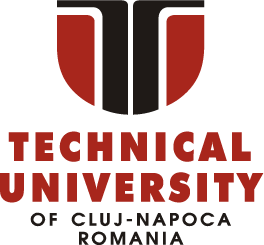
Technical University of Cluj-Napoca April, 2016

Programming Techniques

Laboratory - **HOMEWORK 4**

Bank accounting system

**

*Teacher*: Ioan SALOMIE *Student*: Iulia Maria ṢOMFELEAN

*Teacher Assistant*: Delia BALAJ Group 30425

1. Objective

*TP Lab–Homework 4*

***Objective***

*Design by Contract*

***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.*

1. Dimensions of the problem

**Bank system:** The structural network of institutions that offer financial services within a county. The members of the banking system and the functions they typically perform include: (1) commercial banks that take deposits and make loans, (2) investment banks which specialize in capital market issues and trading, and (3) national central banks that issue currency and set monetary policy.

* 1. Analyzing and modelling the problem

We have to design a bank accounts management system. From the economical point of view, there are many types of accounts, each of them with their own characteristics, differing from transaction fees to interest rates.

As our diagram says, we approach two types of account: **Saving** account and **Spending** account, and the operations that a bank/client can perform on them.

|  |  |  |
| --- | --- | --- |
| Characteristic | SPENDING ACCOUNT | SAVING ACCOUNT |
| ***ACTIVE\_ACCOUNT\_BALANCE\_LIMIT*** | 15 LEI | 5000 LEI |
| ***INTEREST\_RATE\_PERCENT*** |  | 8% |
| ***ACCOUNT\_TIME\_PERIOD*** | 5 years | 5 years |
| ***PROCESSING\_FEES\_PERCENT*** | 0.1% | 1% |

*Bank:*

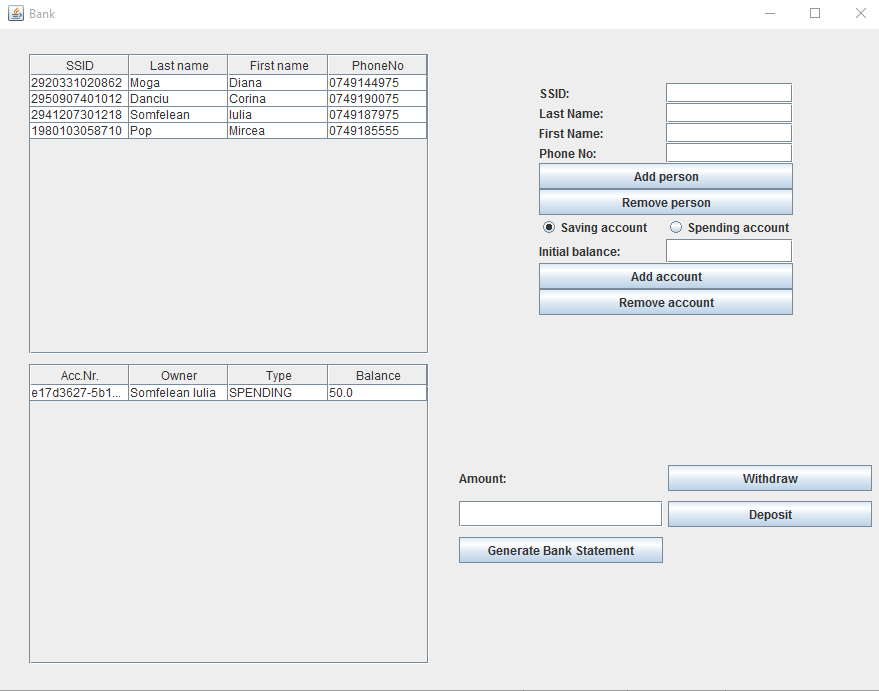
* Add person
* Remove person
* Add account
* Remove account
* Read accounts data
* Write accounts data
* Generate statements

*Client:*

* Withdraw money
* Deposit money
* Get notified of changes in accounts belonging to them
  1. Scenarios and use cases

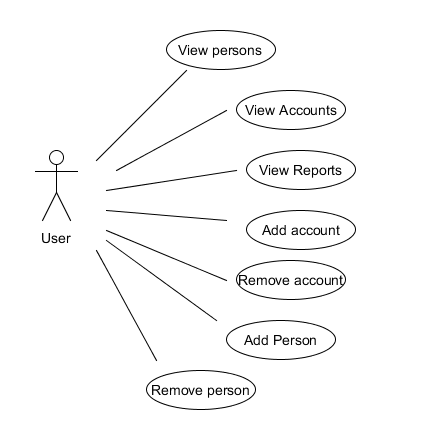
A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

The use cases are strongly related to the user steps. I tried to design my interface in a user friendly mode, and that’s the result:



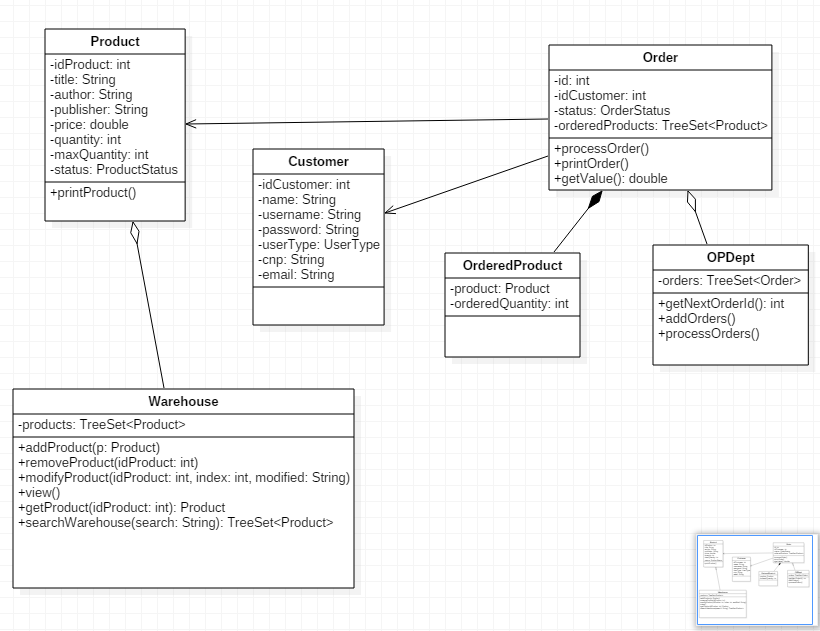
1. Implementation
   1. Diagrams
2. Use case diagrams

The use case diagram presents the actors, which in our case can play the role of *bank* or *regular user*(= client*).*



1. Class diagram

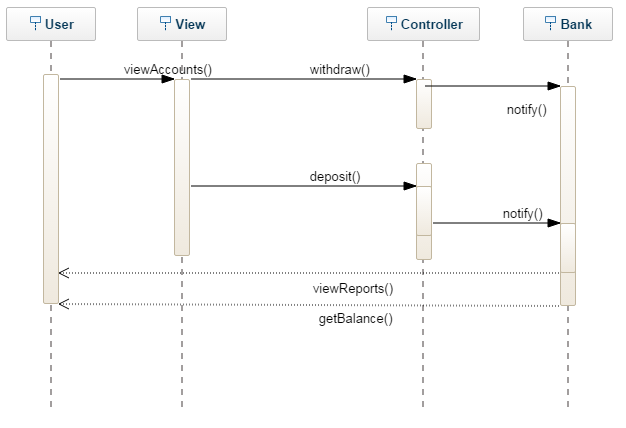
**Class diagram for the  *model* part**



1. Sequence diagram

A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called **event diagrams** or **event scenarios**.

A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.



1. Activity diagram

**Activity diagrams** are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control.

Activity diagrams are constructed from a limited number of shapes, connected with arrows.The most important shape types:

* *rounded rectangles* represent *actions*;
* *diamonds* represent *decisions*;
* *bars* represent the start (*split*) or end (*join*) of concurrent activities;
* a *black circle* represents the start (*initial state*) of the workflow;
* an *encircled black circle* represents the end (*final state*).

Arrows run from the start towards the end and represent the order in which activities happen.

* 1. Data Structures
     1. ArrayList

It’s a resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null.

In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

* + 1. Hashtable

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Below given is the list of constructors provided by the HashTable class.

|  |  |
| --- | --- |
| **Sr.No** | **Constructor and Description** |
| 1 | **Hashtable( )**  This is the default constructor of the hash table it instantiates the Hashtable class. |
| 2 | **Hashtable(int size)**  This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| 3 | **Hashtable(int size, float fillRatio)**  This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. |
| 4 | **Hashtable(Map<? extends K, ? extends V> t)**  This constructs a Hashtable with the given mappings. |
|  |  |

Apart from the methods defined by Map interface, Hashtable defines the following methods:

|  |  |
| --- | --- |
| **Sr.No** | **Methods with Description** |
| 1 | **void clear( )**  Resets and empties the hash table. |
| 2 | **Object clone( )**  Returns a duplicate of the invoking object. |
| 3 | **boolean contains(Object value)**  Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found. |
| 4 | **boolean containsKey(Object key)**  Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found. |
| 5 | **boolean containsValue(Object value)**  Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found. |
| 6 | **Enumeration elements( )**  Returns an enumeration of the values contained in the hash table. |
| 7 | **Object get(Object key)**  Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned. |
| 8 | **boolean isEmpty( )**  Returns true if the hash table is empty; returns false if it contains at least one key. |
| 9 | **Enumeration keys( )**  Returns an enumeration of the keys contained in the hash table. |
| 10 | **Object put(Object key, Object value)**  Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table. |
| 11 | **void rehash( )**  Increases the size of the hash table and rehashes all of its keys. |
| 12 | **Object remove(Object key)**  Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned. |
| 13 | **int size( )**  Returns the number of entries in the hash table. |
| 14 | **String toString( )**  Returns the string equivalent of a hash table. |

* 1. Designs patterns
     1. Observer pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class*Subject* that is extending class *Observer*.

*ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



* + 1. Design by contract

*The Design by Contract (DBC) software development technique ensures high-quality software by guaranteeing that every component of a system lives up to its expectations. As a developer using DBC, you specify component contracts as part of the component's interface. The contract specifies what that component expects of clients and what clients can expect of it.*

*Bertrand Meyer developed DBC as part of his Eiffel programming language. Regardless of its origin, DBC is a valuable design technique for all programming languages, including Java.*

*Central to DBC is the notion of an assertion -- a Boolean expression about the state of a software system. At runtime we evaluate the assertions at specific checkpoints during the system's execution. In a valid software system, all assertions evaluate to true. In other words, if any assertion evaluates to false, we consider the software system invalid or broken.*

*DBC's central notion somewhat relates to the #assert macro in C and C++ programming language. However DBC takes assertions a zillion levels further.*

*In DBC, we identify three different kinds of expressions:*

*Preconditions*

*Postconditions*

*Invariants*

*Let's examine each in more detail.*

*Preconditions*

*Preconditions specify conditions that must hold before a method can execute. As such, they are evaluated just before a method executes. Preconditions involve the system state and the arguments passed into the method.*

*Preconditions specify obligations that a client of a software component must meet before it may invoke a particular method of the component. If a precondition fails, a bug is in a software component's client.*

*Postconditions*

*In contrast, postconditions specify conditions that must hold after a method completes. Consequently, postconditions are executed after a method completes. Postconditions involve the old system state, the new system state, the method arguments, and the method's return value.*

*Postconditions specify guarantees that a software component makes to its clients. If a postcondition is violated, the software component has a bug.*

*Invariants*

*An invariant specifies a condition that must hold anytime a client could invoke an object's method. Invariants are defined as part of a class definition. In practice, invariants are evaluated anytime before and after a method on any class instance executes. A violation of an invariant may indicate a bug in either the client or the software component.*

*Assertions, inheritance, and interfaces*

*All assertions specified for a class and its methods apply to all subclasses as well. You can also specify assertions for interfaces. As such, all assertions of an interface must hold for all classes that implement the interface.*

* 1. Packages

Java packages help in organizing multiple modules and group together related classes and interfaces. Packages avoid name conflicts.

In object-oriented programming development, model-view-controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models. The MVC pattern is widely used in program development with programming languages such as Java, Smalltalk, C, and C++.

The MVC pattern has been heralded by many developers as a useful pattern for the reuse of object code and a pattern that allows them to significantly reduce the time it takes to develop applications with user interfaces.

The model-view-controller pattern proposes three main components or objects to be used in software development:

* A *Model* , which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
* A *View* , which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth)
* A *Controller* , which represents the classes connecting the model and the view, and is used to communicate between classes in the model and view.

Given the fact that I have chosen a Model-View-Controller Pattern, I splitted my classes into the corresponding packages and, alongside them, some useful classes :

* **model**: contains the “brain” of the project, the classes that model the problem.
* **view**: represents the GUI
* **controller**: the controller part interconnects the model and the view
* **utilities:** contains useful classes, such as a class where I keep all my constants, enumeration classes and the main one.

. . . . . . . . … . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* 1. Class Design

The whole idea of splitting your program into classes is based on a general rule named divide and conquer. This paradigm can be used almost everywhere: you divide a problem into smaller problems and then you solve these little, simple and well-known problems .  
Dividing your program into classes is one of the types of division which started to become common in last decade. In this programming paradigm we model our problem by some objects and try to solve the problem by sending messages between these objects.

I tried to design my project in the MVC architecture, that’s why I have 3 principal parts:

* + 1. The model – contains the logic of the application
* **Person**: contains data about the customer, such as name, email, ssid, phone number.
* **Account:** it’s an abstract class which contains account number and balance, withdraw, deposit methods, if an account is or not active/expired/mature.
* **SavingAccount:** subclass of Account, with specific characteristics
* **SpendingAccount:** subclass of Account, with specific characteristics
* **BankProc**: an interface that defines the procedures that a bank can perform: Add person

Remove person

Add account

Remove account

Read accounts data

Write accounts data

Generate statements

* **Bank:** it’s implementing BankProc and it’s methods
  + 1. The controller – contains the linking between the model and the view of the application.

Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

* **SerializationController:** it’s the class that deals with the serialization and deserialization of the Warehouse and OPDept.
* **Main:** contains the reference of a view and a controller in order to initialize and launch the bank application
* **BankController:** interconnects the view and model, has listeners that implement some logic and different methods to reflect changes on the view.
  + 1. The view – View represents the visualization of the data that model contains.
* **LoginView:** represents the login window that pops up at the starting of our application.
* **BankView:** represents the view for the main operations, view of the accounts balances, owners, persons in the bank, etc.

I used JTable for various operations:

The JTable is used to display and edit regular two-dimensional tables of cells.

The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set up easily.

JTables are typically placed inside of a JScrollPane. By default, a JTable will adjust its width such that a horizontal scrollbar is unnecessary. To allow for a horizontal scrollbar, invoke [setAutoResizeMode(int)](https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html" \l "setAutoResizeMode(int)) with AUTO\_RESIZE\_OFF. Note that if you wish to use a JTable in a standalone view (outside of a JScrollPane) and want the header displayed, you can get it using [getTableHeader()](https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html" \l "getTableHeader()) and display it separately.

* + 1. The utilities
* Constants: is a class which contains values used inside the code, such as commission fees, interest rates percentages, usually the values from this table, specific for each type of account:

|  |
| --- |
| Characteristic |
| ***ACTIVE\_ACCOUNT\_BALANCE\_LIMIT*** |
| ***INTEREST\_RATE\_PERCENT*** |
| ***ACCOUNT\_TIME\_PERIOD*** |
| ***PROCESSING\_FEES\_PERCENT*** |

* 1. Algorithms
     1. Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

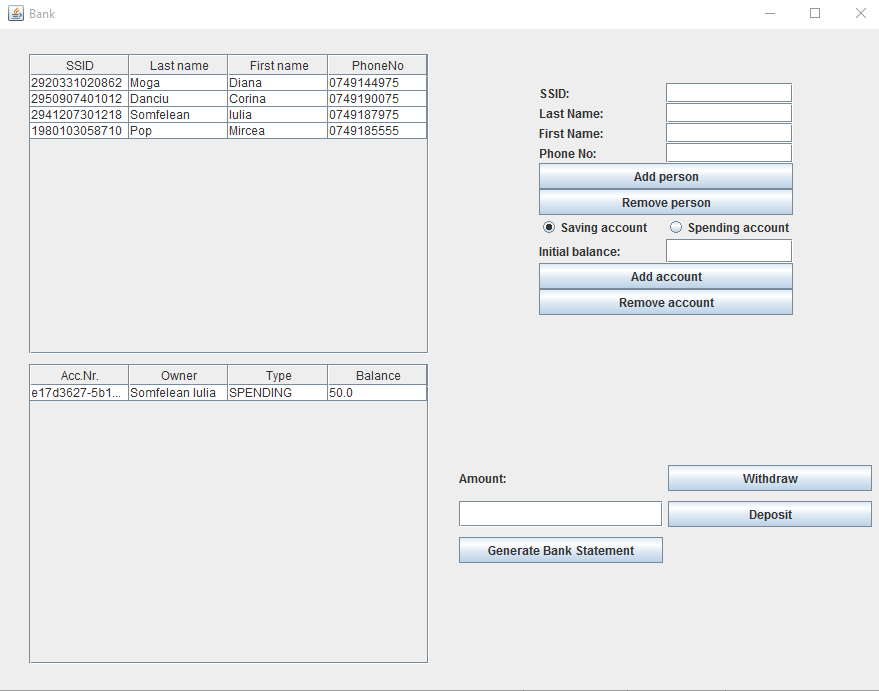
public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

I used serialization to store my warehouse and orders in external files, in order to maintain changes even if the application is closed.

* 1. User Interface

The user interface has the role of connecting the user with our application. He can perform different operations: see persons existing in the bank, see accounts associated with each person, see balance and ssid, withdraw or deposit money, etc.



1. Implementation and testing

This application was developed and tested only in Eclipse, but this thing should not affect it’s portability.

Testing was made using JUnit test on BankProc:

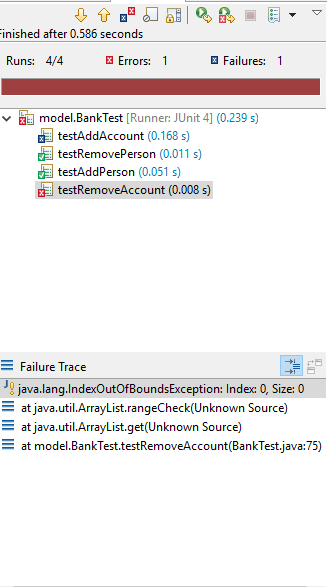
[JUnit](http://junit.org/) is an open source testing framework which is used to write and run repeatable automated tests, so that we can be ensured that our code works as expected. [JUnit](http://junit.org/) is widely used in industry and can be used as stand alone Java program (from the command line) or within an IDE such as Eclipse.

JUnit provides:

* Assertions for testing expected results.
* Test features for sharing common test data.
* Test suites for easily organizing and running tests.
* Graphical and textual test runners.

JUnit is used to test:

* an entire object
* part of an object – a method or some interacting methods
* interaction between several objects



1. Results and improvements

The application is an user friendly and helpful application to perform As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems (as long as they have the Java SDK installed).

As improvements, I would enumerate:

* Adding more persons and accounts associate
* Making some changes to the interface to improve aspect
* Using a database to store and retrieve the information

1. Conclusions

I learned how usefull Junits are, about assertions, about Design By Contract: preconditions, postconditions and invariants.

I’ve learned more about Observer pattern.

By the means of this project I managed to improve my knowledge about I found out how useful is to know how to use properly elements of Swing, such as JTable, JList, etc.

1. Bibliography
   1. Object-Oriented Programming - Lecture Slides of prof. Marius JOLDOS
   2. Programming Techniques – Lectures of prof. Ioan SALOMIE
   3. Head First Java 2nd Edition, Kathy SIERRA
   4. [www.stackoverflow.com](http://www.stackoverflow.com)
   5. <http://whatis.techtarget.com/definition/model-view-controller-MVC>
   6. wikipedia.org
   7. https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html