Bank Management

Student: Suciu Mihai

Group: E_30422

Cuprins

1. Objective

The requirement of this project is :

Design by Contract Programming Techniques

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 (in 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.

The main objective of this homework was to design and implement an application Bank Management for processing some persons and their accounts where hash maps are used to store the persons and the accounts. This account can be spending or saving accounts.

This problem can be split as follows:

- Creating the Person and Account classes.
- Creating the saving account and spending account subclasses for Account.
- Creating the bank account and save data in one hash map.
- Creating the file where will be saved all data from hash map.
- Creating the graphical interface.

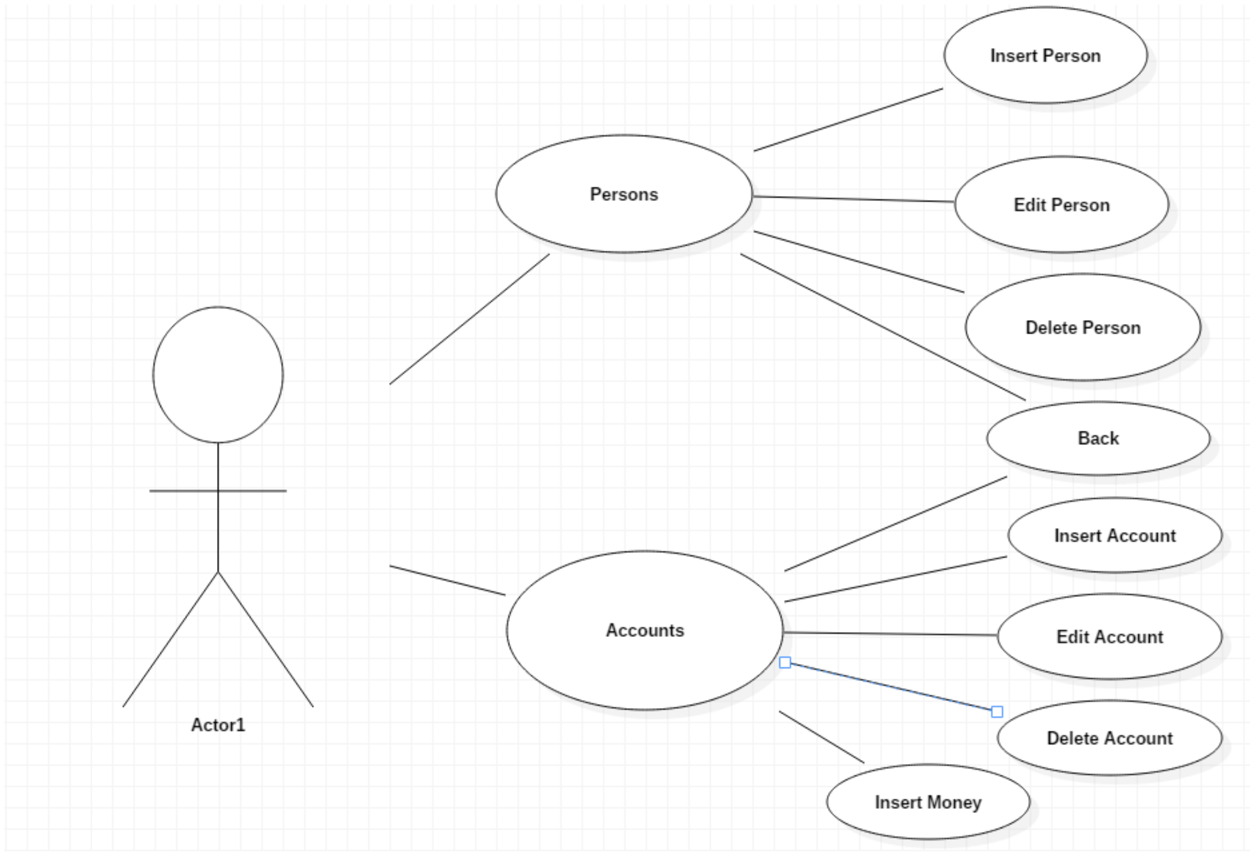
2. Problem analysis, use case diagram, assumptions

2.1 Problem analysis

- For this implementation I used design pattern Model-View-Controller (MVC). **Model–view–controller** (**MVC**) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.
- The *model* is the central component of the pattern. It expresses the application's behavior in terms of the problem domain independent of the user interface. It directly manages the data, logic and rules of the application.
- A *view* can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- The third part or section, the *controller*, accepts input and converts it to commands for the model or view.
- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view means presentation of the model in a format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

2.2 Use Case diagram

The application will contain a graphical interface, where the actor client will be able to introduce the information, and after it to perform the insertion, update and delete the data in hash table and after it in the text file.



CASE I

Actor: Client

Case: **without** error

Insert one Person in Hash Table

a) Press “Persons”
b) The client writes input data: Id , Name and CNP.
   For example:
   Id: 2
   Name : Mihai
   CNP : 12314
c) Press “Insert Person”
d) You will see in the table the new person.

CASE II

Actor: Client

Case: **without** error

Update one Person in Hash Table

a) Press “Persons”
b) The client writes input data: Id , Name and CNP.
   For example:
   Id: 2
   Name : Mihai
   CNP : 12314
c) Press “Edit Person”
d) You will see in the table the person changed.

CASE III

Actor: Client

Case: **without** error

Delete one Person in Hash Table

a) Press “Persons”
b) The client writes input data: Id , Name and CNP.
   For example:
   Id: 2
   Name : Mihai
   CNP : 12314
c) Press “Delete Person”
d) You will see the table updated.

CASE IV

Actor: Client

Case: **without** error

Insert one Account in Hash Table

a) Press “Accounts”
b) The client writes input data: Id ,sold, persons id and account type.

For example:

Id: 2
sold : 500
person id:1
account type:0

c) Press “insert account”
d) You will see the table updated.

CASE V

Actor: Client

Case: **without** error

Edit one Account in Hash Table

a) Press “Accounts”
b) The client writes input data: Id ,sold, persons id and account type.
   For example:

   Id: 2
   sold : 500
   person id:1
   account type:0
c) Press “edit account”
d) You will see the table updated.

CASE VI

Actor: Client

Case: **without** error

Delete one Account in Hash Table

a) Press “Accounts”
b) The client writes input data: Id ,sold, persons id and account type.
   For example:

   Id: 2
   sold : 500
   person id:1
   account type:0
c) Press “Delete account”

You will see the table updated

CASE VII

Actor: Client

Case: **without** error

Delete one Account in Hash Table

a) Press “Accounts”
b) The client writes input data: Id ,sold, persons id and account type.
   For example:
   Id: 2
   sold : 500
   person id:1
   account type:0
c) Press “Delete account”

You will see the table updated

2.3 Assumptions

The client will know that do not request to display one empty Hash table.

The client will not introduce the same id as another object from hash table.

The client will not press insert buttons without write any data in text fields.7

The client will know that do not request to display one empty Hash table.

The client will not introduce the same id as another object from hash table.

The client will not press insert buttons without write any data in text fields.

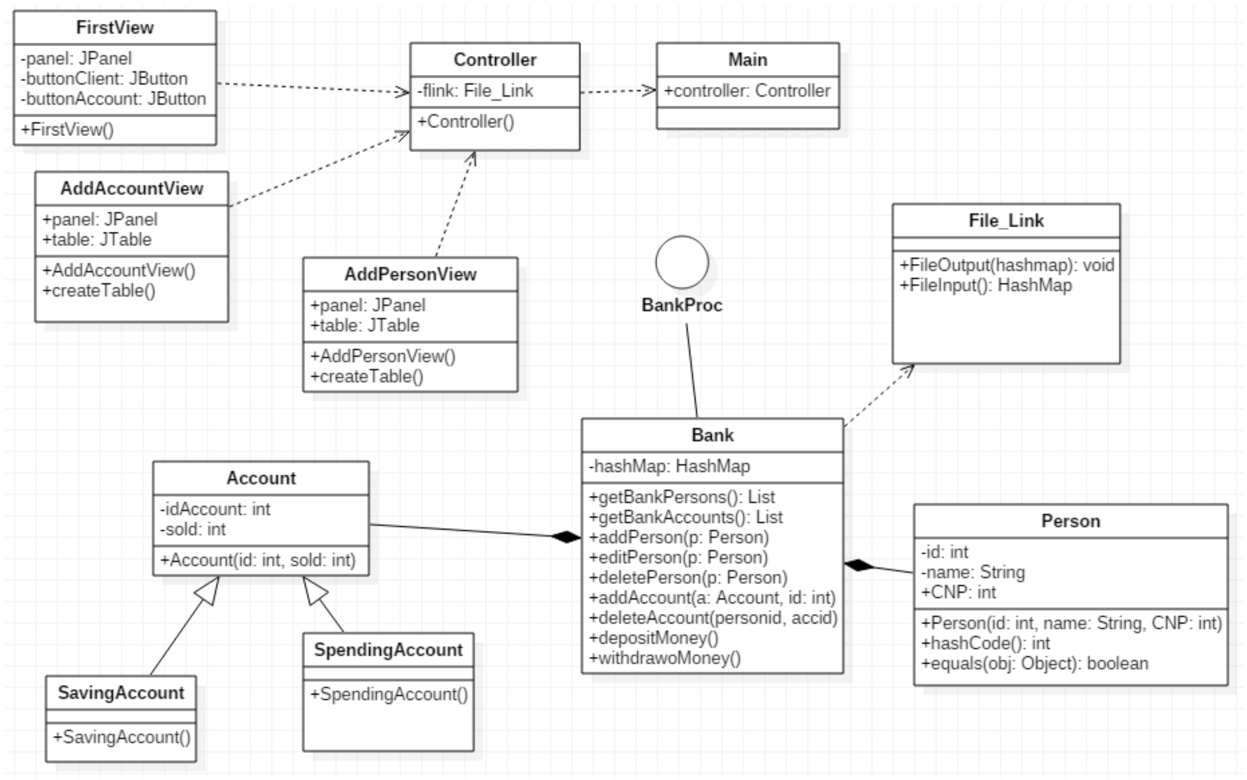
The client will know that do not request to display one empty Hash table.

The client will not introduce the same id as another object from hash table.

The client will not press insert buttons without write any data in text fields.

3. Design

3.1 UML Diagram



3.2 Class design

The projection of this application started Account and Person classes. I will represent below each class with their attributes and operations. Here we have four attributes and one constructor. The same in the Account and Person classes.

After this, I implemented two classes which extends Account. This two are Spending account and Saving account. The saving account allows a single large sum deposit and withdrawal and computes an interest during the deposit period. The spending account allows several deposits and withdrawals but does not compute any interest.

All bank headers are in Bank Proc interface. The next class and the most important is the Connection class. Here I have one string and one connection, one constructor and two methods, to create and close the connection.

3.3 Graphical interface

I will present below the View class with its attributes and methods. The first 12 text fields are used for input data, 4 for each class : Client, Product and Order. Here you need to write: for client -id, name, mail, address, for Product – id, name, price, quantity and for Order – id, Product id, Client id and the quantity you want to buy.

The next 12 buttons are used to help you to access the data bases. Here you can insert one client, one product or one order. If you missed something you can modify with the Update buttons. After it if you don't want one of client / product or order, you can delete it at any time. Last 4 buttons are used to check the tables and show it in right of the graphical interface. The last argument here is the table which help me to show you something when you press View buttons.

Here I have some methods which create the action listener for each button.

4. Implementation

For this implementation I used design pattern Model-View-Controller (MVC). **Model–view–controller** (**MVC**) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

The *model* is the central component of the pattern. It expresses the application's behavior in terms of the problem domain independent of the user interface. It directly manages the data, logic and rules of the application.

A *view* can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

The third part or section, the *controller*, accepts input and converts it to commands for the model or view.

The model is responsible for managing the data of the application. It receives user input from the controller.

The view means presentation of the model in a format.

The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

Classes & important methods

First class is Person. Here I have 3 attributes , one constructor and getters and setters for each attribute.

```
public Person(int id, String name, int CNP) {
        this.id = id;
        this.name = name;
        this.CNP = CNP;
}
```

Like this one I have 2 more classes with the same constructor, getters and setters : Person and account. I will show below only the attributes.

For Person:

```
public int id;
public String name;
public int CNP
```

For Account:

```
public int idAccount;
public int sold;
public int PersonId;
public int accountType
```

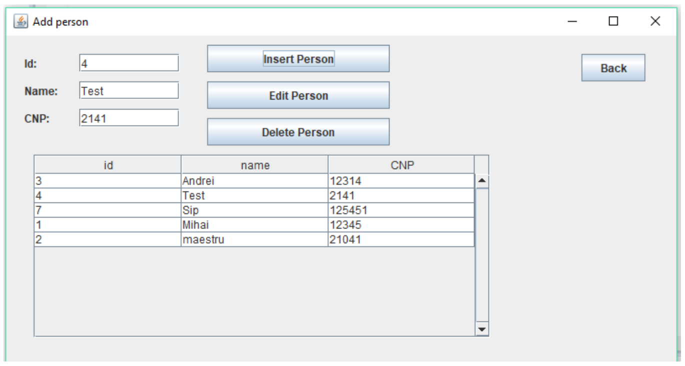
Here I have a list of Accounts, one frame , two attributes (and nr) which help me to calculate the medium wait time for each server , one Boolean (running) which show me if the Thread (thread) runs , and one index for each server. Most methods of this class help me to start the threads (start , stop , run),” “ is used to introduce a client in array list, “” returns the length of the server and “ “ returns the total serve time for each server.

Here I have one view, the variables for input, one thread and one random variable. Through the methods I have one to return a random number between the parameters and one to starts the thread.   The rest methods are used to help me to assign the next client in the server which has the minimum wait time.

The last class is the view one. Here I implemented the graphical interface.

5. Results

You need to write the input data, press “Insert Person” and after it you will see this table



Id:	4
Name:	Test
CNP:	2141

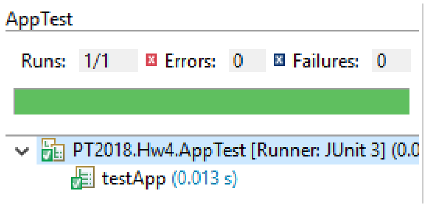
id	name	CNP
3	Andrei	12314
4	Test	2141
7	Sip	125451
1	Mihai	12345
2	maestru	21041

6. Tests

Here I tested if I can introduce something in hash map :

```
public void testApp() {
		Bank bank = new Bank();
		Person person = new Person(13, "mihai", 1241);
		bank.addPerson(person);
		for (Person i : bank.getBankPersons()) {
			if (i.getId() == 13) {
				assertTrue(person.equals(i));
			}
		}
}
```

And here is the result :



```
AppTest
Runs: 1/1    Errors: 0    Failures: 0

PT2018.Hw4.AppTest [Runner: JUnit 3] (0.0
    testApp (0.013 s)
```

7. Conclusions

During the design of this project I improved my abilities in interfaces. Before this I learned how to use the concept of hash table and the connection between this one and java and this is the most important think for every programmer. This project was hard just from the point of view of connection with hash table, the rest contain only previously learned items.

Further development: improve the graphical interface, introduces some imagines of accounts, and more error messages.

8.Bibliography

- https://en.wikipedia.org/wiki/Java_hashCode()
- https://dzone.com/articles/working-with-hashcode-and-equals-in-java