MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Bank Application

- Project #4 -

Student: Dariana Lupea

Teacher: Delia Balaj

Group: 30425

Deadline: April 15, 2016

Table of contents

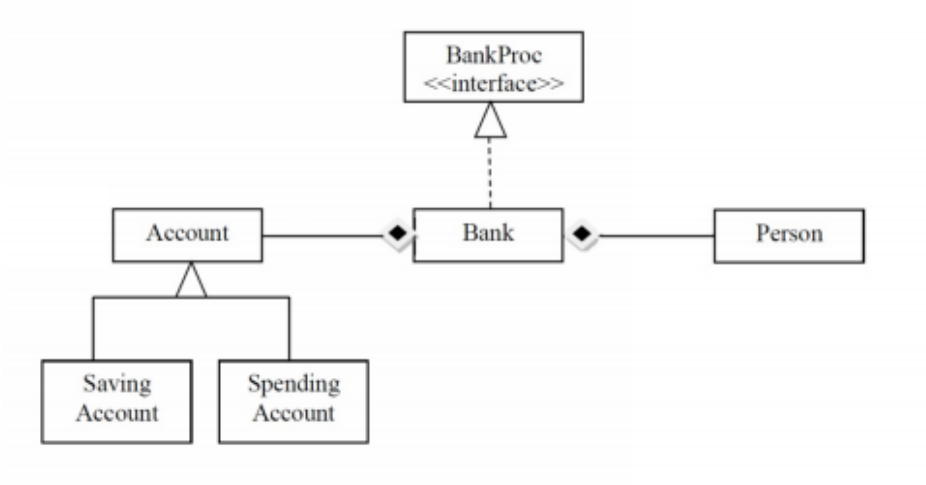
1. Introduction

1.1 Objective

The objective of this project is to use *Design by Contract Programming Techniques.*

1.2 Problem specification

Consider the system of classes in the class diagram below.



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.

2. General aspects

2.1 Problem analysis

The application should implement all the requirements given in the task description. The implementation of the bank will consist in a Hash Table that uses Chaining technique in case of collisions.
Specific methods for adding/removing and listing information from the Hash Table shall be implemented.
In order to allow a better understanding of how this application functions, assertions, contracts and invariants shall be used as well as Java documentation for the project.

2.2 Modeling

All the information that will be registered into the application from the graphical user interface, will be tested to be according to specific criteria, and specific dialog messages will appear in case that the user tries to input some wrong data. Information about clients shall be restricted to their name and email, phone number, which the Bank will use to distinguish clients.

A client can have one or more accounts. In this accounts, some operations can be performed like adding or withdrawing money. If the withdrawn amount exceeds the available amount in the account, an error message shall be generated.

The user has the possibility to delete client accounts or even delete totally clients from the bank. No measure has been taken regarding to the money that a user has at a given moment of time and the user removal at that time. The user can to output information into the graphical user interface, such that the bank’s state can be checked at any time. Information about the number of clients, the number of accounts, detailed information about the bank’s state and listing of account should be generated.

3. Projection

3.1. UML Diagrams

3.1.1 Use – case diagram

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved. This application can be used from two different perspectives: as a *bank client (account holder)* or as a *bank manager*, so the corresponding use-cases are illustrated below:

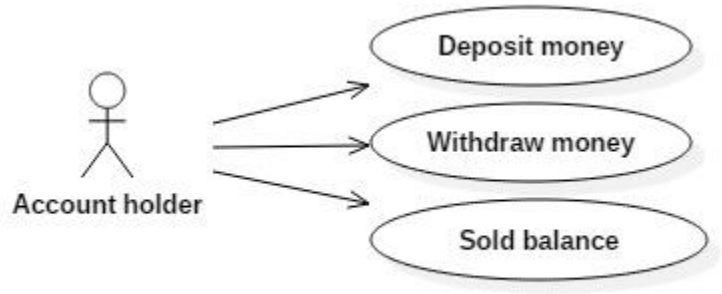


Fig. 1: Account holder: Use-case diagram

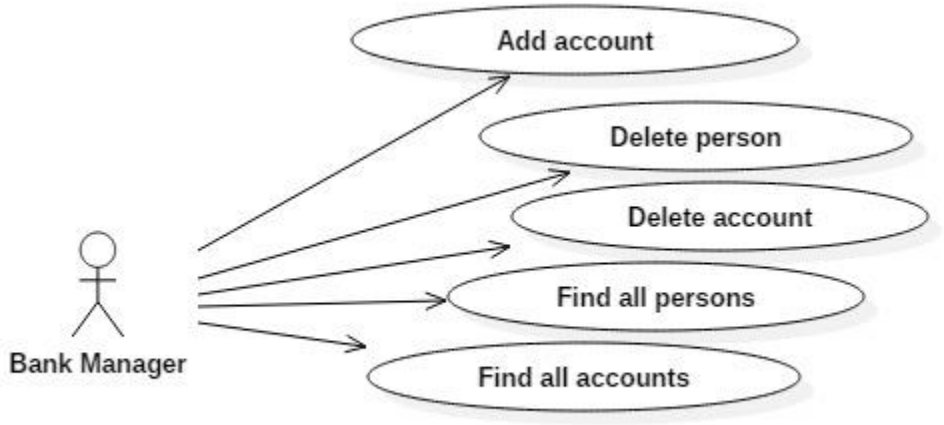


Fig. 2: Bank manager: Use-case diagram

3.1.2. Class diagram

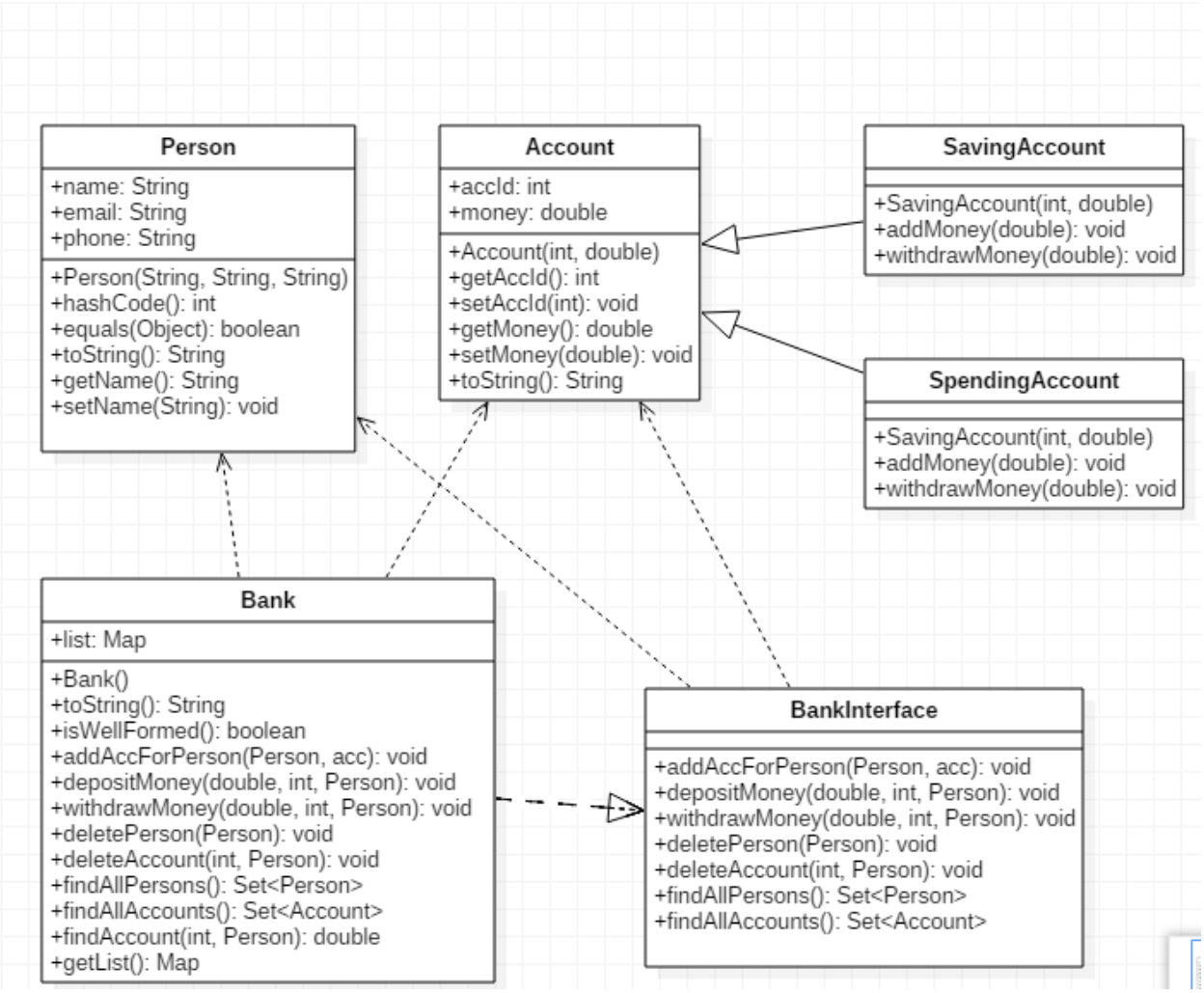


Fig. 3: Class diagram

3.1.3. Sequence diagram

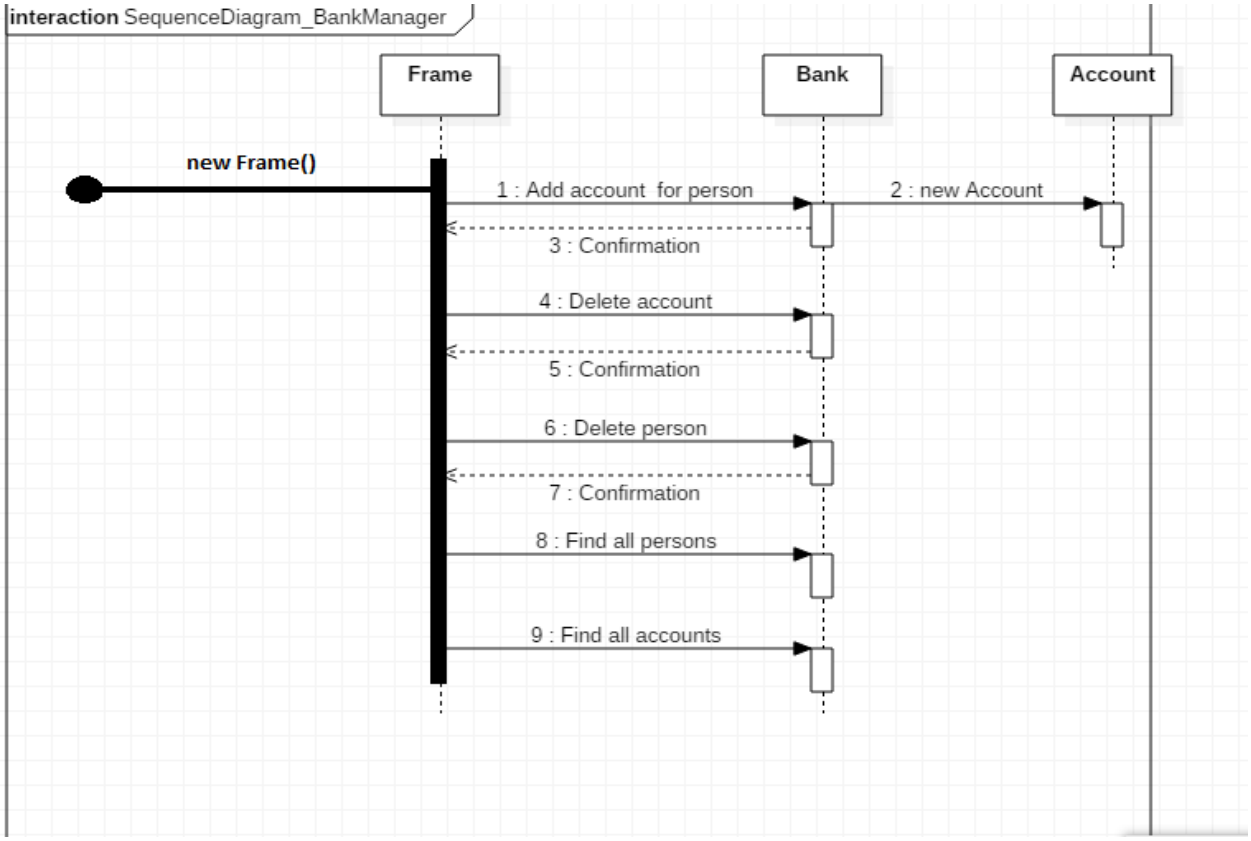


Fig. 3: Sequence diagram- Bank Manager

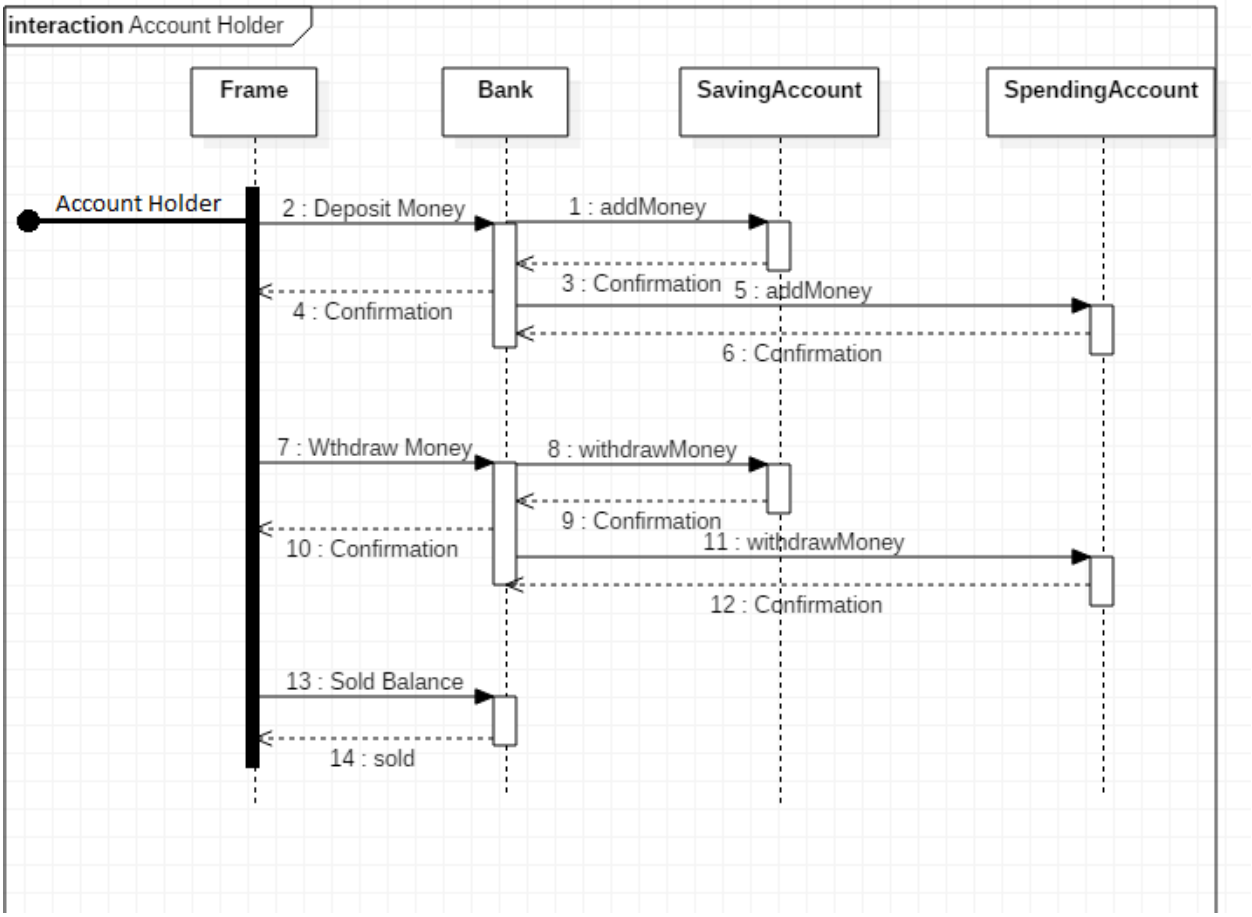


Fig. 4: Sequence diagram- Account Holder

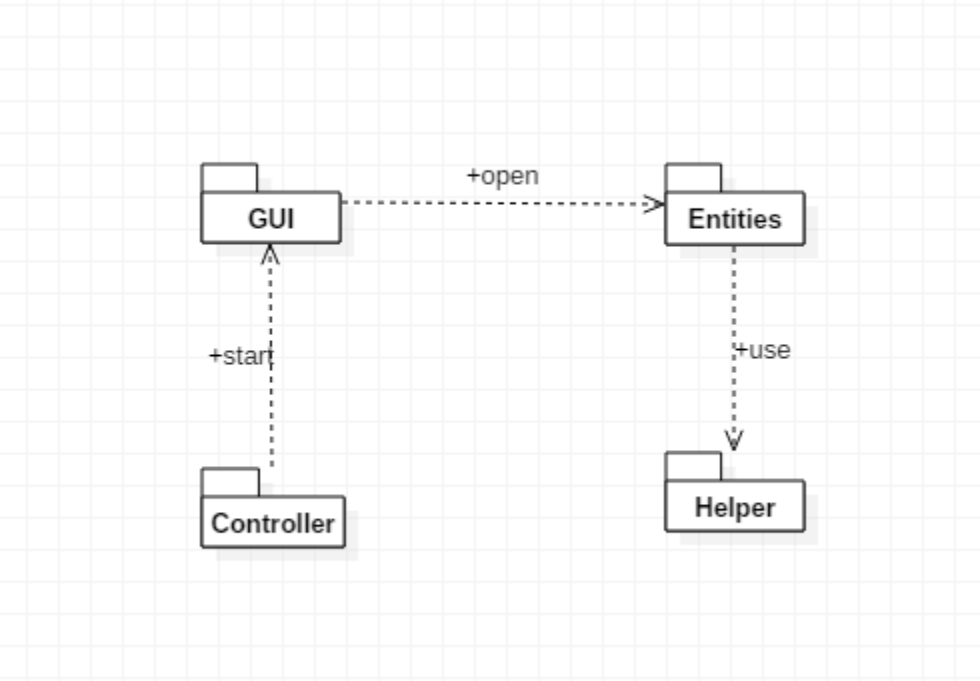


Fig. 5: Package diagram

3.2 Data Structures

In this project implementation I have used several data structures, but the most significant one is:

- **HashMap**

Hash Map is actually Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

Advantage: This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). It's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Disadvantage: **Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies

the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.)

Chosen data structure: Hash Map

Since this application will not use multithreading, there is no problem with using Hash Map. Moreover, we can come across the situation of permitting null entries. Another advantage of using Hash Map is that this collection is collision safe, because the entries of the table act like a linked list. When you put a new entry into the same bucket, it just adds to the linked list. If the hash of the key in the map collides with an existing key, the Map will re-arrange or keep the keys in a list under that hash. No keys will get overwritten by other keys that happen so be sorted in the same bucket.

3.3 Class Projection

After succeeding in analyzing the problem, we can go further to developing the classes that are necessary for the implementation of this application.

I obtained the final version of the system's structure. It contains 5 packages, namely: **Controller, Entities, GUI, Helper** and **TestDriver**. Each of these consists of other classes, which perform specific tasks. I will present them below:

- **Entities** package:
  - **Person** class
  - **Account** class
  - **SpendingAccount** class
  - **SavingAccount** class
  - **Bank** class
  - **BankInterface** class

- **GUI** package:
  - **Frame** class – creates the user interface
- **Controller** package:
  - **Main** class

- **Helper** package:
  - **UserInputChecker** class

- **Serialization** class

- **TestDriver** package:
  - **TestUnit** class

3.4 Packages

3.4.1 **Entities** package

It contains 6 classes, which capture the behavior of the application in terms of the problem domain: **Person**, **Account, SpendingAccount, SavingAccount, Bank** and **BankInterface**.

a) Person class

It models an entity of type **Person**. Each person has the following properties: *name*, *email* and a *phone number*.

```
public Person(String name, String email, String phone) {
    super();
    this.name = name;
    this.email = email;
    this.phone = phone;
}
```

- This class also contains some **getters** and **setters** which allow the modification of the attributes mentioned before.

b) Account class

This class represents an account. Each account has an id and an amount of money. It is an abstract class and it contains 2 abstract methods:

- public abstract void addMoney(double sum)
- public abstract void withdrawMoney(double sum)

The constructor is used for initializing the account state.

```
public Account(int accId, double money) {
    this.accId = accId;
    this.money = money;
}
```

- This class also contains some **getters** and **setters** which allow the modification of the attributes mentioned before

c) **SpendingAccount** class

This class extends the **Account** class and implements **Serializable** interface. It also needs to implement the two methods (addMoney and withdrawMoney) inherited from the superclass **Account.**

```
public void addMoney(double sum) {
      this.money += sum;
      setChanged();
      notifyObservers("The sum " + sum + " was added to your
                                          spending account");
}

public void withdrawMoney(double sum) {
 if (sum < 500 && this.money > -1000) {
      this.money -= sum;
      setChanged();
      notifyObservers("The sum " + sum + " was withdrawn from
      your spending account");
 } else {
      notifyObservers("Sorry, this sum cannot be withdrawn");
 }
 }
```

d) SavingAccount class

This class extends the **Account** class and implements **Serializable** interface. It also needs to implement the two methods (addMoney and withdrawMoney) inherited from the superclass **Account.** The only difference between this class and **SpendingAccount** class is that the *withdrawMoney* method has some different constraints, as it can be seen in the following peace of code:

```
public void withdrawMoney(double sum) {
if (sum < 200 && this.money > 0) {
      this.money -= sum;
      setChanged();
      notifyObservers("The sum " + sum + " was withdrawn from
      your spending account");
} else {
      notifyObservers("Sorry, this sum cannot be withdrawn");
}

      }
```

e) **Bank** class

This class contains the operations that can be performed by the client and also by the bank manager:

```
public void addAccForPerson(Person p, Account assocAcc);
public void depositMoney(double sum, int accId, Person p);
public void withDrawMoney(double sum, int accId, Person p);
```

```
public void deletePerson(Person p);
public void deleteAccount(int accId, Person p);
public Set<Person> findAllPersons();
public Set<Account> findAllAccounts();
public double findAccount(int accId, Person p);
```

f) BankInterface class

It contains the definition of the methods that are implemented in the Bank class. Each method has some *preconditions*, *postconditions* and an *invariant* (isWellFormed).

3.4.2 GUI package

It contains a class responsible with creating the user interface. I used **Swing**, which is a GUI widget toolkit for **Java:**

- **a top level container:** the class *Frame* extends **JFrame**
- **JComponents: JTable, JLabels, JTextFields, JRadioButtons:**
  - **private JTable** table**;**
  - **private JButton ;**

private JButton btnAddAccount, btnDeleteAccount, btnDeletePerson, findAllPersons, findAllAccounts;
private JButton btnDeposit, btnWithdraw, btnSold, saveBank;

  - **private JTextField field1, field2, field3, field4, field5, field6.**

These JTextFields are used for taking the user inputs:
private JTextField textName, textEmail, textPhone, textId, textMoney;

  - **private JLabel;**

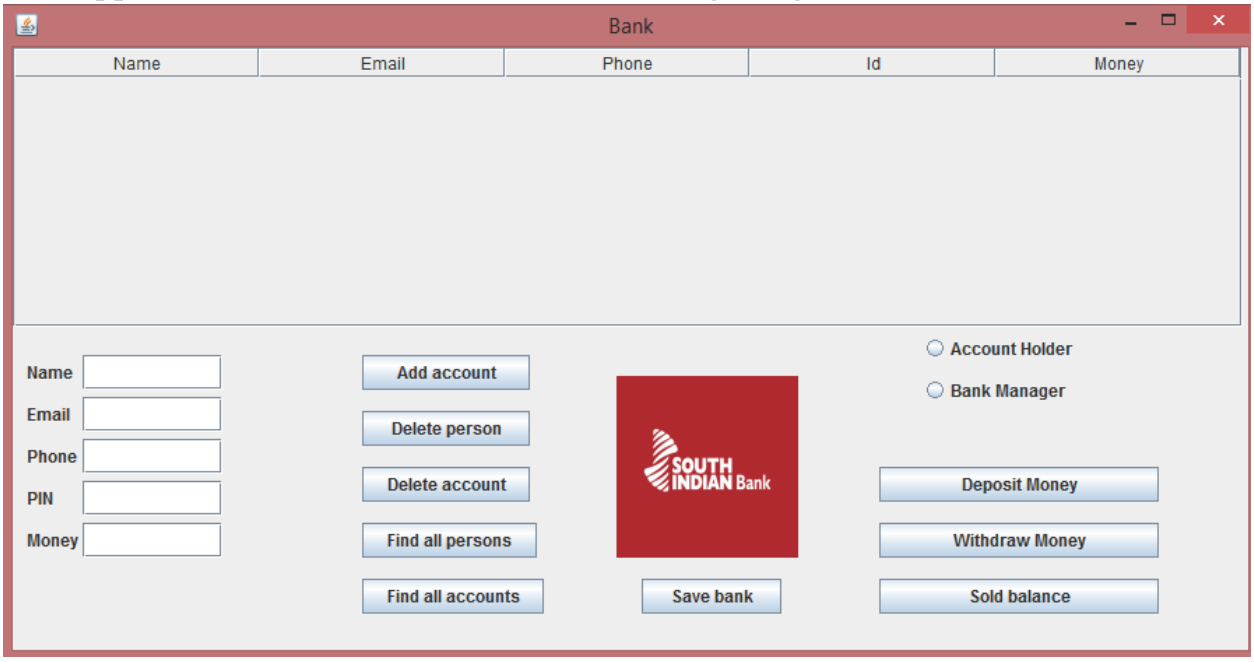
private JLabel lblName, lblEmail, lblPhone, lblId, lblMoney, bankLabel;

- **private JRadioButton;**

private JRadioButton userButton, managerButton;
private ButtonGroup group;

3.6 Interface

This application GUI looks like in the following image:



- The **user** can use this application in 2 different modes: as an Account Holder or as a Bank Manager.

a) Account Holder

This type of user can perform the following operations: **Deposit Money, Withdraw Money** and check the **Sold Balance.** The account holder must:

- Enter the name
- Enter the email
- Enter the phone number
- Enter the PIN
- Enter the amount of money
-

4. Implementation and testing

Regarding the implementation process, I used the **Eclipse IDE**. During the program development steps, many changes were made and as I started to have more and more methods I realized the importance of a good structure. I refer here to organization of the code in packages and classes. I consider that the code I wrote is understandable and reusable. The algorithms I used are relatively easy, based on the well-known mathematical algorithms.

I have also tested each method, in the following way:

-> I used the console at first, for displaying the result
-> I initialized data with some appropriate values
-> I checked if the obtained result was the expected one
-> if YES, I continued the implementation process
-> if NO, I tried to figure out where is the error / problem coming from and solve it

5. Results

Concerning this project, I believe that it covers all the requirements from the problem specification and it fulfills the main purpose, which is performing some specific operations, in order to manage a bank.

6. Conclusion

To conclude, I can say that this project meant hard work, a lot of new things learned, focusing, development and creativity. Even if I encountered a lot of problems, I was able to fix them after all, by searching on the internet or asking a colleague for advice. I think that my application satisfies the requirements and the users will have at their disposal all its functionalities.

7. References:

- http://stackoverflow.com
- http://www.tutorialspoint.com/junit/junit_writing_tests.htm
- http://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html
- http://www.tutorialspoint.com/java/java_serialization.htm