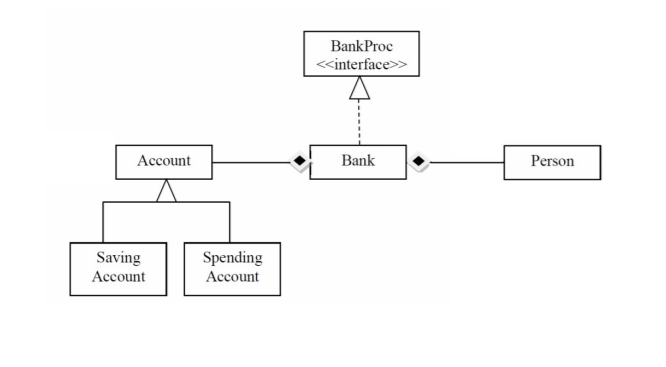
Programming Techniques

Homework 3

Bank Application

1. **Objective:**

Problem Specification:

Design and implement the following class diagram of a bank application:

The objective of this program is to implement a bank application based on the given class diagram, together with a graphic user interface. The data in the program is consistent, being available even after the program is closed and restarted. The user is able to manipulate the persons in the bank and their accounts, while the persons are notified of each change in their accounts.

1. **Analysis, Modelling, Use Cases, Scenarios:**

2.1. Analysis:

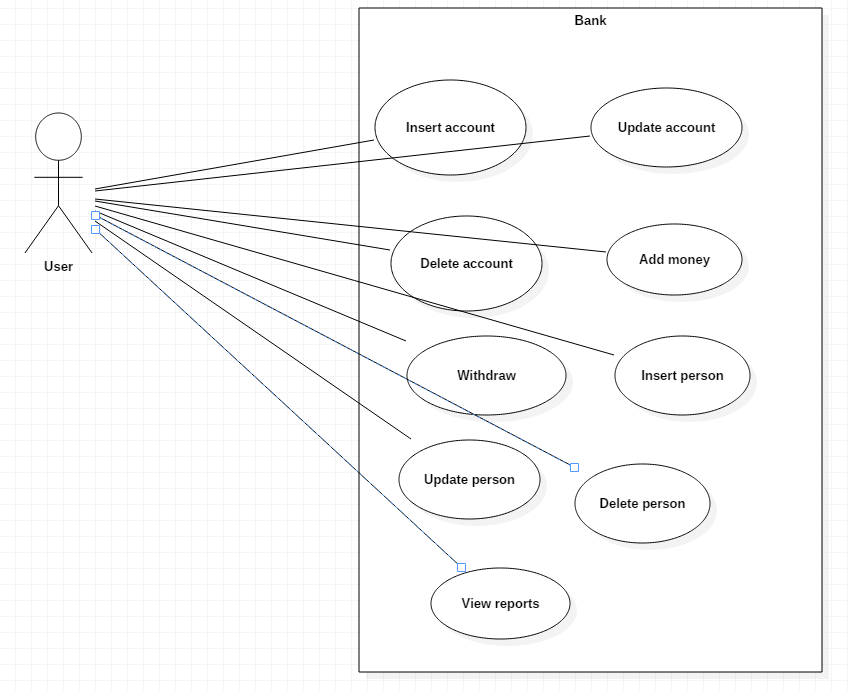
The program is a bank management application. The user can add, remove or edit persons (the bank’s clients), add, remove or edit accounts of persons and also perform account operations like add money and withdraw money. Bank statistics are available at any moment, and the persons are kept informed of any operation on their accounts. The data in the bank is not lost at program exit, but it is saved in files and ready to be accessed again when the program is restarted.

2.2. Modelling:

The approach I have chosen for modelling this problem involves implementing the given class diagram and adding additional functional classes in order to round up the application. Thus, the bank is comprised of persons and accounts, while accounts cand be of two types: saving account (which allow one deposit, one withdrawal and compute interest) and spending account (any number of deposits and withdrawals, no interest). The bank is manipulated by a model class that is used to translate input data into cohesive objects taken by the bank. The model is further used together with other classes to create the graphic user interface.

2.3. Use cases:

The main (and only) actor in the for this program is the user, as per the definition of the use case. The user has all the capabilities of the program at hand, being able to either work with the accounts or with the persons. Interaction between the user and the sysytem is done via the GUI, by clicking buttons and tables and writing the data parameters in the text fields. The use cases are better illustrated by the following use case diagram:



2.4. Scenarios:

Scenarios are sequences of steps that represent the interaction between the system and an actor.

**Scenario 1**

1. Identification Summary

Title: Add a new person.

Summary: This use case allows the user to add a new person to the bank.

1. Flow of events
2. The user starts the application
3. The user inputs the new persons’s name and email
4. The user pushes the „Insert new” button
5. The application checks the correctness of the data
6. The application inserts the new person in the bank
7. The table of persons changes to show all the persons in the bank

Alternative scenarios:

1. The input email address is not correct: could appear at step 4
2. The application notifies the user of the incorrect input with a message, which tells the user where they wrote an invalid value.

The scenario reverts to step 2

**Scenario 2**

1. Identification Summary

Title: Add a new account for a person.

Summary: This use case allows the user to add a new account in the bank for one of the persons already in the bank.

1. Flow of events
2. The user starts the application
3. The user selects one of the entries from the „Persons” table
4. The user pushes the „Get Accounts” button
5. The user writes the data of the new account in the text fields
6. The user selects from „Spending account” or „Saving account”
7. The user pushes the „Create new button”
8. The application checks the corectness of the input data
9. The application inserts the new account in the bank, related to the selected person
10. The table of accounts changes to show all the accounts of the person in the bank.

Alternative scenarios:

1. The input data is not correct: could appear at step 7
2. The application notifies the user of the incorrect input with a message, which tells where the user wrote an invalid value.

The scenario reverts to step 4

**Scenario 3**

1. Identification Summary

Title: Add money to an existing account

Summary: This use case allows the user to add money to an existing account in the bank.

1. Flow of events
2. The user starts the application
3. The user selects one of the entries from the „Persons” table
4. The user pushes the „Get Accounts” button
5. The user selects and account from the table
6. The user writes the amount to be added in the text field
7. The user pushes the „Add money”
8. The application checks the corectness of the input data
9. The application adds the amount to the existing amount from the account in the bank, related to the selected person
10. The table of accounts changes to show all the accounts of the person in the bank.

Alternative scenarios:

1. The input data is not correct: could appear at step 7
2. The application notifies the user of the incorrect input with a message, which tells where the user wrote an invalid value.

The scenario reverts to step 5

1. The selected account is a savings account: coul appear at step 8
2. The application notifies the user of the mistake

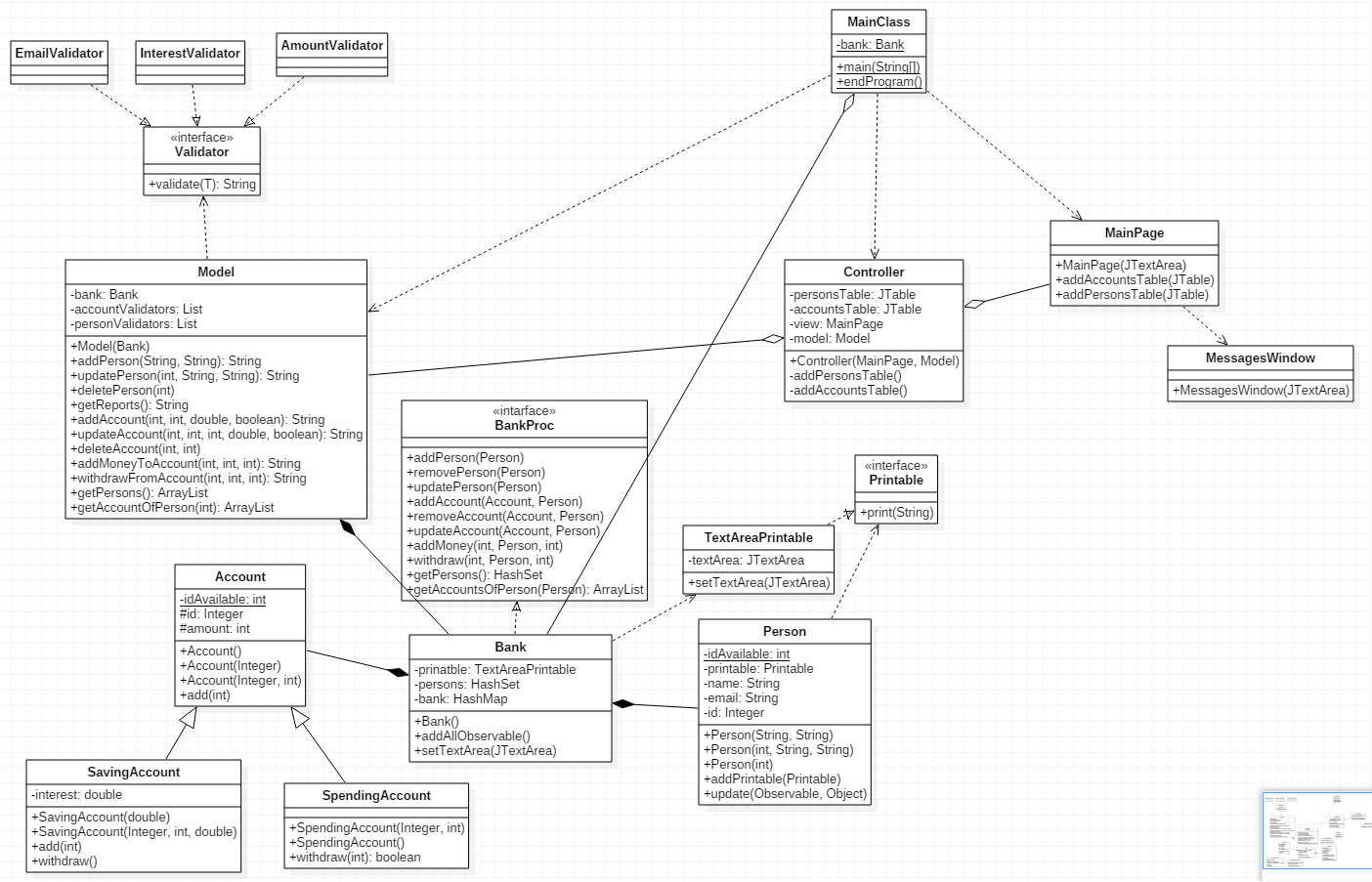
The scenario reverts to step 4

1. **Design**

2.1. Design decisions

The application uses both the Model-View-Controller architecture and the 3-tier layered architecture. Thus, from the MVC point of view, there is a separation between the problem Model (the classes concerned with solving the problem in a corect manner), the View (the classes concerned with the GUI: the parts which must be displayed on the screen, the interface between the user and the applications: text fields, text areas, buttons) and the Controller (the class(es) which make possible for the View classes to send messages to the Model classes and by this allow the user to control the the application). From the layered architecture point of view, the Controller and the View are part of the top layer, while the Model actually contains the other two layers: bussiness logic and data (bank) and also the actual model of the bank.

2.2. UML Class Diagram



2.3. Data Structures

The data structures used in the application are *lists, maps*  and *sets*:

1. ***Lists*** and namely the *Vector* and *ArrayList* implementation of the interface are used for arrays of variable (and often changing) length, because they are easier to use and have already-implemented methods for modifying the size and the required memory, thus being more efficient. They are used to model the lists of objects sent and received by the graphic user interface.
2. ***Maps*** and namely the *HashMap* implementation is used to model the relationship between persons and accounts in the bank. Each person is a key in the HashMap, having as value an ArrayList of accounts.
3. ***Sets*** and namely the *HashSet* impelementation is used to store the persons in the bank in order to avoid duplicates.

2.4. Class Design

The application consists of the following classes:

* **Person:** objects of this class are put into the bank’s set of persons and are also the keys of the hashMap containing the accounts.
* **Account:** objects of this class store the perons’ money. Accounts are stored in ArrayLists which are stored in the bank’s hashMap. Accounts are of two types (subclasses): SavingAccount (also have an interest which is added to the amount in the account after a period of time, but only allow one initial deposit and one withdrawal) and SpendingAccount (no interest, unlimited deposits and withdrawals).
* **Bank:** this class holds all the data of the application, providing methods to manipulate it. All data is stored here about the persons and the accounts: the persons are stored in a hashSet (provide uniqueness) and the accounts are stored in arrayLists which are stored in a hashMap, with the owner (person) being the key).
* **Model:** this higher-level method is used to manipulate the bank, provinding a sort of interface between the raw data (integers, Strings, etc) and the nicely formed objects (Person, Account)
* **MainPage:** this class represents the graphic user interface and it is used to get the user input and output the resulting data.
* **Controller:** Objects of this class are used to assign action listeners to the buttons of the MainPage object, implementing the action listenesr and thus assigning them functionalities. This is done through the use of inner classes.
* **QueueSim:** this class is only used to start the application through its main() method.
* **MainClass:** this class is used to deserialize the data from the file, start the application and then serialize the data back once the execution ends
* **MessagesWindow:** this class is used to create a window with a JTextArea where the Person objects print their messages (when their accounts are modified)
* **Validators:** objects of these classes implement the *Validator* interface and are used to validate some objects before they are appended in the bank
* **TextAreaPrintable:** this objects of this class are passed to other objects that need to print information on the GUI, the printing is done through this class.

2.5. Interfaces

* **Printable:** this interface is used to create a callback so that objects of various classes can print information. It is implemented by the **TextAreaPrintable** class which prints on a JTextArea.
* **Validator:** this interface is used to create objects that are appended in the bank.
* **BankProc:** this interface, implemented by the *Bank* class, is used to properly document the desing by contract requirments of the Bank objects

2.6. Relationships

There are of course a lot of relationships between the classes. The following section presents the most important ones:

* Controller (aggregation) MainPage, Model: The Controller receives its View (MainPage) and its Model from outside, in the constructor
* Model (composition) Bank, Validator: Objects of the Model class cannot exist without their Bank and Validator objects.
* Bank (composition) Person, Account: Objects of the Bank class cannot exist without their Person and Account objects.
* Bank (dependency) Printable: The Bank object uses a Printable object, assigning it to its Person objects.
* MainClass (aggregation) Bank: The MainClass class has a Bank static object, deserialized and serialized when needed.

2.7. Packages

The classes are organized in six packages: **main** pack (the MainClass and Controller classes), **model** pack (the classes comprise the bank: Person and Account), **view** pack (the GUI classes), **printables** pack (the Validator interface and its implementations, **modelLogic** (the Bank, BankProc and Model classes), **validators** (the Validator interface and its implementations).

2.8. Algorithms

At program start, the Bank object is deserialized from the file and added to a Model object, which is then added to a Controller object. The GUI is started and, when the application ends (the user closes the GUI), the GUI calls a method from the MainClass class which serializes back the Bank object into the file.

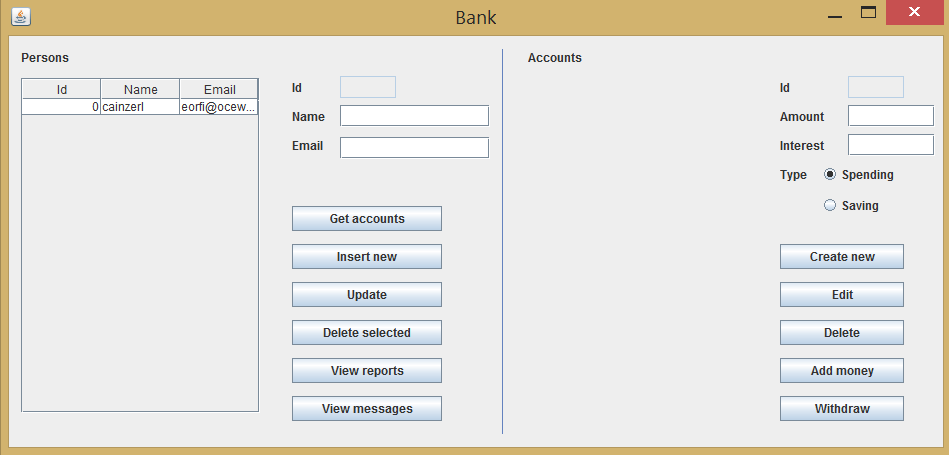
The Person and Account classes override their hashCode and equals methods so that any two objects of those classes are equal if their ids are equals. This makes it easier to access the objects in the hashSet, hashMap and arrayList without having a reference to the object: just create a new object with the same id and use it to manipulate the original object.

In order to correctly implement the difference between saving and spending accounts, their respective classes override the superclass method addMoney and impelment their own withdraw method. In the saving account, adding money to an existing account causes an exception to be throws, while withdrawing is not done by value but the whole amount is withdrawn at once. In the spending account, adding money is restrictionless while withdrawing money is done by amount, but a greater amount than is already present in the account cannot be withdrawn.

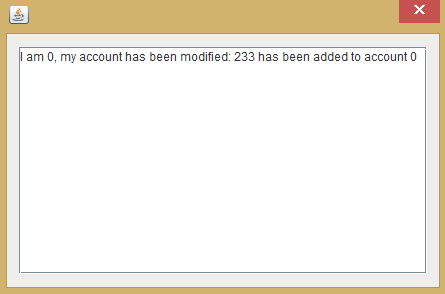
The application implements the Observer design pattern. Thus, each Person is notified when one of its Accounts was changed and their messages are displayed and can be seen by the user..

2.9. User Interface

The windows of the user interface are built using the WindowBuilder Eclipse plugin. The use of the interface is pretty self-explanatory. The main window:.



If the user wants to see the persons’ messages, another window is displayed:



1. **Implementation and testing**

4.1 Implementation (also includes some of the more interesting code segments)

* The **Person** class: has a static int idAvailable (used to automatically add unique ids to new objects), name and email Strings and a Printable object as instance variables. Methods:
  + Constructors:
    - One with just the id, used to create objects that manipulate the bank
    - One with two Strings name and email, used to create Person objects with unique ids
    - One with an int id and 2 Strings name and email, used to create Person objects with already existing ids
  + addPrintable: adds a Printable object to the calling object;
  + update: used as part of the Observer design pattern, it prints a message using the Printable object.
  + getters and setters
* The **Account** class: has a static int idAvailable (used to automatically add unique ids to new objects), an integer id and an integer amount (also a double interest if it is a SavingAccount) as instance varibles. Methods:
  + Constructors:
    - One with just the id, used to create objects that manipulate the bank
    - One parameterless, used to create an object with a new unique id, the amount is set later
    - One with two integers amount and interest, used to create objects with already existing ids.
    - (SavingAccount constructors also have the interest as parameter)
  + add: has an int amount, adds the amount received to the amount in the account. Throws a SavingAccountModificationException if this changed is attempted on a non-empty saving account. This method also notifies the object’s observers.
  + withdraw: implemented separately by the subclasses. In the SavingAccount sublcass, the withdrawal can only be done if there is money in the account and removes the whole sum. In the SpendingAccount subclass, if the amount to be withdrawn is too big (greater than the existing sum), nothing will happen and the method will return false. This method also notifies the object’s observers.
  + getters and setters.
* The **Bank** class: contains a TextAreaPrintable object, a hashSet of Persons and a hashMap of arraylists of Account objects instance variables. Methods:
  + Constructor: has no parameters, all the fields are initialized with new.
  + addAllObservable: each Account object has an Observer object added (Person)
  + addPerson: adds the Person received as parameter to the hashSet. Also creates an empty ArrayList of Account objects in the hashMap
  + removePerson: removes the Person received as parameter from the hashSet. Also removes the ArrayList of Account objects from the hashMap
  + updatePerson: updates the Person received as parameter to the hashSet

**public** **void** **updatePerson**(**Person** p) {

**assert** wellFormed() :"Not well formed";

**assert** p!=**null** :"Null argument";

**assert** persons.contains(p) :"Person not present";

**int** **size**=persons.size();

persons.remove(p);

persons.add(p);

p.addPrintable(printable);

**assert** persons.size()==size :"Size changed";

**assert** persons.contains(p) :"Person not present";

**assert** wellFormed() :"Not well formed";

}

* + addAccount: adds the Account received as parameter to the ArrayList from the hashMap of the Person received as parameter

**public** **void** **addAccount**(**Account** a, **Person** p) {

**assert** wellFormed() :"Not well formed";

**assert** p!=**null** :"Null argument";

**assert** persons.contains(p) :"Person not present";

**assert** a!=**null** :"Null argument";

**assert** !bank.get(p).contains(a) :"Account already present";

**int** **size**=bank.get(p).size();

bank.get(p).add(a);

**for**(**Person** **p1**:persons){

**if**(p.getId()==p1.getId())

a.addObserver(p1);

p1.addPrintable(printable);

}

**assert** bank.get(p).size()==size+1 :"Size not changed";

**assert** bank.get(p).contains(a) :"Account not present";

**assert** wellFormed() :"Not well formed";

}

* + removeAccount: removes the Account received as parameter from the ArrayList of the hashMap of the Person received as parameter
  + updateAccount: updates the Account received as parameter in the ArrayList of the hashMap of the Person received as parameter
  + addMoney: adds the amount received as parameter to the Account with the given id in the ArrayList from the hashMap of the Person received as parameter

**public** **void** **addMoney**(**int** accountId, **Person** p, **int** amount) **throws** **SavingAccountModificationException**{

**assert** wellFormed() :"Not well formed";

**assert** p!=**null** :"Null argument";

**assert** persons.contains(p) :"Person not present";

//other preconditions are asserted in method algorithm

**int** **money1**=0,**money2**=0;

**for**(**Account** **a**:bank.get(p)){

**if**(a.getId()==accountId){

money1=a.getAmount();

a.add(amount);

money2=a.getAmount();

**break**;

}

}

**assert** money1+amount==money2 :"Amount change inconsistency";

**assert** wellFormed() :"Not well formed";

}

* + withdraw: subtracts the amount received as parameter from the Account with the given id in the ArrayList from the hashMap of the Person received as parameter
  + getPersons: returns the hashSet of all the Person objects in the hashSet
  + getAccountsOfPerson: returns an ArrayList of all the accounts of the Person given as input parameter
  + setTextArea: sets the output JTextArea of the TextAreaPrintable object
  + wellFormed: checks whether the object is well formed i.e each Person in the hashSet has a non null ArrayList in the HashMap
* The **Model** class: contains a Bank object and two ArrayLists of Validator objects as instance variables. Methods:
  + Constructor: receives the Bank object as input parameter, used to initialize the ArrayLists of Validator objects.
  + addPerson: constructs a Person object with the input parameters and calls the addPerson method from the Bank object.
  + updatePerson: constructs a Person object with the input parameters and calls the updatePerson method from the Bank object.
  + deletePerson: constructs a Person object with the input parameters and calls the removePerson method from the Bank object.
  + getReports: iterates through all the accounts of all the persons computing the total sum in the bank and also the total sum of each person

**public** **String** **getReports**(){

**String** **msg**="";

**int** **total**=0;

**ArrayList**<Person> **persons**=**new** ArrayList<Person>(bank.getPersons());

**for**(**Person** **p**:persons){

**int** **subtotal**=0;

**ArrayList**<Account> **accounts**=bank.getAccountsOfPerson(p);

**for**(**Account** **a**:accounts){

subtotal+=a.getAmount();

}

msg+="Client "+p.getId()+" has a total of "+subtotal+" in all their accounts;\n";

total+=subtotal;

}

msg+="\nTotal in the bank: "+total;

**return** msg;

}

* + addAccount: constructs an object belonging to a subclass of Account and calls the addAccount method from the Bank object
  + updateAccount: constructs an object belonging to a subclass of Account and calls the updateAccount method from the Bank object
  + deleteAccount: constructs an Account object and calls the removeAccount method from the Bank object
  + addMoneyToAccount: calls the addMoney method from the Bank object with the input data

**public** **String** **addMoneyToAccount**(**int** personId, **int** accountId, **int** amount){

**String** **msg**="";

**if**(amount<0){

**return** "Add a positive amount;\n";

}

**Person** **p**=**new** Person(personId);

**try** {

bank.addMoney(accountId, p, amount);

} **catch** (**SavingAccountModificationException** **e**) {

msg+="Can't add money to savings account;\n";

}

**return** msg;

}

* + withdrawFromAccount: calls the withdraw method from the Bank object with the input data
  + getPersons: returns an ArrayList based on the hashSet returned by the getPersons method from the Bank object
  + getAccountsOfPerson: returns the ArrayList returned by the getAccountsOfPerson method from the Bank object
* The **MainPage** class was built with **WindowBuilder**, but it was heavily modified. The text fields, tables and buttons were transformed in instance variables so that their values can be accessed from outside. Also, additional method were added:
  + addActionListener: adds an action listener to a button
  + addTable: adds one table and its mouse click listener
  + showMessage: displays a message in a dialog box
* The **Controller** class: has a Model and a MainPage object as instance parameters. Methods:
  + Constructor: has a Model and a MainPage object as parameters, sets the corresponding fields and calls all the addActionListener methods of the MainPage object.

**public** **class** **InsertPersonActionListener** **implements** ActionListener{

**public** **void** **actionPerformed**(**ActionEvent** e) {

**String** **msg**=model.addPerson(view.getTextField().getText(), view.getTextField\_1().getText());

**if**(!msg.equals("")){

view.showMessage(msg);

}

addAccountsTable(-1);

addPersonsTable();

}

}

* + **ActionListener**(s): inner classes which implements the ActionListener interface. Methods:
    - actionPerformed: has an ActionEvent as parameter. The method retrieves data from the GUI and calls the methods from the Model object whenever the user pushes a button.
  + addPersonsTable, addAccountsTable: private methods used to create and set the tables in the GUI

**private** **void** **addPersonsTable**(){

**Vector**<String> **columns**=**new** Vector<String>();

columns.add("Id");

columns.add("Name");

columns.add("Email");

**Vector**<**Vector**<Object>> **data**=**new** Vector<**Vector**<Object>>();

**for**(**Person** **p**:model.getPersons()){

**Vector**<Object> **personData**=**new** Vector<Object>();

personData.add(p.getId());

personData.add(p.getName());

personData.add(p.getEmail());

data.add(personData);

}

***@SuppressWarnings***("rawtypes")

**final** **Class**[] **columnClass** = **new** **Class**[] {

**Integer**.**class**, **String**.**class**, **String**.**class**

};

***@SuppressWarnings***("serial")

**DefaultTableModel** **tableModel**=**new** DefaultTableModel(data, columns){

***@Override***

**public** **Class**<?> **getColumnClass**(**int** columnIndex)

{

**return** columnClass[columnIndex];

}

***@Override***

**public** **boolean** **isCellEditable**(**int** row, **int** column) {

**return** **false**;

}

};

personsTable=**new** JTable(tableModel);

view.addPersonsTable(personsTable);

}

* The **MainClass** class: used to deserialize the Bank object, start the application and serialize the Bank object back when the application is closed through its main and endProgram static methods.

4.2. Testing

Testing was done using a JUnit test case generated for the Bank class, which tests each method one by one.

1. **Results**

The resulting application is a pretty good bank management application which illustrates how we can work with serialization (in order to save data when the program is closed) and the observer design pattern.

1. **Conclusions, Lessons learned, lmprovements**

6.1. Conclusions

In building this program the two main difficulties were working with hashSets and hashMaps and building a responsive user interface. I tought that implementing the serialization/deserialization and the observers/observables would be harder but it wasn’t the case (although the two don’t really work together very well)..

6.2. Lessons learned

During the developement of this application I have learned how to work with serialization/deserialization and the observers/observables, how to create design by contract requirments and how to take advantage of overriding the equals and hashCode methods.

6.3. Improvements

* + Enhance the graphic user interface in order to show the bank evolution in a graphical manner
  + Add more statistics to be shown about the bank
  + Add more options to manipulate accounts
  + Implement an authentification method, maybe with different priviledges for different types of users.

1. **Bibliography**
   1. Serialization/deserialization:

http://www.tutorialspoint.com/java/java\_serialization.html

* 1. Overriding equals:

http://stackoverflow.com/questions/185937/overriding-the-java-equals-method-quirk

* 1. Creating new exceptions

http://stackoverflow.com/questions/8423700/how-to-create-a-custom-exception-type-in-java

* 1. Overriding the exit action from the GUI:

http://stackoverflow.com/questions/16295942/java-swing-adding-action-listener-for-exit-on-close

* 1. Observer/Observable:

http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html