Technical University of Cluj-Napoca

Year 2016-2017

Department of Computer Science

Programming Techniques

4th Assignment

Student: Gyarmathy Tímea

Group: 30423

Table of Contents

[Specification of the problem 3](#_Toc482351498)

[Description 3](#_Toc482351499)

[The analysis of the problem 3](#_Toc482351500)

[Modeling 4](#_Toc482351501)

[The hashtable 4](#_Toc482351502)

[The application 4](#_Toc482351503)

[Use cases 4](#_Toc482351504)

[Scenarios 5](#_Toc482351505)

[Design 6](#_Toc482351506)

[Design concepts 6](#_Toc482351507)

[Package diagram 8](#_Toc482351508)

[Class diagram 8](#_Toc482351509)

[The Model 9](#_Toc482351510)

[The View 10](#_Toc482351511)

[The Controller 10](#_Toc482351512)

[Class Main 11](#_Toc482351513)

[Class TestBank 11](#_Toc482351514)

[Sequence diagram 12](#_Toc482351515)

[Graphical User Interface 12](#_Toc482351516)

[Implementation and testing 18](#_Toc482351517)

[Results 18](#_Toc482351518)

[Conclusions 18](#_Toc482351519)

[What have I learned? 18](#_Toc482351520)

[Further developments 18](#_Toc482351521)

[Bibliography 18](#_Toc482351522)

# Specification of the problem

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. Design 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. Design and implement a test driver for the system.

6. The account data for populating the Bank object will be loaded/saved from/to a file.



# Description

Modelling a bank is a common problem nowadays, as banks are the basis of the life in this century. People in a bank become a list of data to which accounts are associated, which can be used by this person. As the requirement specifies, the problem should implement an observer-observable design pattern, which is based on the fact that only the person should have access to its own accounts, for the security of money.

# The analysis of the problem

The specification explicitly requires the implementation and storing of data in a hashtable.

In [computing](https://en.wikipedia.org/wiki/Computing), a hash table (hash map) is a [data structure](https://en.wikipedia.org/wiki/Data_structure) which implements an [associative array](https://en.wikipedia.org/wiki/Associative_array) [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type), a structure that can map [keys](https://en.wikipedia.org/wiki/Unique_key) to [values](https://en.wikipedia.org/wiki/Value_(computer_science)). A hash table uses a [hash function](https://en.wikipedia.org/wiki/Hash_function) to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.

In order to implement the structure of a bank in a hashtable, I have used the Person as a key, and each table entry holds a Set of Accounts. This way these are associated to the Person given as a key.

To not lose data after the application is closed, the serializable design pattern is also implemented, and data is written into a file and read at the start.

# Modeling

## The hashtable

It is declared and mainly used in the implementation of the central class of the application, Bank. It maps Persons to Sets of Accounts. It is implemented using a HashMap. In Java, a HashMap is 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.

A Set is a collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

## The application

It presents the Observer design pattern, as well as the serializable one. The observer pattern is a [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in which an [object](https://en.wikipedia.org/wiki/Object_(computer_science)#Objects_in_object-oriented_programming), called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their [methods](https://en.wikipedia.org/wiki/Method_(computer_science)).

It is mainly used to implement distributed [event handling](https://en.wikipedia.org/wiki/Event_handling) systems, in "event driven" software. Java has built in "event" constructs which implement the observer pattern components, for easy programming and short code.

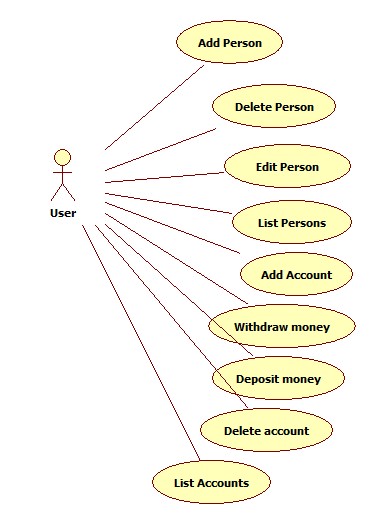
In this application, we’ll consider an observer a person who owns accounts. Only he has access to these, so only him should be notified whenever money is deposited or withdrawn from his accounts.

# Use cases

The user (actor) has to select the preferred action, insert the requested input if it is the case, and react to error messages if any.

When first opening the application, two options appear in front of the user: Person or Account operations. The user chooses according to its preference.

If Person was chosen, a series of available actions appear for the user, as seen in the use case diagram below. These actions each create their own specific frame which appears in front of the user once an action was chosen. For example if Add Person was chosen, the labeled input boxes appear, which have to be filled with the corresponding data. If Edit persons was chosen for example, an editable table appears with all the customers currently in the table, and every change upon their data will be saved in a serializable file. This will be read upon opening the application, and previously manipulated data will be available again.

If Accounts submenu was chosen, a list appears with all the registered persons in the bank and the user has to choose the person who accesses his accounts. After this, a submenu similar to the first described one will appear with options to add, delete or list accounts, each popping up its own specific frame upon selection. Another important option will be available to the user, the Withdraw/Deposit one, which controls the actions suggested by the name. These actions, when performed, notify the person doing them, and notifications will be printed to the console.

Even though accessable by the same user, we should take into consideration, that this is a bank processing system. As mentioned before, because the observer pattern was also implemented, only the account holder should be aware and to be able to access his/her accounts. The actions on the diagram are categorized into different submenus, the first four being grouped into the person operations, and the rest beind part of the account submenu.

Both submenus contains a Back button which redirects to the main menu.

## Scenarios

Precondition: The user successfully launches the application.

Success:

1. Main menu appears, two buttons can be pressed.
2. User presses the ‘Account operations’ button.
3. A list with the registered cuss appears.
4. User chooses a customer to place the order.
5. Submenu appears with the available account operations.
6. User selects ‘Add account’.
7. Add account frame appears, with the necessary input options.
8. User selects the type of account he wants to add, Saving/Spending account.
9. User inputs an amount of initial deposit which can be added to the account.
10. User clicks ‘Save’.
11. Frame closes, accounts submenu is visible.
12. User closes the application.

Alternative scenarios:

**First:** User clicks on the “Person operations” button in the main menu.

1. Submenu appears with the available actions upon the banks registered persons.
2. User selects desired action.
3. User completes desired action.
4. User clicks on “Back” button.
5. Return to step 1.

**Second:** User clicks on an empty field in the customers list.

1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User closes error window.
3. Return to step 4.

**Third:** User enters invalid or no input for desired money amount.

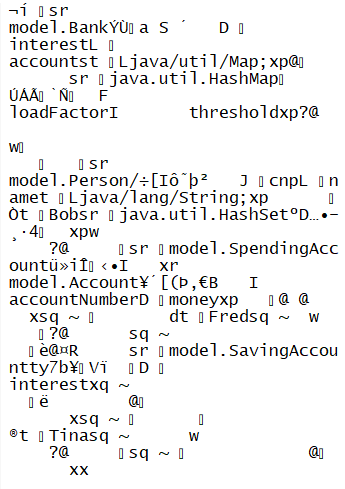
1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User closes error window.
3. Return to step 8.

# Design

## Design concepts

As mentioned before, design includes an observer design pattern. The main design, though, is based on the Model- View- Controller pattern. Thus, the application contains 3 packages, model, view and controller.

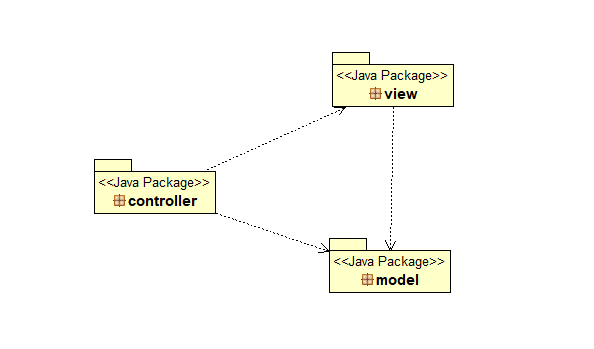
Besides the classes sorted into these packages, the application will also have a Main class in the default package which contains only the start instruction of the application in its main function.

In the package model, we’ll include the classes modelling the real-life concepts, namely the ones mentioned on the given diagram: Bank, Account, Person, and the interface for Bank, BankProc, and the two subclasses of Account. All these classes should implement the Serializable interface in orer to be able to save the bank data. Person should implement Observer and Account should extend Observable to realize the Observer design pattern.

The view should contain JTables to list data in the bank, on these, SelectionListeners were implemented to realize the functionality of the application.

The serializable file the application creates in not user readable, as it contains data written with the help of streams. Illustration abowe.

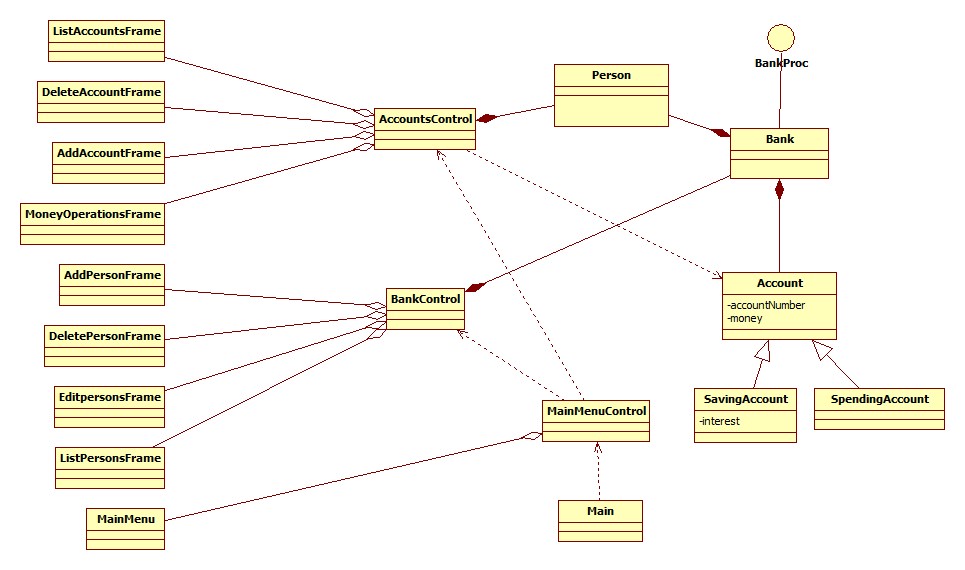
## Package diagram



Represents the basic model-view-controller design concept.

## Class diagram

Summarized class diagram (some non-relevant classes, and some getter, setter and helper methods were omitted for comprehension):



### The Model

#### Class Person

Models relevant information on a single person: his CNP and his name. The class contains getters and setters for these properties, as well as various constructors. Its fields are in accordance to the table in the database. It implements Observer, thus it overrides the update(Observable o, Object arg)method. This will be called whenever the Observable object calls the notifyAll() method.

The instances of this class are used as keys in the hashtable, so method hashCode can also be found overridden here, as well as the method equals().

#### Class Account

Models relevant information on an account: unique ID, amount of money it holds. The class contains getters and setters for these properties, as well as various constructors. It extends Observable, thus being associated to an Observer. It is a superclass, and its subclasses inherit its methods, such as the withdraw method used to retrieve an amount of money from the account.

#### Class SpendingAccount

Extends Account and besides specifying the specific constructors, it contains no additional methods.

#### Class SavingAccount

Extends Account and also implements specific methods. A savings account means that only one deposit is allowed and only one withdrawal, when all the amount of money should be retrieved as well as the additional interest that was calculated upon the deposit time.

#### Interface BankProc

This is the interface Bank is based upon, it declares and describes the methods the latter should contain. In its comments, pre and post conditions are specified verbally that these methods should take into account. These methods refer to operations such as adding, removing or retrieving person or account data, withdrawing or depositing money.

#### Class Bank

The basis of the application. It implements all the functional methods the application is based upon, thus realizing the model of a real like bank. It holds the instance of the hashmap, in which the data are stored, its declaration looks like the following:

**private** Map<Person, HashSet<Account>> accounts;

All the methods within this class are based upon this data structure.

Pre and post conditions are specified in the methods, by using the assert command. For example when adding a person, the cnp should be valid, a Boolean type method is called to check it:

**assert** validateCnp(person.getCnp()) : "cnp must be valid";

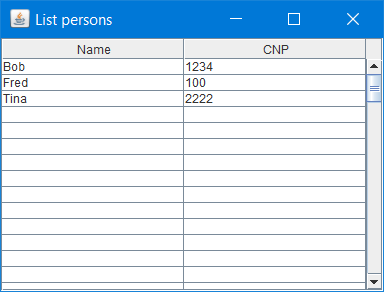
If the assert condition returns false, the message is thrown as an AssertError type objects argument.

Besides all the methods referring to persons, accounts and helper functions, this class also implements the two functions that work on a serial file: writing and reading it. Thus, by using streams, the object bank is entirely written into a file called “myBank.txt”, which is read when starting the application.

It also specifies a well-formed method which states that the bank is well formed if it is not empty.

### The View

It contains several classes which implement the user interface of the application. The main class in this package is MainMenu, as this contains all the buttons which have as functionality displaying all the other frames, as required. For example, ListPersonsFrame appears upon clicking on List persons button in the Person operations submenu and a window with the following format appears:



This is a JTable containing uneditable cells. Its content is set through the method:

**public** **void** setList(List<Person> personsList){

Implemented in class ListPersonsFrame.

### The Controller

#### Class MainMenuControl

As the name suggests, it controls the main menu from the user interface. The whole application is started by calling its static start() method which instantiates the control. It implements in its inner classes the listeners regarding the buttons of the application, and within these, the corresponding methods are called. The whole application is divided into two functional parts: operations regarding the accounts and those regarding the persons. According to this, the static methods are called from the controller regarding the accounts or the persons, accordingly.

#### Class BankControl

It implements the actions regarding the persons submenu, by connecting the view with the model and calling the methods according to the user input. The listeners based on the application buttons have their functionality implemented within this class.

#### Class AccountsControl

Similarly to BankControl, this class implements the functionality of the buttons in the accounts submenu, by defining inner classes which implement the corresponding Listener interface.

### Class Main

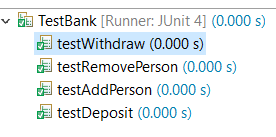
It is in the default package and contains the main method of the application, with a single line in it:

MainMenuControl.*start*();

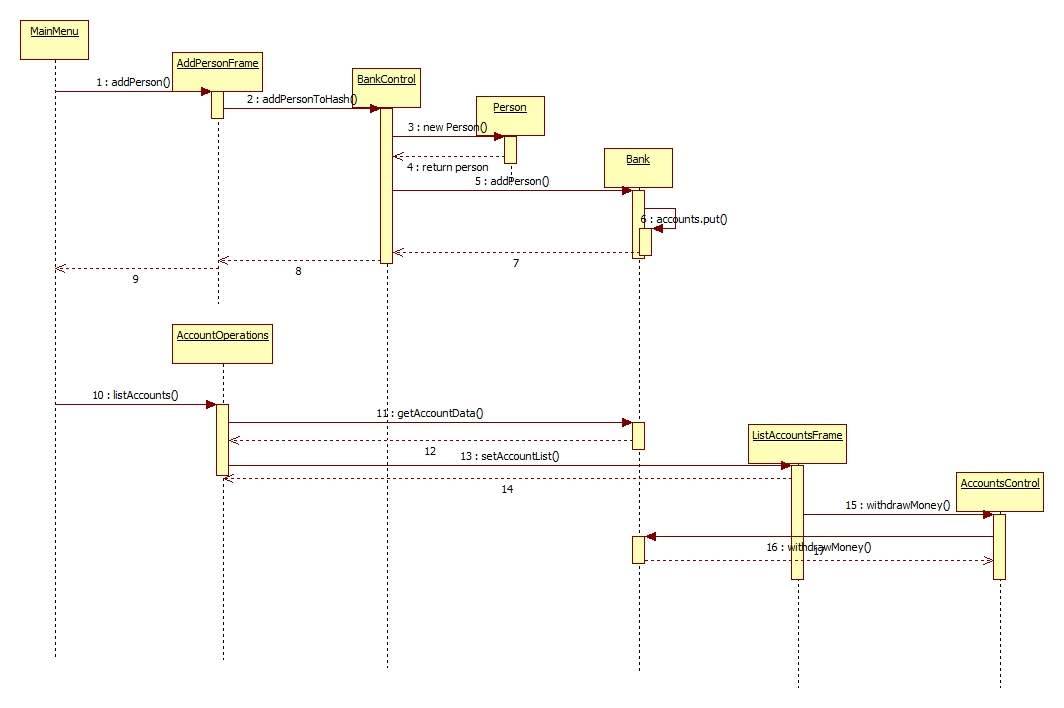
Which instantiates the control and the user interface.

In the commented section of the main function there are some hardcoded inserts in the database, which are useful when the empty database is created or re-created and the application tester does not want to insert data one-by-one, so he just needs to decomment this section before running the application.

### Class TestBank

It is a Junit test case, it tests the functionality of 4 methods from class Bank: addPerson, removePerson, withdraw and deposit. It uses the principle of golden data to test the adding of a person, which means it compares the given result with one that is supposed to be the result. In case they are not equal, the test fails. The other methods are tested by using mathematical verifications and comparisons. 

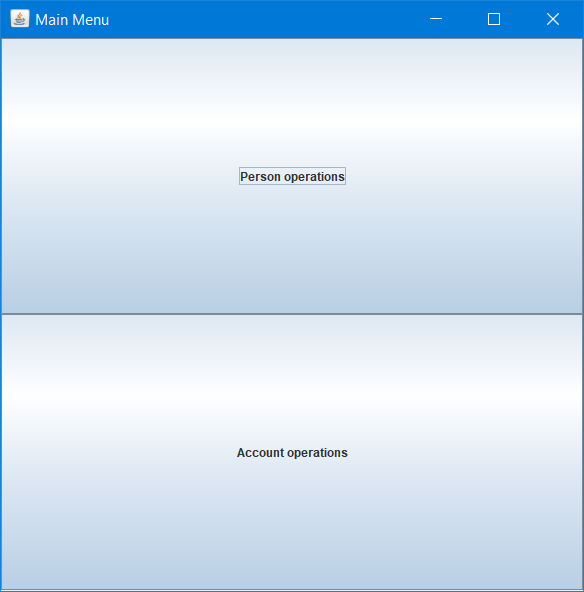
## Sequence diagram



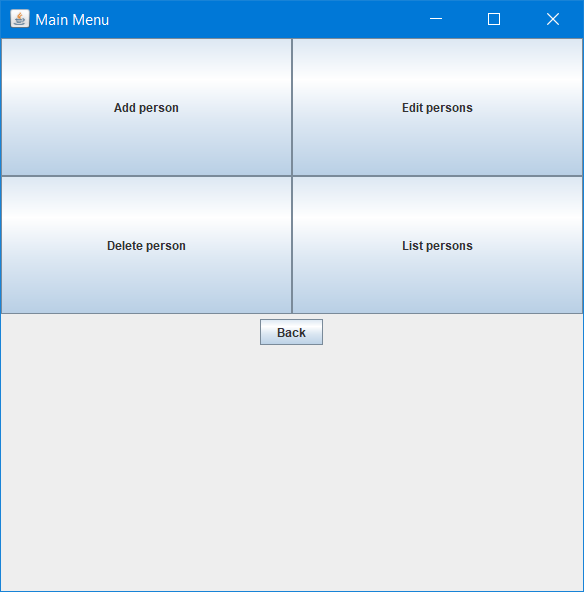
I’ve taken as example the sequence diagram for when the operation Add Person was selected, it creates a chain reaction and all the components of the system are instantiated accordingly.

## Graphical User Interface

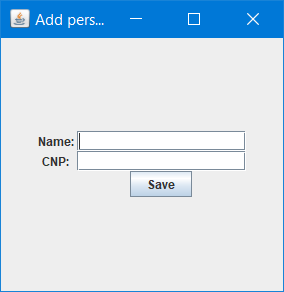
It is user friendly, the user is presented with intuitive actions to be performed. When first opened, the application shows the main menu:



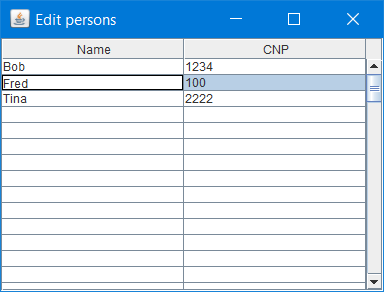
If the user chooses to perform actions regarding the person data stored in the bank, when he presses the “Person operations” button, the following submenu appears:



Each of these buttons is functional and creates a separate action frame when pressed. This way, for example, pressing the “Add person” button pops up the following window:

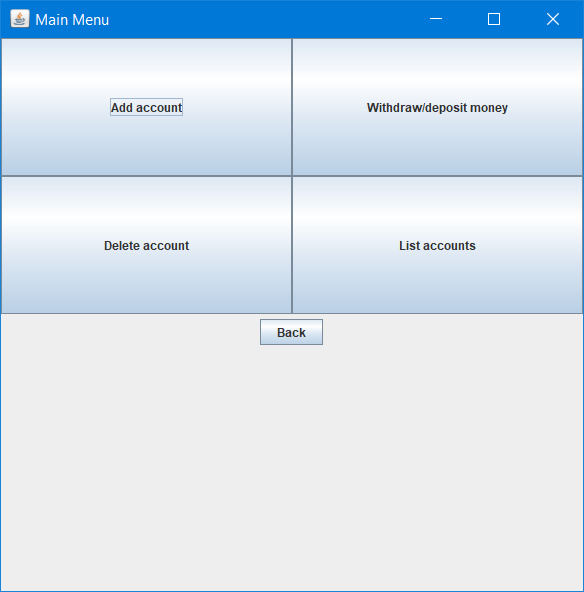


Or, pressing the “Edit persons” button pops up a window with an editable JTable of the following structure:

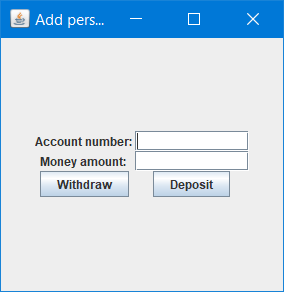


The listed persons are the ones stored in the hashtable, in the ‘bank’.

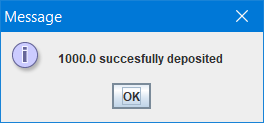
Pressing the “Back” button redirects to the main menu, where if the user chooses to press the Account operations button, thus engaging in being the user which accesses its own accounts, a list with the registered persons pops up. There, he will have to click on one to enter into his specific accounts data. When the person was selected, the following layout appears:



Which is very similar to the submenu of the person operations, except here instead of the edit button we have the Withdraw/Deposit which calls a frame of the following structure:



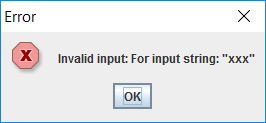
As it can be observed, the operations are based on account number which can be checked from listing all the accounts of the person. Peforming one of these two operations successfully pops up an information message that the operation was executed:



Also, because of the Observer design pattern, the person gets notified that his/her account was accessed so we get the following message in the console:

Notifying Client: Tina, acc number: 2, message: 1000.0 value has been added to the account

Whenever an error occurs during the usage of the application, for example if bad data format was input, an error pop-up window appears which informs the user of the error. For example:



# Implementation and testing

Implementation started from modeling the problem and implementing the given class diagram, so I first implemented the structure of the Model package. Alongside with this, I also created the required design patterns.

During implementation I have verified the correctness of my code by analyzing the log events that I have displayed in the console, and by writing a Junit test case for the most important methods. Whenever an unexpected error occurred, I have used Eclipse’s debugger to find the source of the problem and correct it.

Building the user interface took a lot of time and hard work, because all the actions had to be connected and implemented relatively to each other.

# Results

The development of this assignment resulted in a decent, user friendly application that illustrates in a professional manner the actions which can be taken upon a bank. The developed system of bank processing shows in an effective way how data can be stored, updated, retrieved and deleted in such a way that it could be reread later. This means, that the same data set is recreated whenever the application is restarted and saved upon exiting.

# Conclusions

## What have I learned?

With this assignment I have certainly learned the implementation of a hashtable. The techniques used in this project were also new to me, the Observer design pattern, the serializable classes, the pre and post conditions. I have also achieved knowledge about creating a more complex project, solving programming issues.

## Further developments

This application can be refined by defining security constraints, such as requiring a password when a user wants to access his accounts.

# Bibliography

1. For programming issues: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>
2. For more programming issues: <http://stackoverflow.com/>
3. Class diagram description: <https://en.wikipedia.org/wiki/Class_diagram>
4. Observer design pattern: <https://en.wikipedia.org/wiki/Observer_pattern>
5. Hash table: <https://en.wikipedia.org/wiki/Hash_table>
6. This pdf: <http://users.utcluj.ro/~crisb_pop/PT2017_L9.7z> (tp2017)
7. Last semester’s OOP code on how to use Swing to create user interface, and to realize the MVC.