Technical University of Cluj-Napoca  
Faculty of Automation and Computers  
Department of Computer  
2nd Semester 2015-2016

Assignment 4.

Bank

Student: Nagy Hanna

Group: 30421

1. Objective of the assignment

1.1. 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.

1.2. Define and implement the classes Person, Account, SavingAccount and SpendingAccount. Other classes may be added as needed (give reasons for the new added classes).

1.3. An Observer DP will be defined and implemented. It will notify the account main holder about any account related operation.

1.4. 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.

1. 4.1 Define a method of type “well formed” for the class Bank.

1. 4.2 Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions).

1.5. Implement a test driver for the system.

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

Other details about teh implementation.

I have chosen to implement an application with a graphicat user interface where the user can create new clientss, attach to them the two different types of accounts, simulate withdrawal, depositing, deleting clients from the bank-clients-database and removing accounts from the clients existed in the bank database. After simulating and verifying the results, the user can command the generation of reports obtained in a separate file which is impelemented by serialization (which means that the objects- clients with their accounts- are stored in a separate file).

## Problem Analysis

My first objective was to analyze the characteristics of the saving and spending account.

There is an abstarct class named Account which impelements the general methods and has the general instance variables which is owned either by the Saving Account and the Spending Account too, they are extended this Account class and they implement in their own way the methods which were notb implemented in the abstact Account class:

**public** **abstract** **void** addMoney(**double** sum);

**public** **abstract** **int** withdrawMoney(**double** sum);

These child classes of the Account class have not just implementation differences but meaning differencies also.

The owner of the saving account cannot withdraw more money than the a constant value, in our case this constant value is 500 euro. And also there is a restriction of the number of times the client wants to withdraw from its account, in our case the is no more chances of withdrawal after 3 attempt in a month. The client who has a spending account have no restrictions only the sum he or she wants to withdraw to be less or equal with the amount of money presented in the account. The owner of a spending account has no interest rate but instead of this advantage he can spend more money then in case of a saving account. Of course, we implemented the accounts in a way that there could not be negative balance on the account, or you cannot withdraw or insert negative amount of money, but these restrictions build the whole design by contract design pattern concept. We have pre condition, post conditions and invariant methods too. For example:

/\* \*

\* Adds an account to a person

\*

\* **@param** p

\* **@param** acc

\* **@pre** acc != null

\* **@pre** p!= null

\* **@post** numberOfAccounts@pre +1 = numberOfAccounts

\* **@post** person's accounts contains (account) true

\* **@invariant** isWellFormed

\*/

**public** **void** addAccforPers(Person p, Account acc) {

**assert** isWellFormed(client);

**assert** p != **null** && acc != **null**;

**int** nrOfPreAccounts = nrOfAccounts(p);

System.***out***.println("nr of PREEaccounts" + p.getName() + nrOfPreAccounts);

acc.addObserver(p);

**if** (client.containsKey(p)) {

Set<Account> accounts = client.get(p);

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

**if** (account.equals(acc)) {

**return**;

}

}

accounts.add(acc);

serialization();

deserialization();

**assert** isWellFormed(client);

System.***out***.println("nr of Postaccounts" + p.getName() + nrOfAccounts(p));

**assert** (nrOfPreAccounts + 1) == nrOfAccounts(p);

**assert** containsAccount(p, acc);

**return**;

}

// account/s does not exists

Set<Account> newAcc = **new** HashSet<Account>();

newAcc.add(acc);

client.put(p, newAcc);

serialization();

deserialization();

**assert** isWellFormed(client);

System.***out***.println("nr of Postaccounts" + p.getName() + nrOfAccounts(p));

**assert** (nrOfPreAccounts + 1) == nrOfAccounts(p);

**assert** containsAccount(p, acc);

}

In this simple method we can see that these pre conditions, post conditions, and invariant method : @Override

**public** **boolean** isWellFormed(Map<Person, Set<Account>> client)

are implemented with the assert functions.

Of course, there exists an interface named BankInterface which contains these methods unplemented and above each method there is a comment section where are defined the conditions.

This way, since these two types of accounts are available, it doesn't make sense to instantiate the class Account. For this reason, the Account becomes abstract class and obeys Liskov's Subtitution Principle and Dependency Inversion Principle from the SOLID principles. In such manner it is easy to extend the program to support a new type of account, if needed. Dynamic polymorphism is also used for some of the methods (for example withdraw, implementations of which differ substantially).

In analyzing the relationship between class Person and Account one can deduce that this is a bidirectional relationship, namely association, since a person can have many accounts and every account has an owner.

The problem specification defines that the bank stores the accounts in a hash table or hash map. Every Object that can go into this hashmap must contain a relevant hashCode method. In our case, the accounts and client pairs need to be stored in the table/map, so a relevant hash code for this class can be defined as follows: the hash code equals the product of the ascii codes of the characters contained by the string obtained by concatenating the id and the name of the owner modulo 6666609 (large prime).

Since the scope of this assignment was to practice the design by contract technique, some of the conditions of the bank are in fact conditions for the hash map, since the bank uses delegates to it. That is, for example, the preconditions, postconditions invariants for the add account method are in fact realized by the hashmap, because this methods imply a simple delegation. That is why no preconditions, postconditions and invariants will be implemented for the add and remove methods for the bank class.

## Development

In the process of developing the application.

From preliminary analysis, we need the following classes:

* 1. **Bank** class - for modeling the bank
  2. **BankLog** class - for modeling the log storing events of the bank procedures
  3. **Account** class - for modeling the abstract account
  4. **SavingAccount** class - for modeling the saving account
  5. **SpendingAccount** class - for modeling the spending account
  6. **Person** class - for modeling the person, the owner of account
  7. **Main** class - for modeling the main class
  8. **TestBank class**
  9. **Serialization class**
  10. **Classes related to the graphical user interface**

Precondition, postcondition and invariants

*Well formed state*

A well formed state could be defined as follows: there cannot be client which is null in the bank-database, there cannot be client without name, client without at leat one account, client with negative balance.

*The addition*

A precondition for adding an account to the bank is the account not to be null. A postcondition is that the size increases by 1. Another postcondition is that all the old accounts are also present in the new state. At the end, we must arrive to a well fromed state. This can be formulated using Javadoc as follows:

/\*\*

\* Adds an account to a person

\*

\* **@param** p

\* **@param** acc

\* **@pre** acc != null

\* **@pre** p!= null

\* **@post** numberOfAccounts@pre +1 = numberOfAccounts

\* **@post** person's accounts contains (account) true

\* **@invariant** isWellFormed

\* \*/

**public** **void** addAccforPers(Person p, Account acc);

*The removal*

/\*\*

\* Delete an account from a person

\*

\* **@param** p

\* **@param** acc

\* **@pre** acc != null

\* **@pre** p!= null

\* **@pre** p is a client

\* **@pre** p contains(account)

\* **@post** numberOfAccounts@pre -1 = numberOfAccounts

\* **@post** person's accounts does not contains (account) false

\* **@invariant** isWellFormed

\* \*/

A precondition for the removal is for the account to be different from null and the size to be greater than zero. As a postcondition, the new size must be by 1 smaller than the old one, else everything is left unchanged. As a result, we must arrive to a well formed state.

*The withdraw, deposit and readAccount operations*

deleteAccount(Person person, Account account);

/\*\*

\* Delet a person/client

\*

\* **@param** p

\* **@pre** p!= null

\* **@pre** p is a client

\* **@post** numberOfClients@pre -1 = numberOfAccounts

\* **@post** p is not a client

\* **@invariant** isWellFormed

\* \*/

**public** **void** deletePerson(Person person);

/\*\*

\* Deposit money to an account of a client

\*

\* **@param** p

\* **@param** accId

\* **@param** sum

\* **@pre** p!= null

\* **@pre** sum > 0.0

\* **@pre** p is a client

\* **@pre** p has account with accId

\* **@post** sum > sum@pre

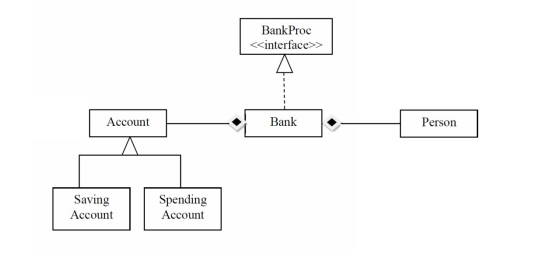
\* **@invariant** isWellFormed

\* \*/

**public** **void** depositMoney(**int** accId, **double** sum, Person p);

The following the class diagram associated with the application.

It could be easily seen the relationships between the elements/ classes in the application.



## Conclusion and further development

For developing this application, the biggest challenge was to familiarize myself to Design by Contract Programming technique. It was hard at first to identify correctly the preconditions, postconditions and invariants for classes, and to implement them in code.

I find these technique very useful for developing applications with a high percent of code tested, although these assertions should be disabled before delivering production code.

Despite these facts, I would not rely only on this technique, this is why I also practiced Test Driven Development for creating the chained hash table.

## Bibliography

* + Java SE 6 documentation: http://docs.oracle.com/javase/6/docs/
  + Design Principles and Design Patterns, Robert C. Martin "Uncle Bob"
  + Thinking in Java, Fourth Edition, Bruce Eckel, MindView, Inc.