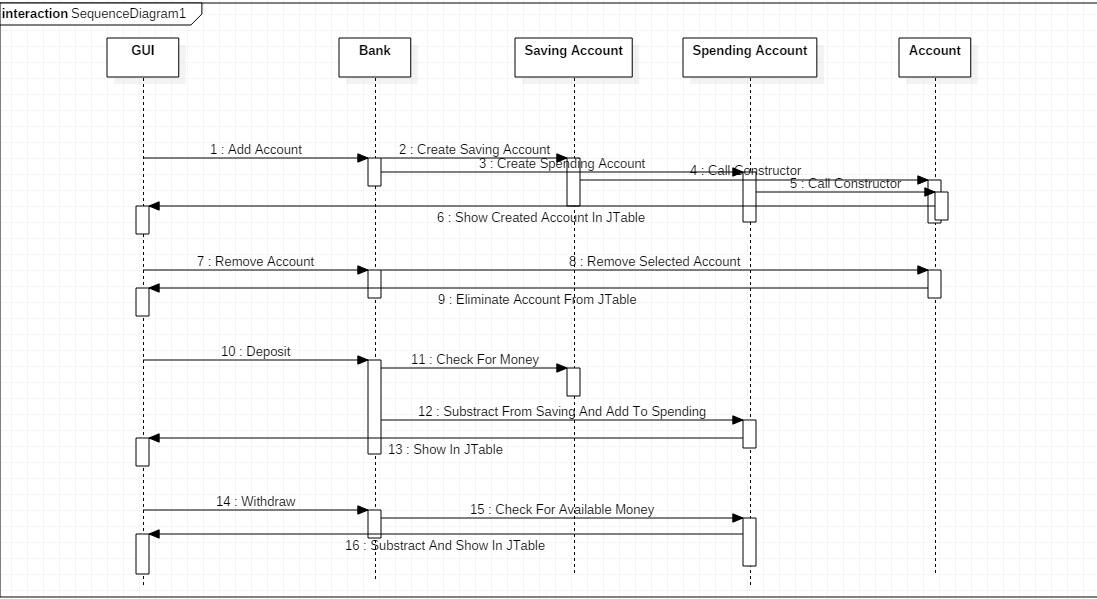


As we can see from the above diagram there are quite a few relations

in this bank application. For starters the most important class is the Bank class because it contains the most used methods which are based on the Account class hence the composition relation. Another important relation forms between Person class and Account class. The Account class is based on this Person class and that is logic because the owner of any account is a person. The aggregation relation between those 2 classes are exists because without a person an account cannot be created.

**2.3 Sequence Diagram**

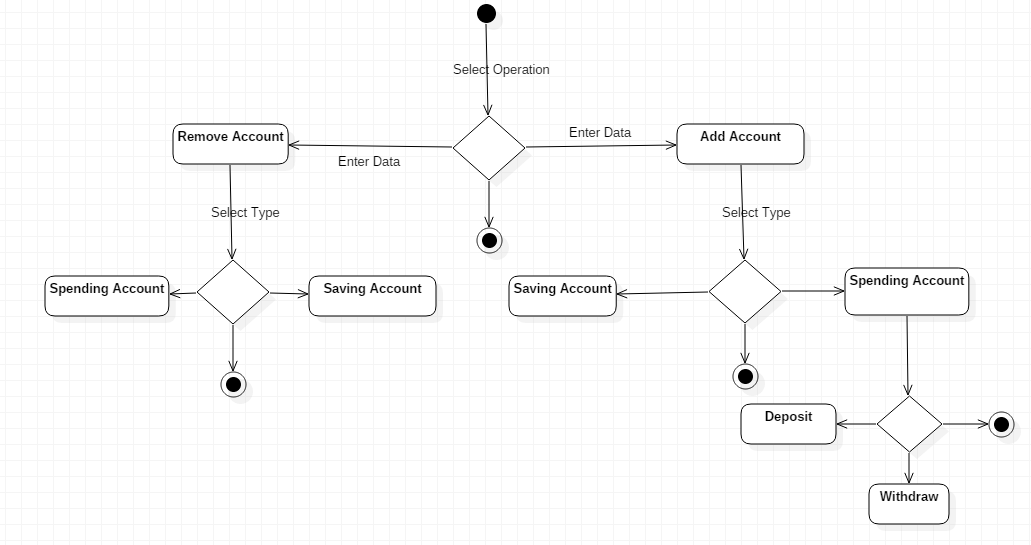
This diagram has the purpose of showing what happens exactly when we give the application a command. It displays the whole process which happens in the background based on what we coded starting from the moment we give the command to the moment the changes are displayed in the graphical user interface.



Each of these operations are described in detail from the moment the command is given, the classes it “activates” and returning the result on the table in the graphical user interface.

**2.4 Activity Diagram**

The final diagram is the activity diagram. This diagram shows us the little decisions that we must make in the process. It is useful because it shows at each step the available options for a task that we gave.



As we can see from the moment the app is initialized we have our first decision. We must choose if we want to add a new account or remove an account. If we want to remove an account we are presented another decision and that is to delete a spending account or a saving account.

The second option to add an account shows us that is necessary to decide the type of the account we want to open. Also if we have an available spending account which has money in it we can deposit more or we can withdraw any available money.

At each step if the user decides to quit he can do so without a problem so the final state can be anywhere between these decisions.

**3. Design**

After the presentation of the diagrams it is time to think about a strategy to design our application in code. It is important to decide the minimum and enough number of classes that we need and also the packages to store these classes. The problem specifies that we must implement the following classes from the beggining: Person, Account, SavingAccount and SpendingAccount. Also this time we must implement an interface called BankProc.

**3.1 Packages**

For this implementation I decided that 5 packages should suffice. I split the classes in the following packages:

-Bank: this package will hold the class Bank and also the interface BankProc.

-Management: in this package I placed all the classes that had a role in the arrangement of the persons to the corresponding accounts: Account, HashTable, Person, SavingAccount, SpendingAccount

-View: this package is designed to hold the graphical user interface class GUI and the class FilteredTableModel for the JTable.

-TestUnit: this package holds the class called JunitTest used for testing the implementation

-MainController: in this package resides the Main class

**3.2 Classes**

The total number of classes that I considered to be neccesary for this implementation is 11. In order to develop this bank application I needed a few things. The first thing that I implemented was the Person class and that is because the others won’t work without humans. In this class I “constructed” a person with some details. The next thing was the Account class which is made for the bank having a person’s attributes and the characteristic things of a bank account like an ID and balance. Because it needed to be “well formed” this bank contains a class called HashTable which deals with the correct “placement” of the accounts in the bank. In the bank class are the main methods which are developed from the interface BankProc. Finally to display all these to anybody a graphical user interface class is needed and also a class to deal with the display of the accounts and persons in that GUI.

After this implementation is done the problem requires of us to also have a “test” class to check the correct functionality of the methods implemented for the bank.

**4. Implementation**

After we disscused the classes and packages that we need to get the job done we must talk about the actual implementation of all this.

*Person Class*

This class was created with the aim to have a few characteristics about people in general. In this implementation my humans have an ID, a name and an address. Besides the constructor which does this we need the getters and setters for all the characteristics. Also, the interface Serializable is needed to help us remember all the details of the person after we change them in the bank or after we open a new account for example.

*Account Class*

In the Account class the focus is on the main characteristics about a bank account. Each bank account must have an unique ID, a client ID, the amount of money for each client and also the personal details of each human. All of these are created with the help of the constructor and the getters and setters. Besides the methods mentioned before we also need 2 additional ones. The first one is the equals method which helps us by checking the equality of 2 accounts based on the client ID and the account ID from our application and the second method is a method for the hash function which places the account in the coressponding position in the hash table. All of the accounts must be kept at any time so this class implements the Serializable interface which helps us to do that each time details are modified of removed the changes are saved for the next time.

*SavingAccount Class*

This class is an extends the Account class and its only role is to call the constructor of that class in order to create an account of type Saving.

*SpendingAccount Class*

Another class like the one mentioned before this is the SpendingAccount class which extends Account class and once again calls that constructor to create an account of type Spending.

*HashTable Class*

This class is an important one because it deals with the placement of the accounts in the bank based on a hash function from the Account class which generates an unique ID saved in the correct spot in the application. It contains a hashtable formed with linked lists. It has the method insert which adds the new account based on the index of it in the hash table. A remove method to delete the correct account from the table and a “wellFormed” method to check if the number of elements found in the hash table is equal to the actual number of the elements stored and the correct position of each index in the hash table.

*BankProc Interface*

A requirement of this homework was to implement the mentioned interface which has the main methods of this bank application like add account, remove account, write account etc. Also before each method some pre conditions and post conditions were introduced with comments which will be later implemented in the Bank class.

*Bank Class*

The class bank is responsible with the behaviour of application and is using the hash table previous implemented. Also a requirement was that this class would implement the BankProc interface and all the pre and post conditions to be introduced using “assert” command and also the “wellFormedMethod()” from the HashTable class to be verified. Another interface that must be implemented is the Serializable interface because the data must be saved every time we have changes. Besides the “writeFile” and “readFile” method used for Serialization I also implemented a method for searching for a specific account and one method to print the details from the hashTable.

*GUI class*

To display all the implementations above a graphical user interface is required. This class contains all the labels and boxes and the table needed for introducing the account data. In the constructor of this class I initialize all the elements needed for the window. Also, each method mentioned above has a button which when pressed activates the action listeners implemented on the buttons. Another thing that is important to mention is the deserialization of the bank accounts and the placement in the JTable. For the ease of introducing data I placed a method inside this constructor that automatically fills the boxes when pressed.

*FilteredTableModel class*

I considered necessary to implement a different class to deal with the JTable and there I implemented all the methods I considered to be necessary. This class extends the AbstractTableModel and thus I have access to the methods of this model. This class is useful because I don’t need to fill the GUI class with all the table method implementations. The most important methods from this class are: “setRow” to create a row with desired data, “setValueAt” to easely change the value in a column in a table and a “getRow” method to go directly to the needed account or client.

*JunitTest class*

Another requirement for this bank application is to implement a “test bank” which contains the methods already implemented tested on a separate instance besides the bank app itself. In this class I tested the hash function, the add method, the remove method, the wellFormed method and the findAccount method an they were all successfull.

*Main Class*

The final class is the class that makes all this work. It is the main class and its only job is to launch the graphical user interface.

**5. Improvements**

The first thing that needs to be mentioned in this section is that I haven’t implemented the Observer design pattern so this will be the first improvement. Another improvement that comes to my mind is the implementation of a separate GUI for each person in which all the available data will be displayed like a real world online bank application where users can pay bills an check their money and so on.

Another thing that could be mentioned is the implementation of a system which could send money to a different account and a different person like Western Union.

**6. Conclusions**

This homework introduced some new elements like the design pattern or the hash table method to save the accounts and that was new for me. It was not that hard but I was not used to work with pre and post conditions or assert or this Observer design pattern. Also another new things was the Junit testing mechanism which I haven’t heard before but is useful for testing the implemented methods for errors.

Once again this bank application is only a small implementation of a really big system that is rulling the world right now and a mere attemp to learn how to build a futher application with security, all the already existing features and maybe more awesome new features.

**7. Bibliography**

**1.** [*http://www.tutorialspoint.com/junit/junit\_writing\_tests.htm*](http://www.tutorialspoint.com/junit/junit_writing_tests.htm) *-* for the JUnit test implementation

**2.** [***http://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html***](http://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html)

- for the HashTable implementation

**3.***http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html*

-for understanding the “assert” method

. . . . . . . . . . . .. . . . . .. .. .. . . . . . . . . . . . . . . ... .. . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . .