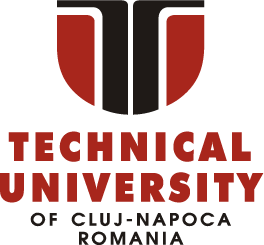
**TECHNICAL UNIVERSITY OF CLUJ-NAPOCA**

2nd year of study, Computer Science



Laboratory Work – Assignment 4

Bank Management

Lecturer: Prof. Dr. Eng. Ioan Salomie

Teaching Assistant: Delia Balaj

Student: Diana-Gabriela Popescu

Group: 30425

**Table of contents**

**1. Introduction**

1.1 Task objectives . . . . . . . . . . . . . . . . . . . . . . . . . .page 3

1.2 Personal approach . . . . . . . . . . . . . . . . . . . . . . . .page 3

**2. Problem description**

2.1 Problem analysis . . . . . . . . . . . . . . . . . . . . . . . . .page 4

2.2 Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

2.3 Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

2.4 Use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

**3. Projection**

3.1 UML diagrams . . . . . . . . . . . . . . . . . . . . . . . . . . page 8

3.2 Data structures . . . . . . . . . . . . . . . . . . . . . . . . . .page11

3.3 Class projections . . . . . . . . . . . . . . . . . . . . . . . .page 12

3.4 Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 17

3.5 Relationships . . . . . . . . . . . . . . . . . . . . . . . . . . . page17

3.6 Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 18

3.7 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 18

3.8 Graphical User Interface . . . . . . . . . . . . . . . . . . page18

**4. Implementation and testing** . . . . . . . . . . . . . . . . . . . . . . . .page 19

**5. Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**6. Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**7. Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**1. Introduction**

1.1 Task objectives

The task of the assignment is defined as it follows: “Consider an application BankManagement for processing customer and admin operations. 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. Define and implement the classes Person, Account, SavingAccount and SpendingAccount. Other classes may be added as needed (give reasons for the new added classes). An Observer DP will be defined and implemented. It will notify the account main holder about any account related operation. Implement the class Bank using a predefined collection which uses a hashtable. The hashtable key will be generated based on the account main holder. A person may act as main holder for many accounts. Use JTable to display Bank related information. Define a method of type “well formed” for the class Bank. Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions). Implement a test driver for the system. The account data for populating the Bank object will be loaded/saved from/to a file”. In other words, one should create a user interactive system (by means of console interaction or Graphical User Interface) that is able to manipulate bank accounts and holders and to perform basic operations on the available data.

1.2 Personal approach

This documentation paper aims to present a way of solving the problem of Bank Management. As for user interface, there will be developed algorithms for user to interact with the Graphical User Interface (GUI) and, for some methods, the output will also be displayed in the console. The solution is obtained by means of implementing several operations specific for bank accounts and holders such as: deposit, withdraw, adding a new account, removing an account, adding a new holder. This type of operations were chosen due to the fact that they are the most used and the most important operations regarding Bank Management.

**2. Problem description**

2.1 Problem analysis

The analysis of a problem starts from examining the real model of Order Management 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. Always, having a good model will ease the way the operations are performed and will make more complex programs be clear to read and easy to maintain. My implementation of the solution started with trying to find a suitable and clear model for bank accounts and holders. Also, due to the fact that the user has an important role in running the application, the Graphical User Interface was also been given a high importance in the process of problem analysis.

So, first of all, one has to analyse the real model of bank accounts and holders, for the implemented model to be as close to the real one. The problem domain is defined by the real definition of a bank account or a holder.

To a bank holder, there is associated an ID and a name. One should see that the ID and the name must be unique strings, and that any description might be missing or unnecessary. Also, to a bank account must be associated an ID, the available money in the account and the type of the account. One should see that the ID must be a unique string for each holder ( i.e. must be unique regarding the other accounts of the same holder ). Also, the money field is of double type, since it can refer to real sum of money ( banknotes and coins ). The type can be either “SAVING” or “SPENDING”.

This idea will also be developed when implementing the constructor of the classes. Furthermore, it is important to be mentioned that the bank contains a hash map of holders and their corresponding accounts.

2.2 Modeling

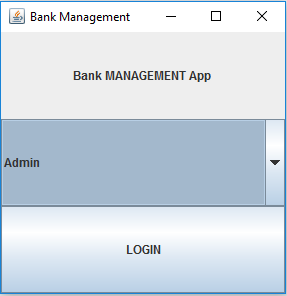
Based on the information presented above, I thought on the actual implementation of the solution. I started to think about what classes are required and how classes should be organized in packages. Also, I realized the need of having an abstract class for the customer / administrator frames, because of the possibility of them being similar.

2.3 Scenarios

The scenarios are as it follows: the user can choose to log in as administrator or as customer (or guest). Then, a new frame is open. Depending on the log in information, the user can choose from the available operations to be performed on the input products or orders, i.e. deposit, withdraw, adding a new account, removing an account, adding a new holder. The user has the possibility of changing the log in information by logging out and choosing the other option. If this happens, the operations will mold around the new inputs and available operations.

2.4 Use cases

The use cases are strongly related to the user running the application. Therefore, the user should choose to log in as administrator or as guest, i.e. customer.



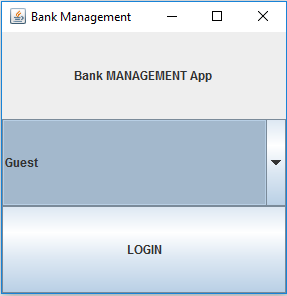


Illustration 1 : Graphical User Interface ( logging in )

Then, the user should choose the information that suits best to the situation. Then, depending on the operation desired to be done, the user can choose from the following: deposit, withdraw, adding a new account, removing an account, adding a new holder.

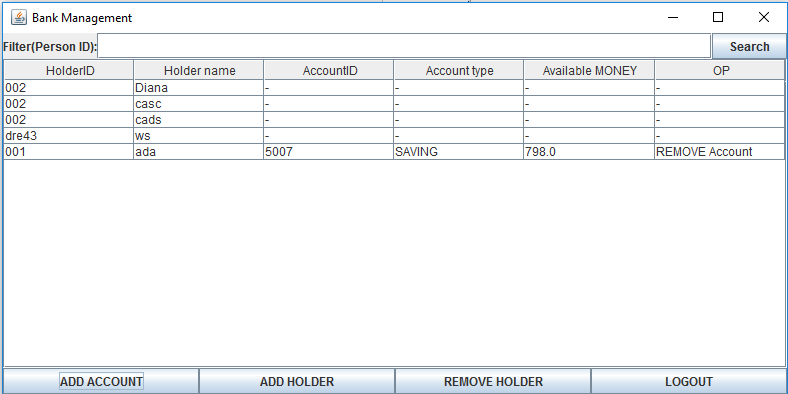
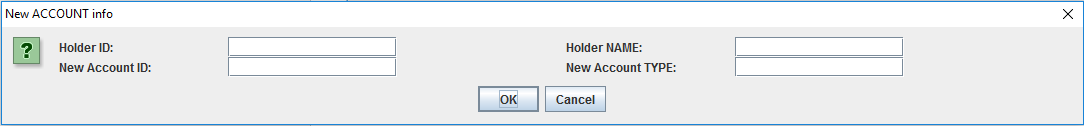


Illustration 2 : Graphical User Interface ( admin VIEW )



llustration 3 : Graphical User Interface ( admin VIEW – adding a new account )

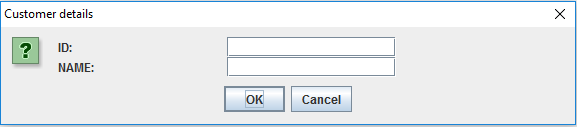


Illustration 4 : Graphical User Interface ( admin VIEW – adding / removing a holder )

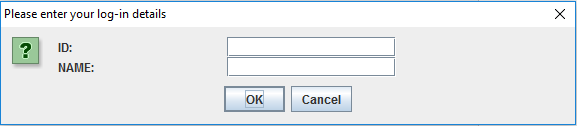


Illustration 5 : Graphical User Interface ( customer VIEW – introducing log – in details)

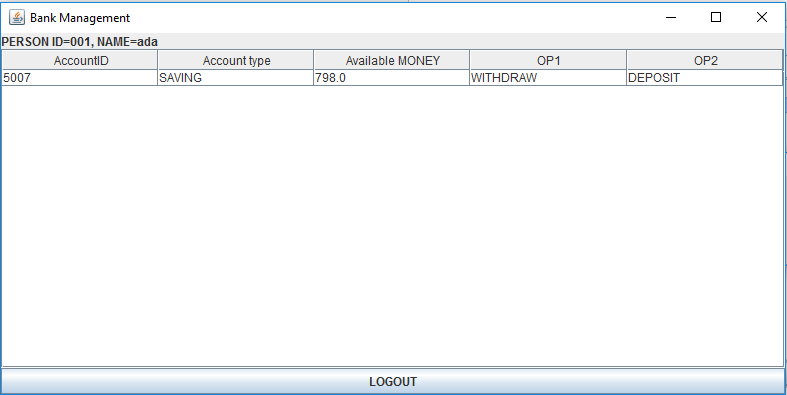


Illustration 6 : Graphical User Interface ( customer VIEW – account info)

To be mentioned is the fact that there are used a lot of interactive “JOptionPane” for requesting user – input. For example, when adding a new account, adding a new holder, when providing the log in information, when removing a holder, and so on.

**3. Projection**

3.1 UML diagrams

a) Use-case diagram

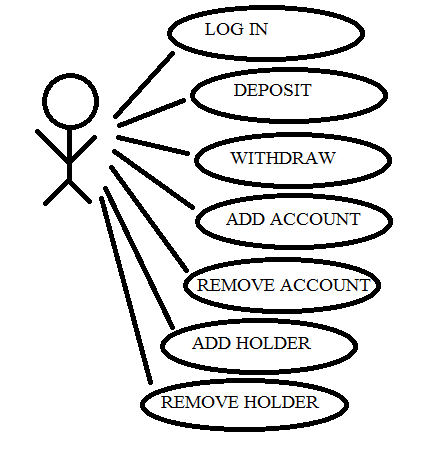


Illustration 7 : Use case diagram

The use case diagram presents the actor, which is the user that interacts with the application. He can perform several actions such as adding a new product, removing a product, modifying the characteristics of a product, placing an order, visualizing the placed orders.

b) Class Diagram

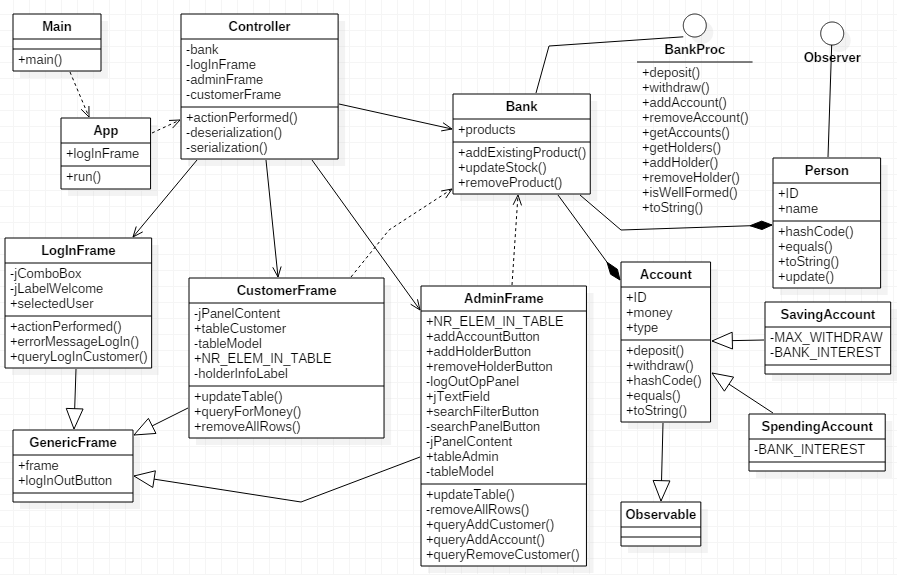


Illustration 8 : Class diagram

As presented in the class diagram, there are various relationships between the classes of the system. For example, there is one abstract class “GenericFrame”, which is extended by three classes: “LogInFrame”, “CustomerFrame” and “AdminFrame”. Between class “Bank” and classes “Person” and “Account” there is a composition relationship, just because a bank instance does not have sense without persons and accounts.

c) Sequence Diagrams

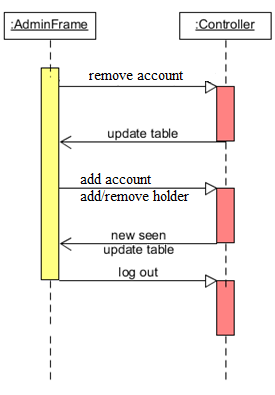


Illustration 9 : Sequence diagram #1

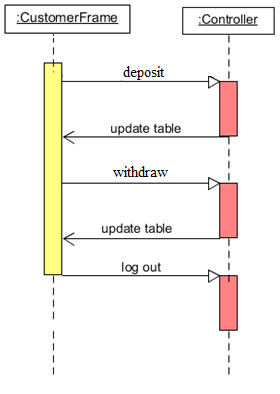


Illustration 10 : Sequence diagram #2

d) Activity Diagram

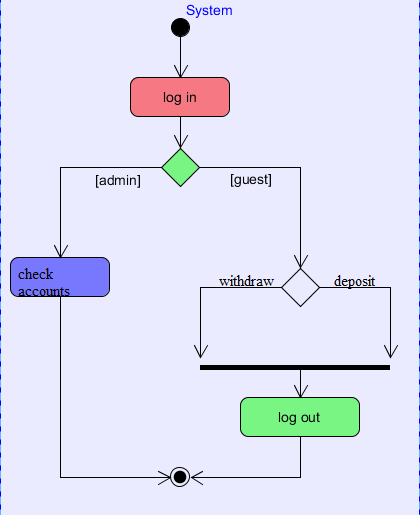


Illustration 11 : Activity diagram

3.2 Data structures

The data structures used in the application are either primitive data types ( int or double ) or new types that are defined by the designed classes ( for example, Person, Account, Account types ). Furthermore, there were used arrays of the types presented above. Not to be forgotten is the use of GUI-like types, such as: JFrame, JButton, JLabel, JPanel, JComboBox and JOPtionPane.

3.3 Class projections

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. For the beginning, it has to be mentioned that, in my design, I used four different packages for organizing the classes and the interfaces. These are: “IO”, “models”, “main” and “test”. The mentioned packages are to be described next, along with the classes belonging to them.

1. Package “IO” – contains the design related to the user interface (GUI and console).

* class **GenericFrame** – abstract class to be extended by the others frame – like classes. There are plenty of other instance variables, as described below:
  + public JFrame frame; -the specific frame.
  + public JButon logInOutButton; -for going to the next frame.

The constructor of the class instantiates and initializes the instance variables to the default values.

* class **AdminFrame** - extending GenericFrame. There are plenty of other instance variables, as described below:
* **public** **static** **int** *NR\_ELEM\_IN\_TABLE*;
* **private** JPanel logOutOpPanel;
* **public** JTextField jTextField;
* **public** JButton searchFilterButton;
* **private** JPanel searchPanelButton;
* **private** JScrollPane jPanelContent;
* **public** JTable tableAdmin;
* **private** DefaultTableModel tableModel;
* **public** JButton addAccountButton;
* **public** JButton addHolderButton;
* **public** JButton removeHolderButton;

The constructor of the class instantiates and initializes the instance variables to the default values.

The implemented methods are:

* **public** **void** updateTable(Bank bank)
* **public** **void** updateTable(Bank bank, String searchForID)
* **private** **void** removeAllRows()
* **public** Person queryAddCustomer(Bank bank)
* **public** **boolean** queryAddAccount(Bank bank)
* **public** **boolean** queryRemoveCustomer(Bank bank)
* class **CustomerFrame** - extending GenericFrame. There are plenty of other instance variables, as described below:
* **public** **static** **int** *NR\_ELEM\_IN\_TABLE*;
* **private** JLabel holderInfoLabel;
* **private** JScrollPane jPanelContent;
* **public** JTable tableCustomer;
* **private** DefaultTableModel tableModel;

The constructor of the class instantiates and initializes the instance variables to the default values.

The described methods are:

* **public** **void** updateTable(Set<Account> accounts)
* **public** **double** queryForMoney()
* **public** **void** removeAllRows()
* class **LogInFrame** - extending GenericFrame. There are plenty of other instance variables, as described below:
* **public** **static** **final** String ***GUEST*** = "Guest";
* **public** **static** **final** String ***ADMIN*** = "Admin";
* **private** JComboBox<String> jComboBox;
* **private** JLabel jLabelWelcome;
* **public** String selectedUser = ***ADMIN***;

The constructor of the class instantiates and initializes the instance variables to the default values.

The described methods are:

* **public** **void** actionPerformed(ActionEvent event)
* **public** **void** errorMessageLogIn()
* **public** Person queryLogInCustomer(Bank bank)

1. Package “models”

* interface **BankProc** - describes the capabilities of the bank. The methods are:
* **public** **void** deposit(**double** sum, String accountID, Person person);
* **public** **void** withdraw(**double** sum, String accountID, Person person);
* **public** **void** addAccount(Person person, Account account);
* **public** **void** removeAccount(Person person, Account account);
* **public** Set<Account> getAccounts(Person person);
* **public** Set<Person> getHolders();
* **public** **void** addHolder(Person p);
* **public** **void** removeHolder(Person p);
* **public** **boolean** isWellFormed();
* **public** String toString();
* class **Bank** -implementing BankProc and Serializable interfaces. There are two instance variables, as described below:
* **private** **static** **final** **long** ***serialVersionUID*** = 1L;
* **public** Map<Person, Set<Account>> bankMap;

The constructor of the class instantiates and initializes the instance variables to the default values.

The class implements the methods described by the “BankProc” interface and some other:

* **public** **void** deposit(**double** sum, String accountID, Person person);
* **public** **void** withdraw(**double** sum, String accountID, Person person);
* **public** **void** addAccount(Person person, Account account);
* **public** **void** removeAccount(Person person, Account account);
* **public** Set<Account> getAccounts(Person person);
* **public** Set<Person> getHolders();
* **public** **void** addHolder(Person p);
* **public** **void** removeHolder(Person p);
* **public** **boolean** isWellFormed();
* **public** String toString();
* **public** **boolean** equals(Object obj)
* **public** **int** hashCode()
* class **Person** - implementing Observer and Serializable. It contains one and only three instance variables:
* **private** **static** **final** **long** ***serialVersionUID*** = 1L;
* **public** String ID;
* **public** String name;

The constructor of the class instantiates and initializes the instance variables.

The class implements the methods:

* **public** **int** hashCode()
* **public** **boolean** equals(Object obj)
* **public** String toString()
* **public** **void** update(Observable arg0, Object arg1)
* class **SavingAccount** - **extending** Account implementing Serializable. It contains one and only two instance variables:
  + - **private** **static** **final** **long** ***serialVersionUID*** = 1L;
* **private** **static** **final** **double** ***BANK\_INTEREST*** = 0.5 / 100;

The constructor of the class instantiates and initializes the instance variables.

The class implements the methods:

* **public** **void** deposit(**double** money)
* **public** **boolean** withdraw(**double** money)
* class **SpendingAccount** - **extending** Account implementing Serializable. It contains one and only two instance variables:
  + - **private** **static** **final** **long** ***serialVersionUID*** = 1L;
* **private** **static** **final** **double** ***BANK\_INTEREST*** = 0.1 / 100;

The constructor of the class instantiates and initializes the instance variables.

The class implements the methods:

* **public** **void** deposit(**double** money)
* **public** **boolean** withdraw(**double** money)

1. package “main”

* class **Controller** -implementing ActionListener describes the relationships between the GUI interface and the logical structures of the application.

The instance variables are:

* **private** **static** **final** **int** ***LOG\_IN*** = 0;
* **private** **static** **final** **int** ***ADMIN*** = 1;
* **private** **static** **final** **int** ***GUEST*** = 2;
* **private** **int** CURRENT\_FRAME = ***LOG\_IN***;
* **private** Bank bank;
* **private** LogInFrame logInFrame;
* **private** AdminFrame adminFrame;
* **private** CustomerFrame customerFrame;

The constructor of the class instantiates and initializes the instance variables to the default values.

Implemented methods:

* + - public void actionPerformed(ActionEvent event) - for dealing with the events that occur when interacting with the GUI.
    - **private** **void** deserialization()
* **private** **void** serialization()
* class **App** - implementing Runnable describes the way the application runs. It has a “logInFrame instance” variable:
* private LogInFrame logInFrame;

The constructor of the class instantiates and initializes the instance variable to the given value.

Then, it instantiates a Controller object by calling its constructor with the instantiated “logInFrame” as parameter, all this in one method:

* public void run()
* class **Main** - contains the main method. It runs the App.

**public** **static** **void** main(String[] args) {

**new** App().run();

}

1. package “test”

* class **Test** – is where the testing of the available methods.

The instance variables are:

* **private** Bank bank;
* **private** Person person;
* **private** Account account;

The constructor of the class instantiates and initializes the instance variables to the default values.

Implemented methods:

* **public** **void** deposit(**double** sum, String accountID, Person person);
* **public** **void** withdraw(**double** sum, String accountID, Person person);
* **public** **void** addAccount(Person person, Account account);
* **public** **void** removeAccount(Person person, Account account);
* **public** **void** addHolder(Person p);
* **public** **void** removeHolder(Person p);

3.4 Interface

This section being already developed in detail until now, I will remember briefly some important facts. The user interface is mainly realized by means of “java.swing” package. I used instances of JFrame, JPanel, JButon, JLabel. Also, I used FlowLayout and GridLayout for organizing components inside the frame and panels. The buttons and the selection of a cell in the table are the main way of the user interacting with the application.

3.5 Relationships

As presented in the class diagram, there are various relationships between the classes and interfaces of the system. For example, there is one interface “BankProc”, which is implemented by “Bank”. Between class “Bank” and classes “Person” and “Account” there is a composition relationship, just because a bank instance does not have sense without holders and accounts.

There are also some other dependencies and associations due to the fact that some methods use as parameter types of a different class.

3.6 Packages

The program is divided in four packages, as mentioned before. The package “IO” contains classes: “GenericFrame”, “LogInFrame”, “AdminFrame” and “CustomerFrame”. The package “models” contains classes: “Bank”, “Person”, “Account”, “SavingAccount”, “SpendingAccount” and the interface “BankProc”. The package “test” contains class “Test”. The package “main” contains classes: “Controller”, “App” and “Main”.

3.7 Algorithms

The main algorithms regarding bank, holders and accounts are the following: deposit, withdraw, adding a new account, removing an account, adding a new holder. They are designed as methods handling holders and accounts, as it follows:

* **public** **void** deposit(**double** sum, String accountID, Person person);
* **public** **void** withdraw(**double** sum, String accountID, Person person);
* **public** **void** addAccount(Person person, Account account);
* **public** **void** removeAccount(Person person, Account account);
* **public** **void** addHolder(Person p);
* **public** **void** removeHolder(Person p);

3.8 Graphical User Interface

An important fact to be mentioned is the use of “java.swing” package. I used instances of JFrame, JPanel, JButoon, JSlider, BasicArrowButton, JLabel. Also, I used FlowLayout and GridLayout for organizing components inside the frame and panels. The buttons and the sliders are the main way of the user interacting with the application.

**4. Implementation and testing**

The implementation was done in Eclipse and it was also 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 algorithm. The personal touch in the implementation is also felt in the way the Graphical User Interface is thought.

Testing implies checking for any errors and warnings in the program or limitations of this program. A testing class was implemented. In this case, for each method, an exception is thrown, which can also indicate a possible error when running the app. So, from this point of view, there are lots of errors that will be avoided this way. Other possible scenarios will be tried as future development.

**5. Results**

The application is user friendly and useful in performing basic order/customer operations such as: deposit, withdraw, adding a new account, removing an account, adding a new holder. As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems. The application is to be used by anyone who is interested in performing these types of operations and has a basic knowledge about bank, holders and accounts.

**6. Conclusions**

The application can be further developed, by adding some other operations on bank, holders and accounts. For me personally, the design of this system helped me think about the way classes should be organized in packages.

**7. Bibliography**

* <http://staruml.io/>

-for downloading a tool for creating class diagrams.

* <http://bellekens.com/2010/12/20/uml-composition-vs-aggregation-vs-association/>

-for better understanding of the relationships between classes.