UNIVERSITATEA TEHNICĂ CLUJ-NAPOCA

Programming Techniques

Laboratory Work - Homework 4

Name: Bologa Marius Vasile

Group: 30425

1.Project objectives

Objective

Design by Contract Programming Techniques Description 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. Problem analysis , modeling scenarios , use cases

2.1 Problem analysis

The analysis of a problem starts from examining the real model or the model we confront with in the real world and passing the problem through a laborious process of abstractization. Hence we identify our problem domain and we try to decompose it in modules easy to implement. Starting from the concept of magement of the account, I have started to underline the main, objects presented in the process of administrating an account, the admin, user, account, bank etc. In order to be able to design and build a software program that performs and satisfies all of the specifications and requirements presented above it is very important to understand all the opperations needed in order to hava a sustainable application.

2.2 Modeling

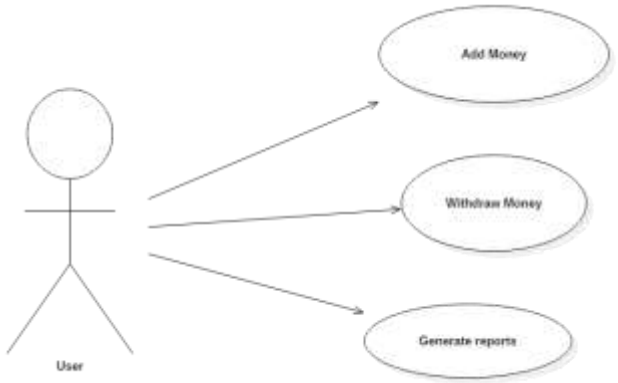
First of all, we need to find some means to store the Accounts.In order to save the accounts and information regarded to them, I have used the interface Serializable, in order to have my object written as a sequence of bytes when the program ends, and to restore the bank at it's previous status, when the app was closed. In order to do that, all the classes that contributed in a way or another to the built of the other classes that implements the interface Serializable, must implement by their own the interface. If we want that a field from a class not to be serialized we must declare it as „transient". I have used a HashMap<Person,ArrayList<Account>> in order to get the persons and the accounts of each person from the bank.

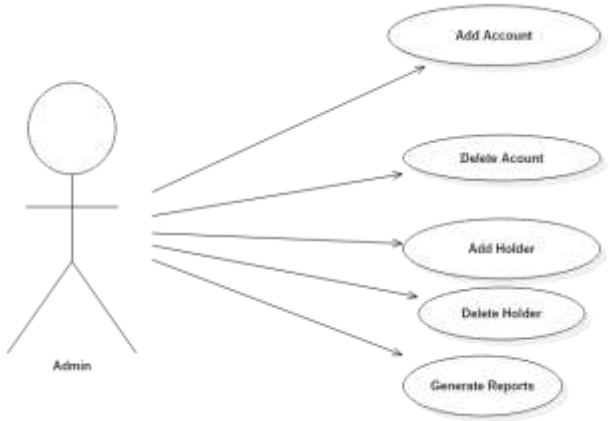
3. Projection and Implementation

In what the implementation is concerned this project was developed in Eclipse and it was only tested in this environment. However the program should maintain its portability. Concerning the code implementation I did not make use of laborious algorithms, but I have rather stayed faithful to the classical implementation of an order management system.

3.1 UML diagrams

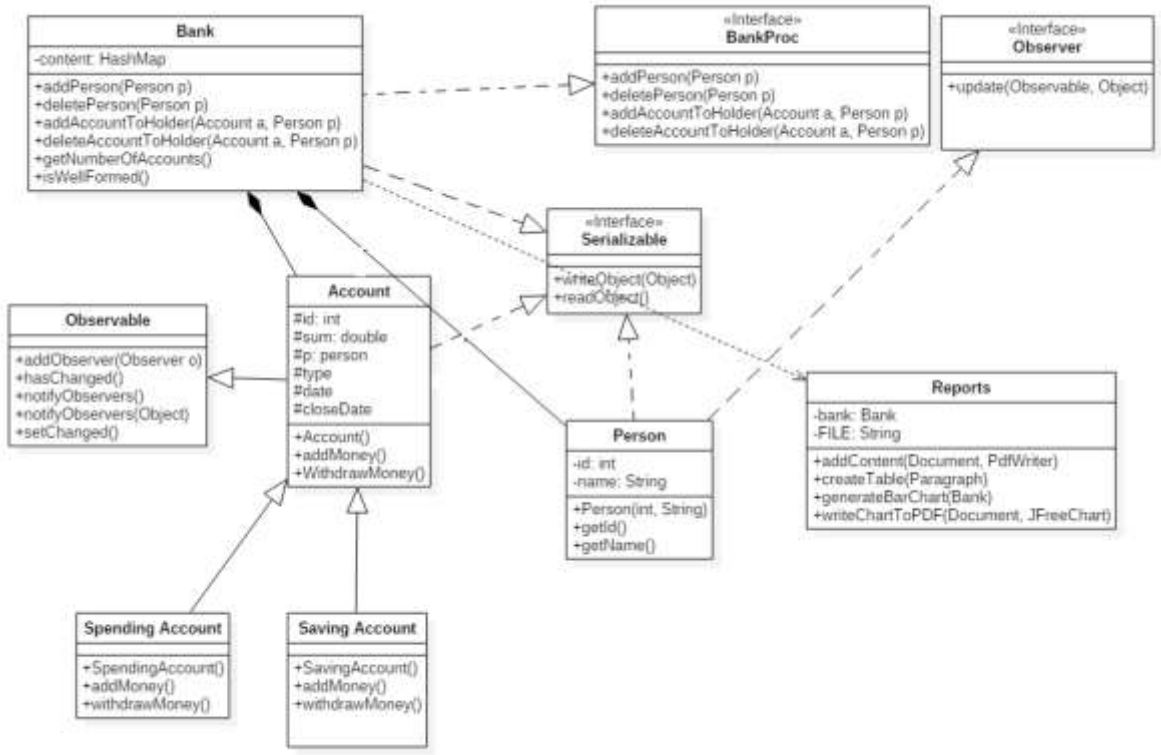
3.1.1 Use case diagram



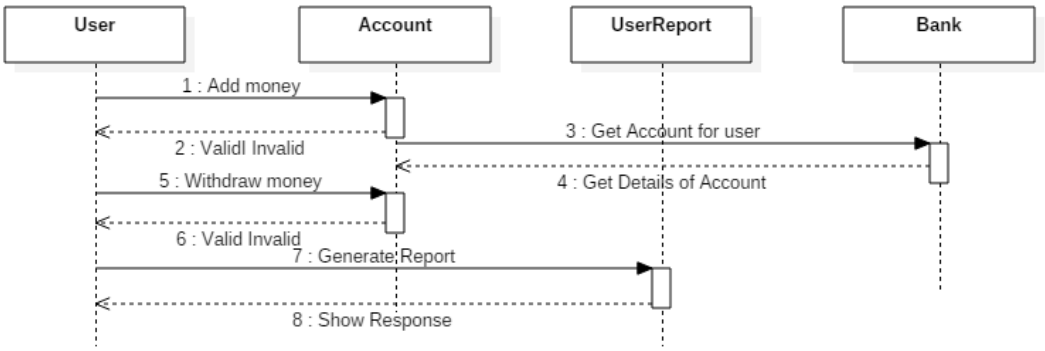


The use case diagram is nothing but the system functionalities written in an organized manner, presentig the actor, which is the admin in this case and the operations provided by the system for the user.

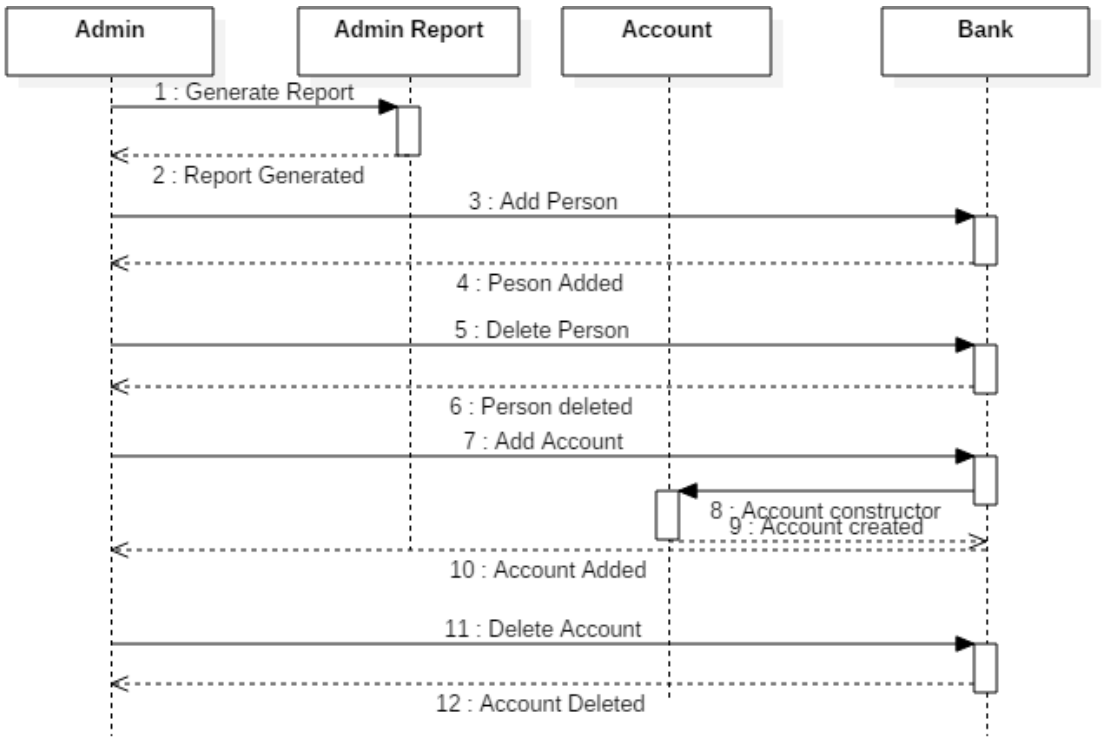
3.1.2 Class diagram



3.1.3 Sequence diagram

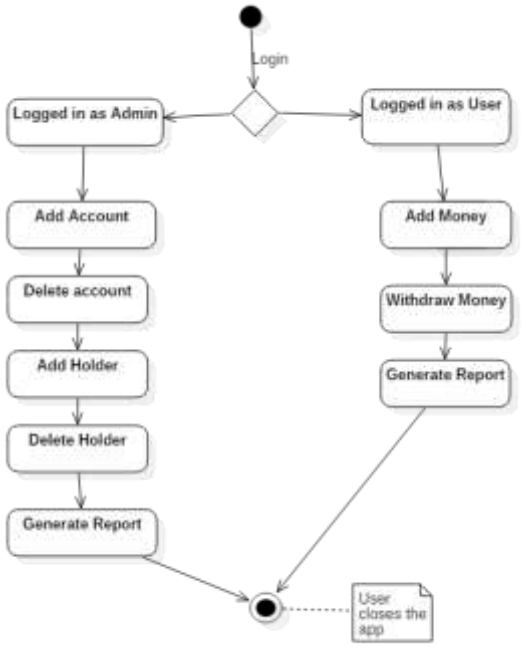


This is the sequence diagram for the user use case.



This is the sequence diagram for the admin use case.

3.1.4 Activity diagram



3.2 Data structures

The data structures used at this problem are either primitive data types such as integers or more complex data structures such as a HashMap or new created objects such as Person,Account, Banl. The HashMap was introduced, to have a way to store the accounts and the owners of the account.

3.3 Class projection

Class projection refers mainly to how the model was thought, how the problem was divided in sub-problems, each sub-problem representing more or less the introduction of a new class. First I will start by mentioning exactly how my problem was divided into packages and afterwards each package with its own classes. I begin by creating the four packages I used: the first one being called „view" the second one being called „model", as a third one is a subpackage of the model package and the fourth one „controllers". I named them intuitively because the first one handles the interface part, the part that deals with the user, the second one handles the implementation part, the third package is the one that provides usefull functions and generates reports and the fourth one, is the one that contains the controllers of the interfaces. I will begin with the interface, then I will continue with the models, models.utilities and controller part.

3.4 Interface – package view

LoginFrame.java - extends JPanel , with which we made a login window for the application -some elements of the interface are: labels , text fields, radio buttons, buttons and JPasswordField. This class has only 3 instance variables, presented below:

private static JRadioButton *userButton*: a JradioButton to choose to login in as a user
private JRadioButton adminButton: a JradioButton to choose to login in as an admin
private static String *userName*: a String to keep the name of the user that has logged in.

The main method of this class „createLoginWindow", constructs a Jpanel that has one text area where the user can introduce his name, a password field, and 2 buttons „OK" and „Cancel".We also have setters and getters for the instance variable:
public static JRadioButton getUserButton(): get the status of the user button
public void setUserName(String userName): set the username
public static String getUserName():get the name of the user
public void setAdminButton(JRadioButton adminButton): set the status of the user button

MessageDialogs.java – this class contains the main dialog windows that are created in order to get for exemple the name of the object searched, or all the details needed for an order to be computed. The class does not contain instance variables, and have only 4 methods:

public static String[] addAccountWindow() : this method is used for creating a Joptionpane that will collect the data that is put into the fields of the pane, into an array of strings, from where we construct an Account,and add the new object to the Bank and to a table.

public static String[] addHolderWindow():this method is used for creating a Joptionpane that will collect the data that is put into the fields of the pane, into an array of strings, from where we construct a Person,and add the new object to the Bank and to a table.

public static String[] addMoneyWindow():this method is used for creating a Joptionpane that will collect the data that is put into the fields of the pane, into an array of strings, from where we modify the sum of money from the account.

public static String[] withDrawMoneyWindow():this method is used for creating a Joptionpane that will collect the data that is put into the fields of the pane, into an array of strings, from where we modify the sum of money from the account.

UserFrame.java- this class represents the GUI for the user. The class contains the following instance variables:

private JPanel topPanel: used only for settle the components into the frame
private JTabbedPane jtp: used for going easy between the frames
private JPanel btnPanel: used to settle the buttons in to the bottom of the page
private JButton addMoney, withdraw,generateReports: the main buttons representing the actions a user can do
private static JTable *table* :a table where we have the content of the bank
private static JTable *reflectiontable* :a table where we have the reflection of the class account
private JScrollPane scrollPane : used especially for the tables to see the data easier
private static JTable *orderTable :* a table where we have all the orders of the orders departament

Constructor of this class places all the components enumerated above on the frame.
We have also getters and setters for the instance variables, and also we set the Listeners for the buttons.

```
public static JTable getTable()
public void setTable(JTable table)
public final void setAddMoneyButtonActionListener(final ActionListener a)
public final void setwithDrawMoButtonActionListener(final ActionListener a)
public static JTable getOrderTable()
public static void setOrderTable(JTable orderTable)
```

AdminFrame.java- this class represents the GUI for the admin. The class contains the following instance variables:
private JTabbedPane jtp : used to navigate easy between the frames
private JScrollPane scrollPane: used especially for the tables to see the data easier
private static JTable *table:* a table where we have the contenst of the bank
private static JTable *reflectionTable:* a table where we have all the persons obtained from the reflection of the person's class
private JButton addAcc, delAcc, addHolder, deleteHolder, generateReports: buttons for the main operations that an admin can operate.

Constructor of this class places all the components enumerated above on the frame.
We have also getters and setters for the instance variables, and also we set the Listeners for the buttons.
public static JTable getTable()
public void setTable(JTable tableStock)
public final void setAddAccActionListener(**final** ActionListener a)
: add action listener for the add account button
public final void setDelAccActionListener(**final** ActionListener a)
: add action listener for the delete account button
public final void setAddHolderButtonActionListener(**final** ActionListener a)
: add action listener for the add holder button
public final void setDeleteHolderButtonActionListener(**final** ActionListener a)
: add action listener for the delete holder button
public final void setGenerateReportsActionListener(**final** ActionListener a)
: add action listener for the generate reprts button
public static JTable getTableOrder()
public void setTableOrder(JTable tableOrder)

3.5 Models

3.5.1 Models

Person.java –the class represents the user that has logged in with it's name and the password.

The class has only 2 instance variable a String that is actually the name of the user and the id of the user.There is just 2 methods besides the constructor, that get the name of the user/id of the user. Implements serializable, in order to be saved, and to recover the information regarding the user. Also the class implements Observer that contains only one method update, and when the user will modify something on the account the user will be notified.

```
@Override
public void update(Observable arg0, Object arg1)
@Override
public boolean equals(Object pers)
```

The part of the contract here which is important is: **objects which are .equals() MUST have the same .hashCode()**.

```
@Override
public int hashCode()
```

The problem you is with collections where unicity of elements is calculated according to both.equals() and .hashCode(), for instance keys in a HashMap.
Account.java- this class is one of the most important classes in this project. It contains the following instance variables:

```
protected int id;--id of the account
	protected double sum;--sum of the money from the account
	protected Person p;--person that owns the account
	protected String date;--open date for the account
	protected String closeDate;--closing date for the account
	protected String type;-- type of the account
```

The constructor of the Account class:

```
public Account(double sum, Person p, String date, String type) {
		this.id = 100000 + rand.nextInt(900000);
		this.sum = sum;
		this.p = p;
		this.date = date;
		this.type = type;
		this.closeDate = Utilities.getDateRandom(3650);

	}
```

The class is abstract and has 2 abstract methods that will be implmented by it's subclasses

```
public abstract void depositMoney(double parseDouble);
--add money to the account
public abstract void withdrawMoney(double parseDouble);
--withdraw money from the account
```

The class implements serializable, in order to be saved after the application is closed. We also have getters and setters for the instance variables.

SpendingAccount.java- The class extends the class Acount and implements the methods defined by superclass

The constructor of this class is:

```
public SpendingAccount(double sum, Person p, String date, String type) {
		super(sum, p, date, type);
	}
```

The difference between Spending Account and Saving Account is that, when the user withdraws money from the account, there will be applied an interest rate for the sum that is withdraw.The interest rate will be computed regarding the numbers of days left until the account is valid.

SavingAccount.java- - The class extends the class Acount and implements the methods defined by superclass

The constructor of this class is:

```
public SavingAccount(double sum, Person p, String date, String type) {
		super(sum, p, date, type);
	}
```

The difference between Spending Account and Saving Account is that, when the user adds money to the account, there will be applied an gain rate for the sum that is added.The gain rate will be computed regarding the numbers of days left until the account is valid.

Bank.java- this class represents the content of the bank (the persons / accounts) . The bank is represented as a HashMap that has Persons as keys and Accounts as values.The constructor :

```
	public Bank() {
		this.content = new HashMap<Person, ArrayList<Account>>();}
```

The following methods are from the Bank class, and for the first method I left the implementation of the assertion for the pre/post conditions.

```
@Override
public void addPerson(Person p) {
        assert isWellFormed();
        assert p != null;
        assert p.getName() != null;
        int n = getNumberOfPersons();
        if (!content.containsKey(p)) {
                content.put(p, new ArrayList<Account>());
        }
        assert (getNumberOfPersons() != 0) && (getNumberOfPersons() == n + 1);
        assert isWellFormed();
}
```

public void deletePerson(Person p)
public void addAccountToHolder(Account a, Person p)
public void deleteAccountToHolder(Account a, Person p)
public int getNumberOfPersons()—get the number of persons from the bank
public int getNumberOfAccounts()—get the number of accounts from the bank
private boolean isWellFormed()—this method checks if the bank is well formed, as follows, get the numbers of persons and accounts and stores into a variable, then iterate through haspmap, and count each person and account, the at the end if the number of persons/accounts obtained at the begining of the function is equal with the number of persons/accounts obtained from the iteration than the bank is well formed, otherwise no.

BankProc.java- the interface BankProc contains the main functions that a bank can have like addPerson, deletePerson, addAccountToHolder(), deleteAcountFromHolder(), generateReport() and for each of this function are defined preconditions and postcondition definede as comments and implemented using assert instruction in Bank class. Also each method have defined an Invariant for the bank class method isWellFormed (will be used as both precondition and postcondition for each method in Bank). Methods from the interface are listed below and for the first function there are presented the pre/post conditions and the invariant.

```
/**
        * Method for adding a new person
        *
        * @invariant isWellFormed()
        * @pre p != null && p.getName()!=null
        * @post getNumberOfPersons() == getNumberOfPersons()@pre + 1
        * @post get(p) != null
        * @invariant isWellFormed()
        * @param p
        *             Person to be added
        */
       public void addPerson(Person p);
       public void deletePerson(Person p);
       public void addAccountToHolder(Account a, Person p);
       public void deleteAccountToHolder(Account a, Person p);
       public void generateReports();
```

3.5.2 Models.Utilities

AdminReport&UserReport these 2 classes generates a pdf that contains informations and reports about accounts and persons. The constructor of the classes is almost the same:

```
public AdminReports() {
        try {
                bank = m.deserializeBank();
                Document document = new Document();
                PdfWriter writer = PdfWriter.getInstance(document, new
FileOutputStream(FILE));
                document.open();
                addTitlePage(document);
                addContent(document, writer);
                document.close();
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

The main methods from these 2 classes are:

public static JFreeChart generateBarChart(Bank b) –generates a bar chart for the bank, and shows the ammount of time the account is still valid

public static void writeChartToPDF(Document document, JFreeChart chart, **int** width, **int** height, PdfWriter writer) –puts the chart into pdf

public static void addTitlePage(Document document) –add title to the pdf page

public static void addContent(Document document, PdfWriter writer)—add content to the pdf page

public static void createTable(Paragraph subCatPart)—creates a table that will be later added to the pdf

public static void addEmptyLine(Paragraph paragraph, **int** number)- add a number of spaces into the pdf file.

3.6 Controllers

SerializableManager.java- the role of this class is to save the classes that implement the Serializable interface when the program ends, and to restore them when the application starts again. This class has no instance variables, but has 4 methods:

public Bank deserializeBank(): this method will take from a file input stream the contents and creates an object of type bank that is returned by the method

public void serializeBank(Bank b): this method will take as an argument the class bank that needs to be serialized, and write it's content to an output file stream

AdminFrameController.java- this class creates the frame AdminFrame that we created in the package View and adds ActionListener to the buttons to handle the events given. To do that, we create classes AddAccListener, DeleteAccListener, AddPersonListener, DeletePersonListener, GenerateReportsListener that implements ActionListener. This implies the implementation of the methos actionPerformed.

```
public AdminFrameController() {
frame.setAddAccActionListener(new AddButtonActionListener());
frame.setDelAccButtonActionListener(new DeleteButtonActionListener());
frame.setAddPersonButtonActionListener(new AddPersonButtonActionListener());
frame.setDeletePersonActionListener(new DeletePersonButtonActionListener());
frame.setGenerateReportButtonActionListener(new GenerateReButtonActionListener());
this.bank=manager.deserializeBank();
}
```

UserFrameController.java- this class creates the frame UserFrame that we created in the package View and adds ActionListener to the buttons to handle the events given.To do that, we create classes AddMoneyListener, WithDrawMoneyListener,GenerateReportListener that implements ActionListener. This implies the implementation of the methos actionPerformed.

```
public UserFrameController() {
frame.setAddMoneyButtonActionListener(new AddMoneyButtonActionListener());
frame.setWithDrawMoneyButtonActionListener(new
WithDrawMoneyButtonActionListener());
frame.setGenerateReportButtonActionListener(new
GenerateReportButtonActionListener());

this.bank = manager.deserializeWharehouse();
}
```

MainController.java

The most important thing from this class is the main function that calls the function of creating a new login window, from where the application starts:

```
public static void main(String[] args) {
                    LoginFrame p = new LoginFrame();
                    p.createLoginWindow();
}
```

4.Using and testing the application

In order to test the main algorithms in the application, Junit for testing was used such that operations like addPerson(), deletePerson(),addAcountToHolder(),deleteAcountFromHolder() were tested, to function in a proper way.

5. Results

The application is an user friendly and useful application to perform basic management operations such as: addind a sum of money, withdraw money, deleting a person/account, add a person/account, generate reports, see details. Output results of the application are actually the results of operations performed on the list of accounts and list of persons.The information and the data are displayed in a table, as can be seen from the picture that shows the graphical user

interface.The results of every action that it is done upon the accounts/persons can be seen in the tables. Even though being limited, this application can be considered a good and helpful tool in understanding the operations that stand behind an account management system.

6. Conclusion and future developments

Achieving such a program may be hard both in terms of algorithms, graphical structure. The best is to represent the customers and products as a tree type structure because this kind of structure makes it easier for some operations to be done: add, remove or search for an element from the structure.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made by the user or the admin. Also, this project could be implemented with a data base in the backround, to be more close to the reality and usefulness of this kind of applications.

7. Bibliography

http://stackoverflow.com/
http://www.tutorialspoint.com/java/java_serialization.htm
http://www.programcreek.com/2013/09/java-reflection-tutorial/
http://www.javacodegeeks.com/2013/09/java-reflection-tutorial.html
http://tutorials.jenkov.com/java-reflection/index.html
http://javarevisited.blogspot.ro/2011/02/how-hashmap-works-in-java.html
http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html
http://javarevisited.blogspot.ro/2012/01/what-is-assertion-in-java-java.html
http://stackoverflow.com/questions/11415160/how-to-enable-the-java-keywordassert-in-eclipse-program-wise*E/*