

Technical University of
Cluj-Napoca
Faculty of Automation
and
Computer Science

Bank Management System

Discipline: Programming Techniques

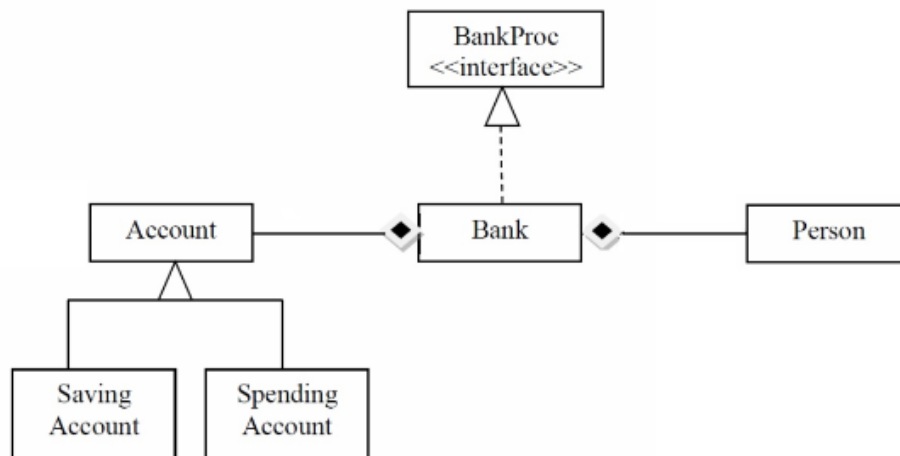
Date: 14.05.2018

Gavril Luca

Group:30422

1.Main objective:

The objective of this homework is to design and implement a banking management system, by processing clients and their respective accounts, following the next schema:



In order to store data about the clients and their respective accounts, we are going to use HashMaps.

It is desired that the project be organized in packages, holding different classes that accomplish multiple roles. In order to implement the system, it will be decomposed into a multiple phase process:

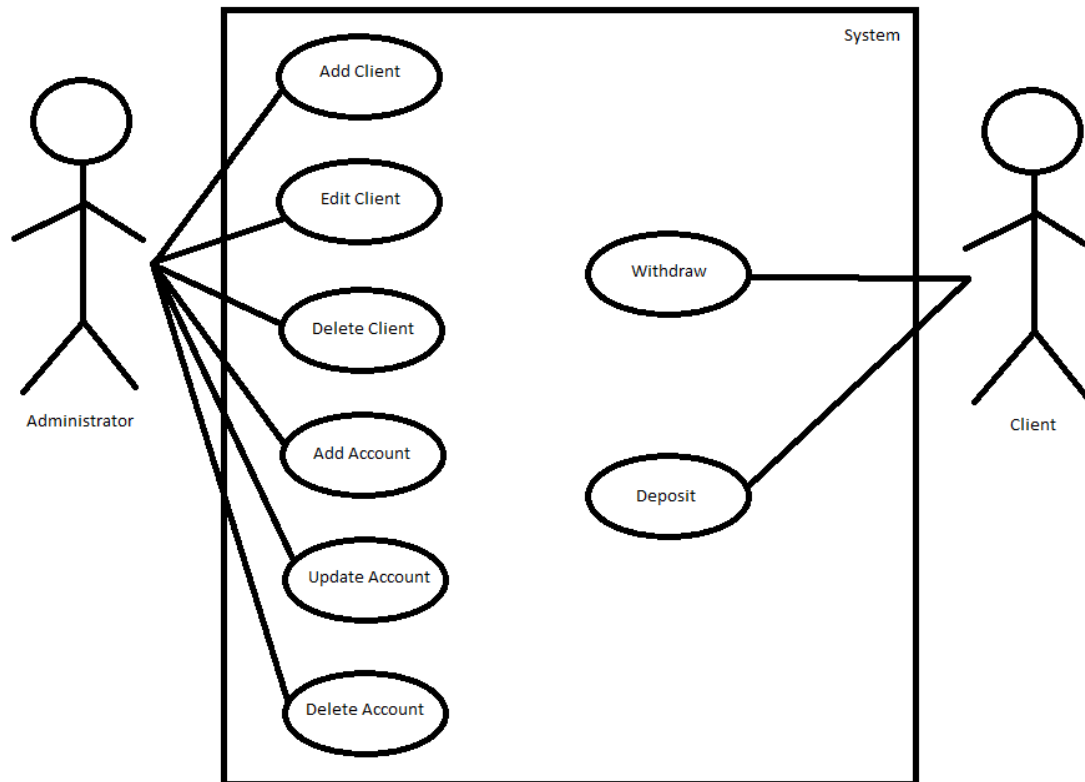
- **Client operations:** The administrator can add, update and delete clients from the bank.
- **Account operations:** The administrator can add, update and delete accounts from the bank.
- **User operations :** Each client is able to withdraw or deposit a sum of money in any of his accounts.
- **Graphical Interface:** The system is simultaneously designed for an administrator who is in charge of clients and accounts, as well as for each of the clients that wish to make transactions in any of their accounts.
- **Design by Contract:** The bank's way of handling clients and accounts is done based on the Design by Contract Programming Techniques.

2. Analysis:

Looking at the scheme, it is obvious that there must be a connection between the clients and their accounts, through the intermediate that is the bank.

The graphical user interface also contains the ability to show the administrator the accounts and clients tables, automatically updated for him.

2.1. Use case diagram:



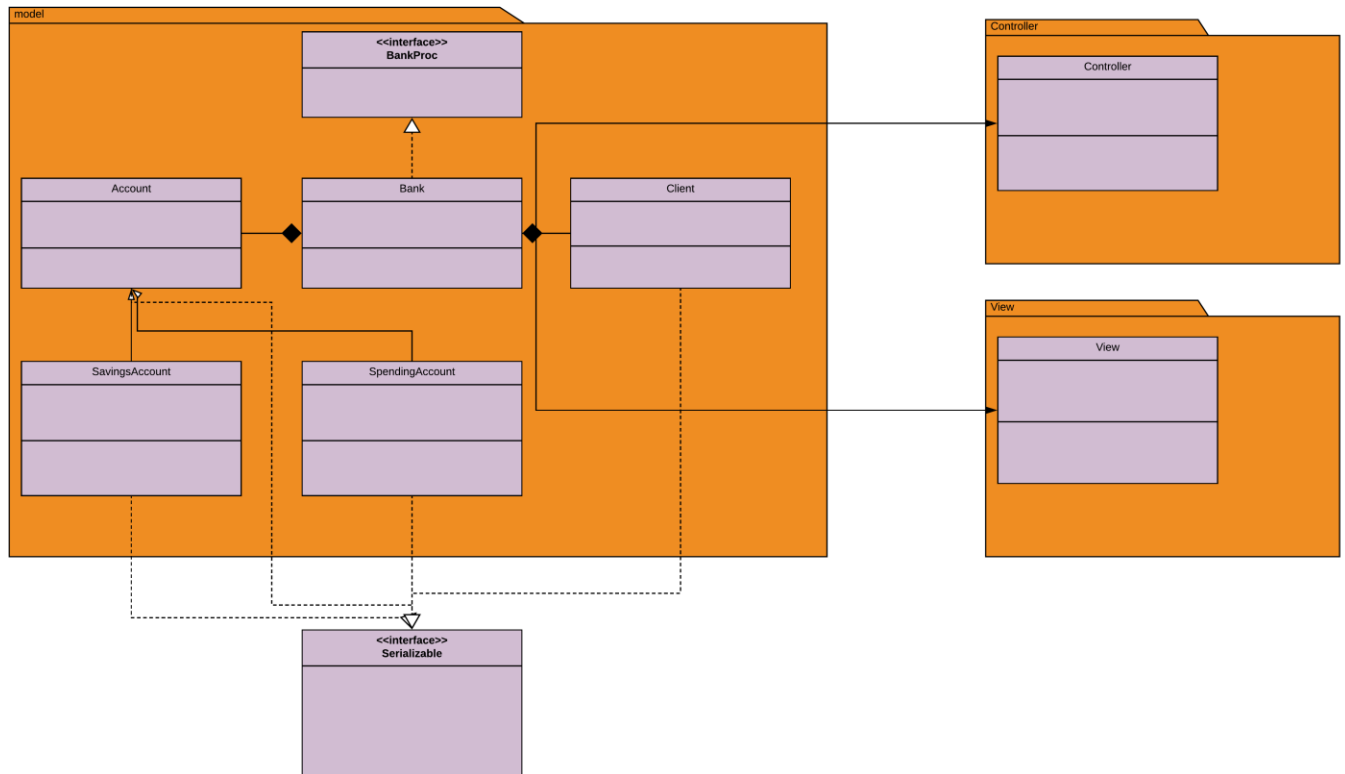
Again, the system is designed for both an administrator in charge of client and account operations: adding a new client or account, deleting a client or account, updating a client or account, as well as for the typical client who can withdraw or deposit money in his accounts.

2.2. Assumptions:

It is fair to assume that the administrator is capable of working his way around the operations for the clients and accounts. Also, the general client is considered to be capable of understanding the difference between a savings account and a spending account, as well as being capable of withdrawing and depositing an amount of money in his accounts.

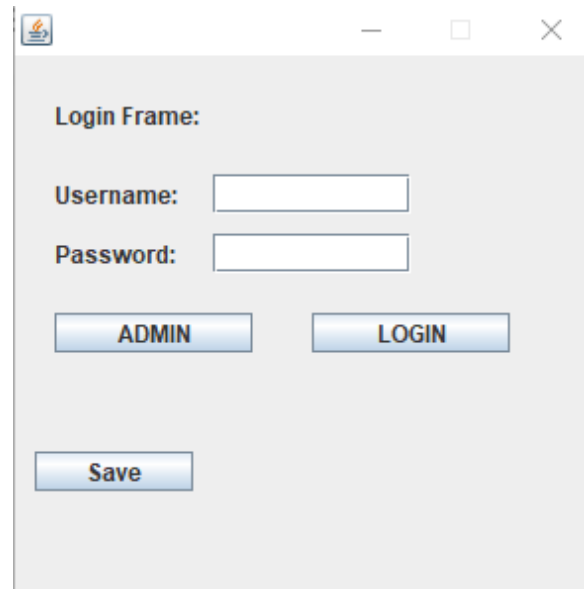
2.3.Design:

2.3.1.UML Diagram:



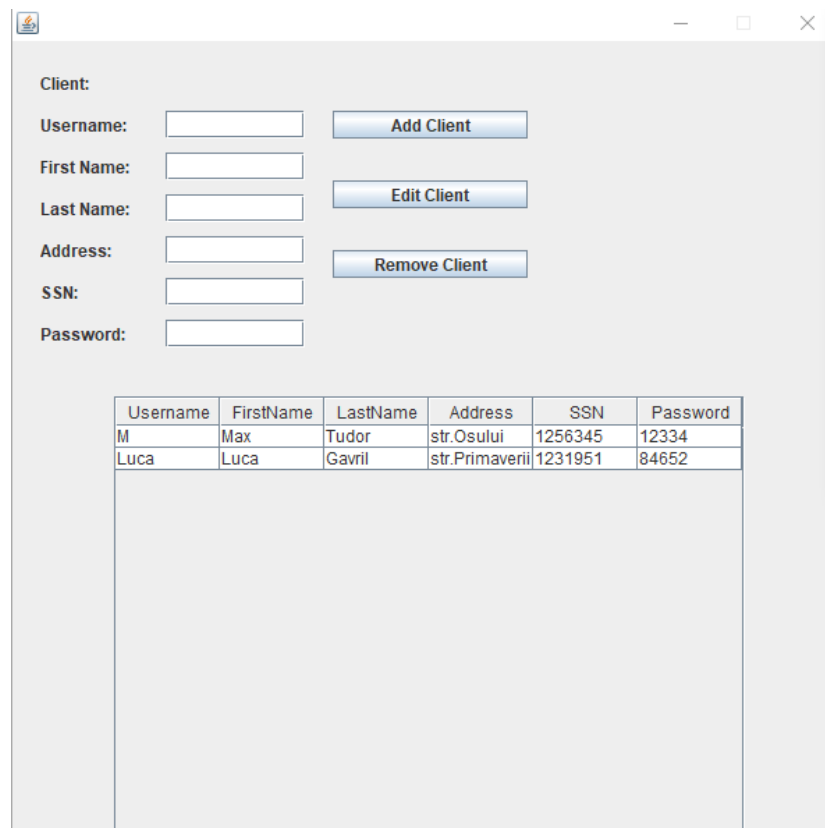
As it can be seen from the UML Diagram , the system is split in multiple packages, each holding a specific purpose in the implementation of the program. The system is built under the MVC model, starting from the original schema that was implemented by the system designers. The view package contains the GUI class, which is abundant for this specific system. The model package contains the original schema given by the supervisor, with the Bank, Account, Client classes. The controller package logically connects the model and view packages, holding together the system.

2.3.2. Graphical User Interface:



A screenshot of a 'Login Frame' window. It features a title bar with a small icon and standard window controls. The main area is light gray and contains the following elements: a label 'Login Frame:' at the top; a 'Username:' label followed by a text input field; a 'Password:' label followed by a text input field; two buttons labeled 'ADMIN' and 'LOGIN' positioned side-by-side; and a 'Save' button at the bottom left.

The Graphical User Interface firstly presents a log in frame, where the admin and client can differentiate between each other. In properly logged in, the administrator will be faced with 2 new frames, one controlling client operations and the other controlling account operations.



A screenshot of a 'Client' management window. It has a title bar with a small icon and standard window controls. The main area is light gray and contains: a 'Client:' label; a form with labels 'Username:', 'First Name:', 'Last Name:', 'Address:', 'SSN:', and 'Password:' each followed by a text input field; three buttons labeled 'Add Client', 'Edit Client', and 'Remove Client' arranged vertically to the right of the input fields; and a table at the bottom with 6 columns: 'Username', 'FirstName', 'LastName', 'Address', 'SSN', and 'Password'. The table contains two data rows.

Username	FirstName	LastName	Address	SSN	Password
M	Max	Tudor	str.Osului	1256345	12334
Luca	Luca	Gavril	str.Primaverii	1231951	84652

Account:

AccountID:

Client:

Sum account:

TypeofAcc:

InterestRate:

AccountID	Client	Sum	Type Of Account
1	Luca	214.0	Savings
1	M	2143.0	Savings

Each of these 2 frames contain 3 operations : add client, update client, delete client and respectively add account, update account and remove account. Each frame contains numerous labels for the clients and accounts, as well as text fields where the admin can insert data that will be further transmitted. Also, in both these frames there exists a panel holding a JTable for the clients, respectively for the accounts.

If the client logs in with the right credentials, he will be met by a new frame holding options for withdrawing and depositing into his accounts, a certain amount of money. The client has 4 text fields where he can input data, sum and accountId, for each of the two operations.

A Java Swing window with a light gray background and a blue border. It contains two sections: 'Deposit' and 'Withdraw'. Each section has two text labels, 'AccountID:' and 'Sum to deposit:' (or 'Sum to withdraw:'), followed by empty text input fields. To the right of the input fields are buttons labeled 'Deposit' and 'Withdraw' respectively. The window has standard OS window controls (minimize, maximize, close) in the top right corner.

The log in frame also has a JButton “Save” that is used for saving the data in the bank through serialization.

3.Implementation:

As previously mentioned, the system is split in packages, each holding one or more classes.

- controller package:

The controller package contains one class: Controller. Inside the Controller class there exist two different attributes : view and bank.

Methods wise, the class only holds a constructor: Controller(). Inside the constructor, we initialize the view attribute as well as the bank attribute. We call the serialization method for the bank attribute and then we follow with the ActionListener methods calls. Such a method is called from the view attribute for each button in the system’s frames.

Inside the respective method calls, the Controller constructor calls each needed bank method that should give a result when a button is pressed.

- view package:

The view package contains one class : View. Attributes wise, the View class holds numerous attributes, each with their own getter and setter methods.

The main method in the View class is again the constructor, where we take each attribute of the class, as well as a number of labels that we instantiated inside the constructor, and we add them with proper placement inside the frame of the Graphical User Interface where they are requested. For each frame in the Graphical User Interface, there are init() methods that will later be called in the Controller constructor method. Moreover, the View class contains ActionListener methods for every one of the JButtons that are required in the GUI. In order to create the JTables, we use DefaultTableModel attributes. For each JTable, we create and add a panel in the respective frames. The panel is then added the two table models, in order to display the JTables. We also use scroll panes for these panels, in case the data in the tables overlaps the margins of the frames. The View class is the most complex of this system.

- model package:

The model package contains 5 classes : Account, SavingsAccount, SpendingAccount, Bank, Client and an interface : BankProc.

Account Class:

The Account class implements the Serializable interface, used as a component in our system as it enables the serializability of a class. The Account class is also a superclass for the SavingsAccount and SpendingAccount classes. It has 4 attributes: accID of type int, holder of type String representing the username of the client that holds the account, money of type double representing the amount of money in the account, and typeOfAccount of type String representing the type of account. The class has getters and setters for these attributes, as well as a constructor for account instantiation. It also has a toString() method for displaying the account information. Moreover, the class has two methods for depositing and withdrawing money: withdraw(double amount) and deposit(double amount).

SavingsAccount Class:

The SavingsAccount class implements the Serializable interface, used as a component in our system as it enables the serializability of a class. The

SavingsAccount class is a subclass of the Account superclass. It has two extra attributes compared to its parent class : interestRate of type double and MIN = 5, a static final int attribute representing the minimum amount of money that a SavingsAccount can hold. The class overrides the two operation methods : withdraw(double amount) and deposit(double amount) of its parent class, as it needs more specific implementations.

SpendingAccount Class:

The SpendingAccount class implements the Serializable interface, used as a component in our system as it enables the serializability of a class. The SpendingAccount class is a subclass of the Account superclass. It has one extra attribute compared to its parent class : FEE = 1.5, a static final double attribute representing the amount of money that will be taken from the client when he makes a transaction of any kind. The class overrides the two operation methods : withdraw(double amount) and deposit(double amount) of its parent class, as it needs more specific implementations.

Client Class:

The Client class implements the Serializable interface, used as a component in our system as it enables the serializability of a class. The Client class has 6 attributes: username, firstname, lastname, address, ssn representing the Social Security Number of the client and password, all of type String. The Client class has a constructor, used to instantiate objects of type Client. For each of its attributes, the class has getter and setter methods, as well as a toString() method for proper display of a client. The two most important methods in the class are the overridden methods : hashCode() and equals(Object obj). The Object class defines the method: public int hashCode(). This method has to be overridden for the K object to return an integer computed based on the object's fields. The hashCode method must return the same integer for two equal, and different integers for different values. Implementing the hashCode() overridden method, we used a prime number, in this case, 31. The equals() method has to be overridden because the designer has to compare key equality for get operation or to check if the object exists in the map, in case of put operation.

Bank Class:

The Bank class implements the BankProc interface, created by the system's designer. The Bank class has one attribute: `HashMap<Client, ArrayList<Account>> bank`, representing the core of the bank system.

Hashmaps are data structures (like arrays and lists), that use hashing to store data. In a hashmap, a hash is produced (either from a provided key, or from the object itself) that determines where in the table the object is stored. A hashmap is used for storing Key & value pairs.

Moving on from the hashmap side of the class(with getter, setter and a constructor), the Bank class implements all the methods that have their header defined in the BankProc interface: `addClient`, `editClient`, `removeClient`, `addAccount`, `editAccount`, `removeAccount`, `deposit` and `withdraw`. The relationship between Bank and the BankProc interface is where we see the Design by contract Programming Technique. Using assert statements we can check the pre and post conditions declared as comments in the BankProc interface.

The Bank class also implements the two methods needed for serializing the system: `readBank()` and `writeBank()`. The methods use the java.io collection components : `FileInputStream`, `FileOutputStream`, `ObjectInputStream` and `ObjectOutputStream`. The serialized bank content is saved in the "bank.txt" file with the `writeBank()` method, called in the Controller constructor when the JButton "Save" is pressed. The `readBank()` method is called at the start of the Controller constructor.

BankProc:

An interface designed by the user. For each method header defined in the interface, later to be implemented in the Bank class, there are certain pre and postconditions that must be fulfilled. For example, the `addClient(Client client)` method's precondition is : the client must be different from null.

- main package:

The default package of the system, holding a single class: Main, where we instantiate a new Controller object.

4.Results:

The application is fully functional: the administrator is able to interact with client and account data (add, edit/update, delete) and the client is able to withdraw/deposit money from/to a specific account that he is the holder of.

The client has to log in with the credentials that the admin inserts into the bank, otherwise he will be unable to log in. The only combination for the administrator to log in is username: "admin" and password: "admin".

5.Junit Test Class:

The application also has a Junit Test Class, verifying the methods implemented in the Bank Class : addClient, updateClient, deleteClient, addAccount, updateAccount, deleteAccount, deposit and withdraw. Again, in order to verify the correct implementation of these methods, the user uses the assert statement. All 8 methods are correctly verified, and the tests go through.

6.Conclusions:

The student is presented with a close to real life system design, similar to Homework 3. If followed correctly, the given schema is easily implementable. The student's capability of working with the javax.swing components is furthermore developed as this homework requires great understanding of the Graphical User Interface. Also, the student further improves in working with the MVC data model, if he wishes to implement the system as such. The homework also presents inheritance relations and working with different interfaces, one of which the student implements. The design by contract programming technique is required for extra points.

For implementing these conditions, the student uses and learns about the assert statement. Using pre and postconditions is a great way of implementing methods with more strictness. The student also learns about the HashMap collection (in the case of this system), a data structure using

hashing in order to store data of type <Key,Value>. Also, regarding the HashMap, the student must override the Object methods hashCode() and equals(Object obj).

7.Further developments:

The system can implement a Design Pattern Observer for notifying the account holder when one of its accounts is modified. Also, for the two JTables, click listeners can be implemented. The JTables could also be refreshed in a better way after the serialization.

8.Bibliography:

http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW4_Tema4/HW4_Tema4_Hashing.pdf

<http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html>

<https://beginnersbook.com/2013/12/hashmap-in-java-with-example/>

<https://stackoverflow.com/questions/3572528/what-are-assert-statements-in-java>