Programming Techniques

Assignment 4

Student : Rujita Alexandra

Group : 30422

Contents

[1. Problem specification 3](#_Toc451541897)

[2. Problem analysis , modeling , scenarios , use cases 3](#_Toc451541898)

[2.1. Problem analysis 3](#_Toc451541899)

[2.2. Modeling 4](#_Toc451541900)

[2.3. Use cases 4](#_Toc451541901)

[3. Design 5](#_Toc451541902)

[3.1. Package diagram 5](#_Toc451541903)

[3.2. UML Class Diagram 6](#_Toc451541904)

[3.2.1. BankInterface Interface 6](#_Toc451541905)

[3.2.2. Bank Class 6](#_Toc451541906)

[3.2.3. Account Abstract Class 7](#_Toc451541907)

[3.2.4. SpendingAccount Class 7](#_Toc451541908)

[3.2.5. SavingAccount Class 7](#_Toc451541909)

[3.2.6. FileOp Class 7](#_Toc451541910)

[3.2.7. GUI Class 7](#_Toc451541911)

[3.2.8. Controller Class 7](#_Toc451541912)

[3.2.9. mainC Class 8](#_Toc451541913)

[3.3. Data structures 8](#_Toc451541914)

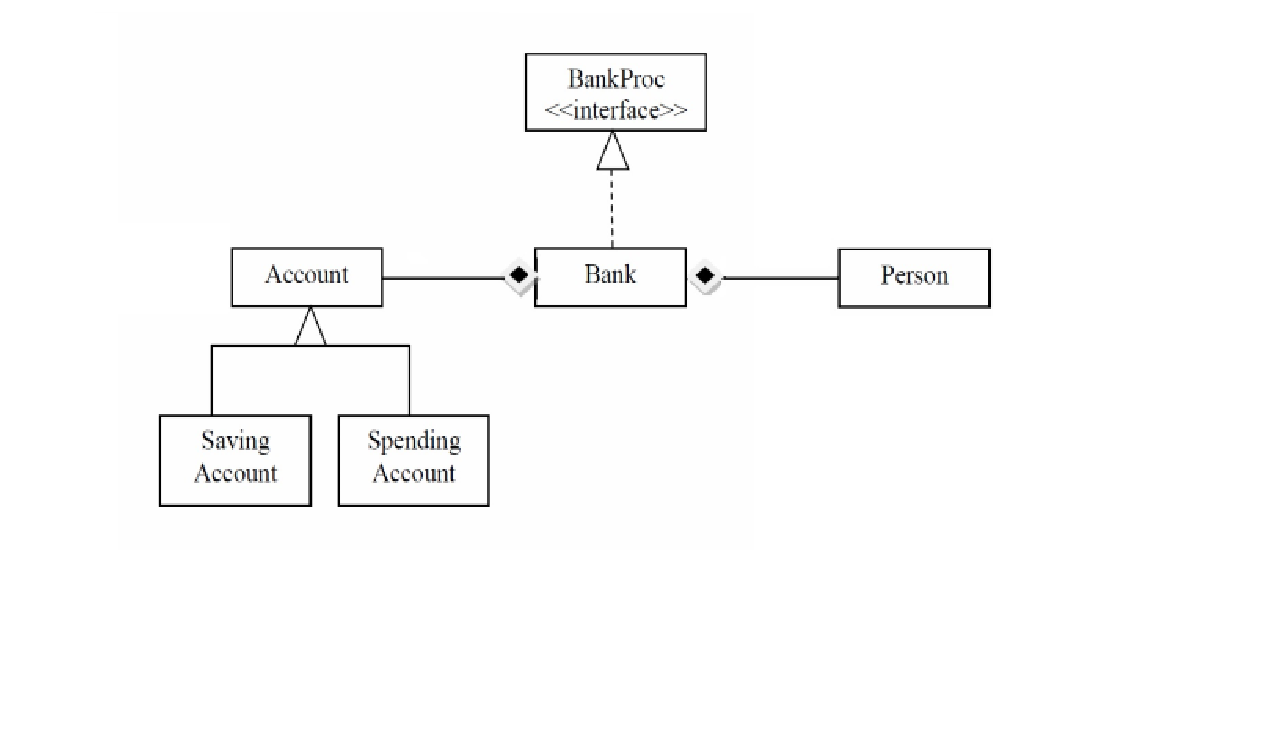
[3.4. Relationships 8](#_Toc451541915)

[3.5. User interface 8](#_Toc451541916)

[4. Implementation 9](#_Toc451541917)

[5. Further developments and importance 11](#_Toc451541918)

1. Problem specification



Define the interface BankProc that holds the next methods’ headers:

* add/remove persons,
* add/remove holder associated accounts,
* read/write accounts data
* report generators, etc

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 aboutany 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 (ro. titularul contului).

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

1. Problem analysis , modeling , scenarios , use cases

## Problem analysis

The problem can be modeled in several ways, it can take the information it needs from a serializable text file or from a data base. In my impelementation I chose the first way, because the data can be easily modified without having to change any code, and also because it is more flexible. The result of the operations can be seen in different Jtables, according to the operation.

To simulate a real life bank, only the admin can create a new account after the client has filled some forms, and is the sameway when adding money, because maybe the client wants to deposit money in somebody else’s account.

## Modeling

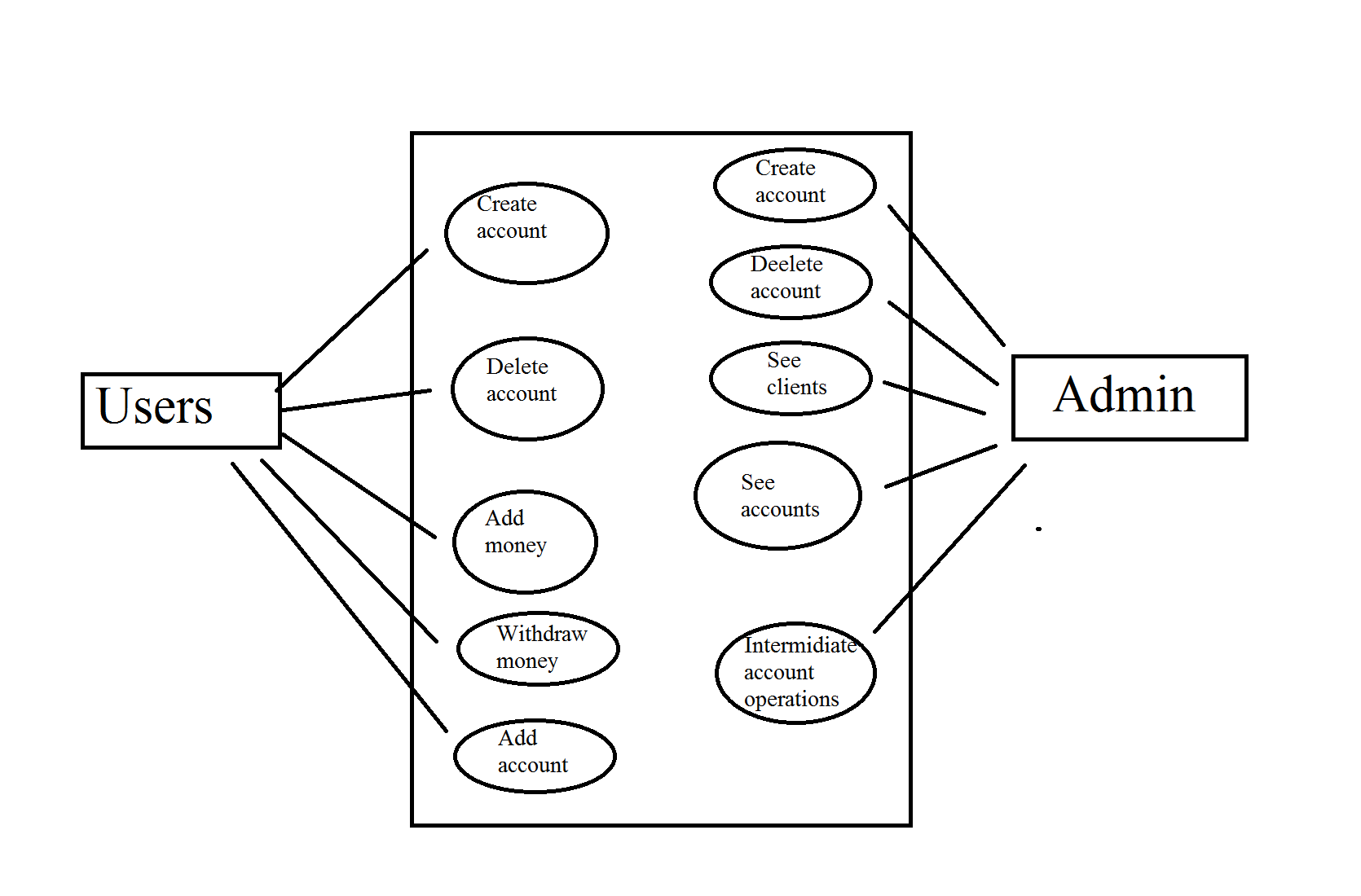
In this implementation I choose to read data from a serializable text file every time I run the program, to retrieve the information and populate the bank’s records. Also, everytime I add a new account or any other operations are done on the accounts or persons I re-write the information in the file.

Only the admin can modify the bank’s records structure after the client has told him everything he needs to know.

The bank records are modeled like a hashMap which has as key the person and as value a set of accounts, because a person can have more than one account. Accounts are of 2 different types: saving and spending, they differentiate by the intereset or the maximum sum of money that is allowed to be reetrieved every month.

Also to implement the bank structure I used an interface, that holds the headers oh the methods that have as parameters generic types.

## Use cases



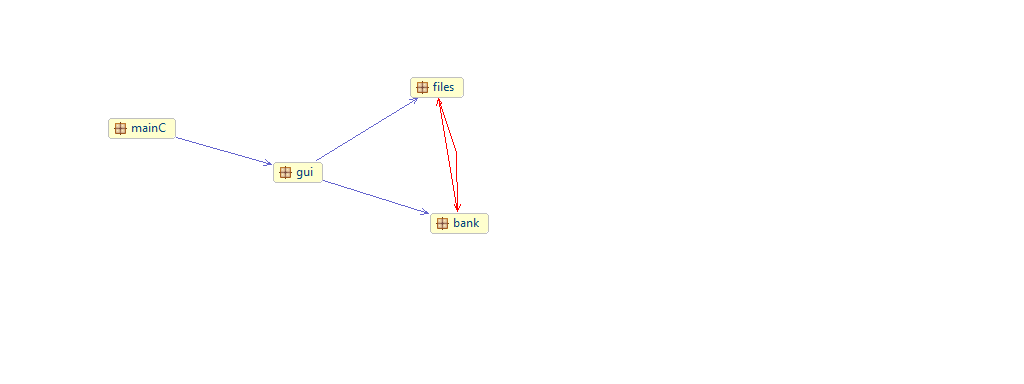
Clients can perfom different operations on their account, but they have to do it through the admin. He is responsible to intermidiate the transformations requierd by the client on the account.

When choosing to open their first account at the bank the clients have to choose what type of account they want and have to deposit some money in it.

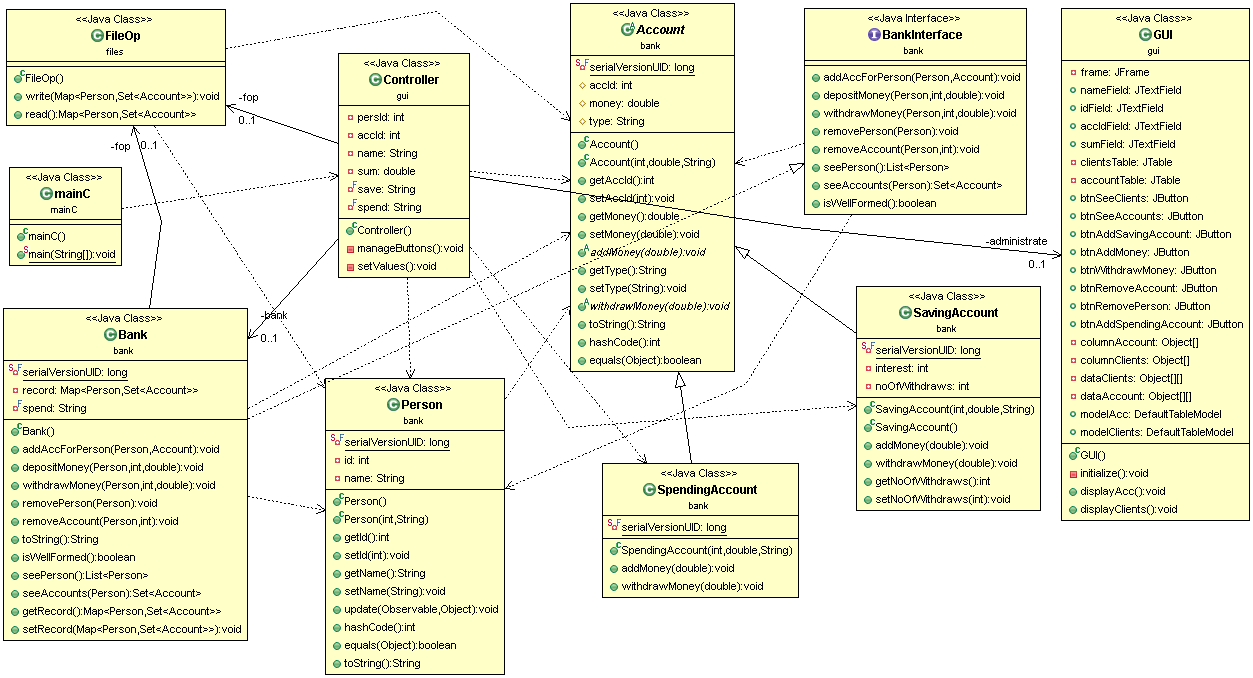
1. Design

## Package diagram

Package Dependencies Diagram shows the dependencies between packages and highlights circular dependencies. Each dependency line also displays the involved classes.



## UML Class Diagram

The UML diagram is a class diagram that shows a set of classes, interfaces and collaborations and their relationships. Class diagrams may also contain packages or sub-systems, both of which are used to group elements of the model.

### BankInterface Interface

This is the bank’s class interface, it holds all the methods that will be later implemented in the Bank class. Also in this class all the pre and post conditions are speciefied, that will make the bank be well-formed and be certain no unexpected error will appear during the execution of the program.

### Bank Class

This is the main class for account and person processing. It has as base data structure a HashMap with the key Person and the value a Set of Account. It implements all the methods from the interface ( BankInterface ) and also does the necessary verifications according to the pre and post conditions specified for each method in the bankInterface. It does so using the assert rules. It also has a method for checking if the bank is consistent ( has at least one person with one account).

When adding an account for a person it first checks if the person exists in the bank’s records and if not, creates a new fields in the bank’s records.

### Account Abstract Class

This is an abstract class, because some methods are implemented ( but for the generic type ) but there are some abstract methods as well, that are not implemented and have to be implemented in the classes that will extends this class.

### SpendingAccount Class

This class extends the Account class and has to implement their own add and withdraw money methods in a way that it will differentiate from the SavingAccount class. For example as long as there are money in the account the client can withdraw them with no restriction and can add any some of money, no matter how small or big it is.

### SavingAccount Class

This class also extends the Account class and has to implement their own add and withdraw money methods in a way that it will differentiate from the SpendingAccount class. For example the client can withdraw money only 3 times per month and has do deposit at least 1000 units at once.

### FileOp Class

This class purpose is to work with text files. It does the reading from file, reading all at once in a HashMap<Person, Set<Account>> object. That is the same type as the bank’s records.

It also writes in the file, every time an operation on the persons or accounts is done.

### GUI Class

This class creates the user interface for the admin, which has the table of clients and accounts corresponding to a particular client. It also creates all the fields necessary for the available operations.

### Controller Class

This is the main class for user interface operations. It holds all the actionListener for all the buttons or text fields. From this class “signals” are so operations are done on the text file. Also here are done certain verifications such as: some filed is empty or the introduced value does not have the appropriate format. This class updates the content of the tables that are displayed when pressing the See clients or See accounts.

### mainC Class

It is the class that starts the application. It creates a new generator object and a main thread that holds the generator object.

## Data structures

There are 3 main data structures in this project. The Person object, the Account and the Bank. The bank holds records of the person an how many accounts has.

The Account object has as fields: type, sum and id.

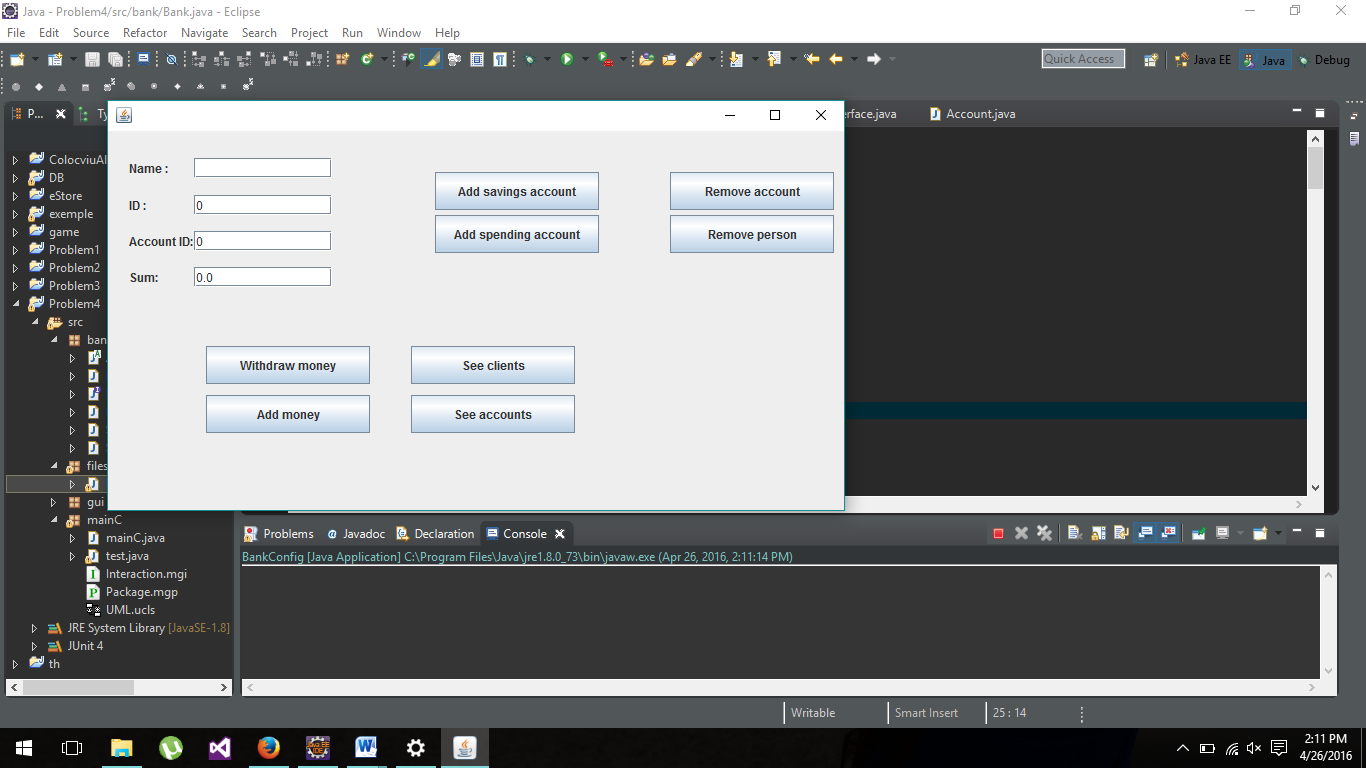
## Relationships

There is a strong relationship between the Generator class and the CashRegister Class, but also between it and the GUI class and FileOp class ( used to communicate with the user ).

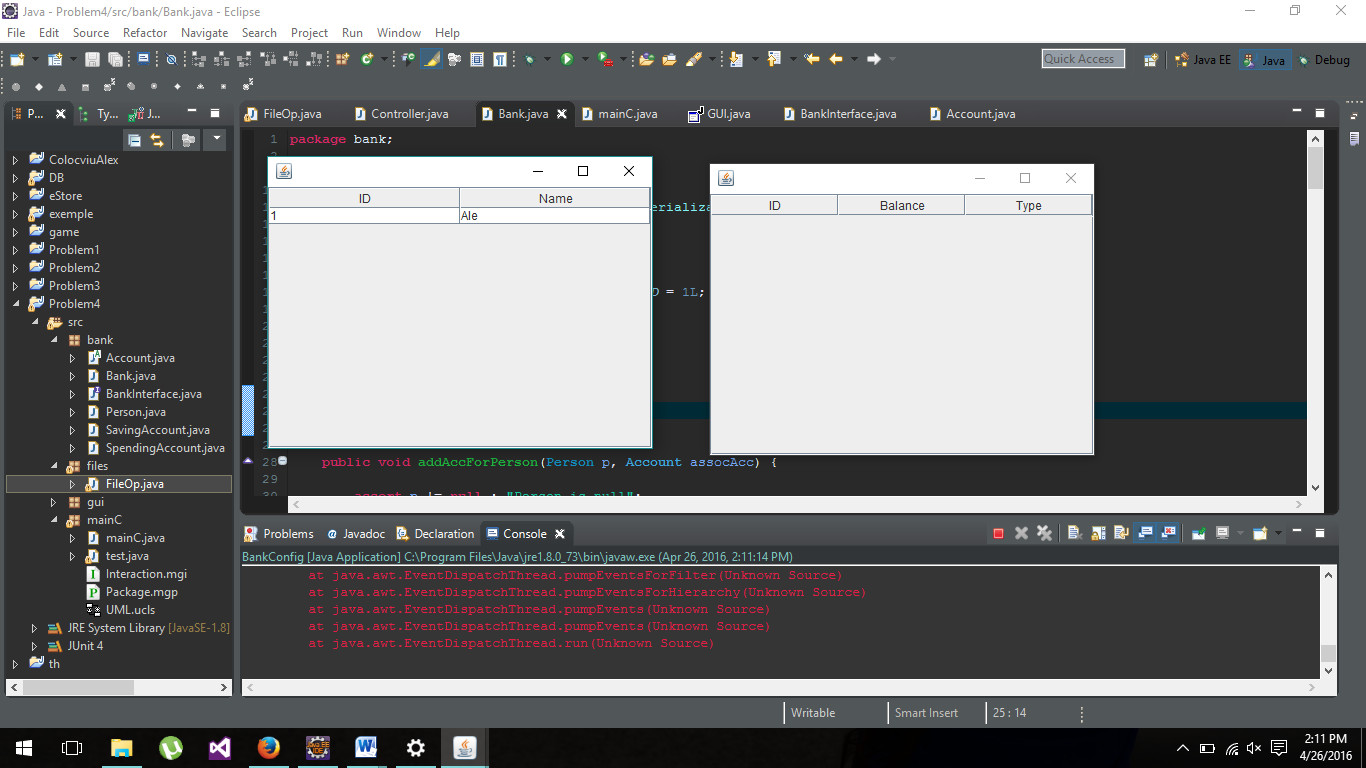
A strong relationship is also between the CashRegister class and the Server class, and further between the Server class and the Task class.

Following the object oriented programming rules there is a weak coupling and strong cohesion between classes.

## User interface



As you can see different operations can be done on the persons list or on the accounts of a persons. When choosing to open a new account (you are the first time here ) you have to choose between saving an spending account;



Here you can see the persons in the bank after pressing the „See persons” buttons.

1. Implementation

In the Account class, which extends the Observable class we have the add or withdraw abstract method that is later implemented in the SpendingAccount and SavingAccount class:

|  |
| --- |
| @Override |
|  |

|  |
| --- |
|  |
| public void withdrawMoney(double money) { |
|  |
|  |
|  |
|  |
|  |
| if (getNoOfWithdraws() > 3) |
|  |
|  |
| notifyObservers("You reached the maximum nr of withdraws for this 6 months !"); |
|  |
|  |
| else { |
|  |
|  |
| this.money -= money; |
|  |
|  |
| setChanged(); |
|  |
|  |
| notifyObservers("Sum " + money + " withdraw form the saving account " + this.getAccId()); |
|  |
|  |
| } |
|  |
|  |
|  |
|  |
|  |
| } |
|  |

|  |
| --- |
|  |

And in the Person class, which implements the Observer interface we have:

|  |
| --- |
| @Override |
|  |

|  |
| --- |
|  |
| public void update(Observable account, Object sum) { |
|  |
|  |
|  |
|  |
|  |
| if( account instanceof Account){ |
|  |
|  |
| Account acc= (Account) account; |
|  |
|  |
| System.out.println(acc.toString()+" "+sum); |
|  |
|  |
| } |
|  |

|  |
| --- |
| } |
| All the data is stored in a serializable file, which is re-written every time we make some modifications   |  | | --- | | public void write(Map<Person, Set<Account>> bankRecord) { | |  |  |  | | --- | |  | | try { | |  | |  | | FileOutputStream fileOut = new FileOutputStream("Bank.ser", true); | |  | |  | | ObjectOutputStream out = new ObjectOutputStream(fileOut); | |  | |  | | out.writeObject(bankRecord); | |  | |  | | out.close(); | |  | |  | | fileOut.close(); | |  | |  | | } catch (IOException e) { | |  | |  | | e.printStackTrace(); | |  | |  | | } | |  | |  | | } | |  | |  | |  | |  | |  | | public Map<Person, Set<Account>> read() { | |  | |  | | FileInputStream fileIn; | |  | |  | | Map<Person, Set<Account>> b = new HashMap<Person, Set<Account>>(); | |  | |  | | try { | |  | |  | | fileIn = new FileInputStream("Bank.ser"); | |  | |  | | ObjectInputStream in = new ObjectInputStream(fileIn); | |  | |  | | b = (HashMap<Person, Set<Account>>) in.readObject(); | |  | |  | | in.close(); | |  | |  | | fileIn.close(); | |  | |  | | } catch (IOException | ClassNotFoundException e) { | |  | |  | | e.printStackTrace(); | |  | |  | | } | |  | |  | | return b; | |  | |  | |  | |  | |  | | } | |  |  |  | | --- | |  | |

1. Further developments and importance

As for future devolpments we can add a time stamp so we know when the user has withdraw money from the saving account. Also instead of printing in the console the result of the notify() and assert instructions we cand add a JoptionPane in which the messages should be displayed.