Student: Ardelean Eugen-Richard

Group: 30423

Programming Techniques

Homework 4

Contents:

1. Problem Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
2. Analyzing the problem, modelling, scenarios, use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3. Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
4. Implementation and testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
5. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
6. Conclusion and future developments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
7. Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
8. **Problem Specification**

*Objective*

Design by Contract Programming Techniques, Design Patterns

*Description*

Consider the system of classes in the class diagram below.

1. Define the interface BankProc (add or remove persons, add or remove holder associated accounts, read or 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 or saved from or to a file.

1. **Analyzing the problem, modelling, scenarios, use cases** 
   1. **Analyzing the problem**

We have to create a bank that allows persons to have multiple accounts of either the spending or saving type. This is done, by using the ideas from homework 3 but without actually creating a connection between the project and the database. We practically do the same thing and create the classes that represent tables with the columns from the tables as fields in the classes (person and account). Then by using the inheritance relationship between saving or spending account class and account class, we create a differentiation between the two.

The linking between the person and his accounts in the bank class is done using hash tables with the choice of data structure left to us.

There will also have to be created a class that uses a bank object to read from and write into the bank.

The GUI part of the project will be explained in the User Interface part of the documentation.

* 1. **Modelling**

Modelling is the activity to make an abstract concept easier to understand by finding its main characteristics and defining some laws which make the given phenomenon quantifiable. This process also includes the decomposition of a complex problem into smaller and simpler problems which will be easier to implement.

Fortunately, this part of the problem was given by class design required, because the problem of implementing a bank is already decomposed in implementing persons and accounts and combining them into the bank class.

* 1. **Scenarios**

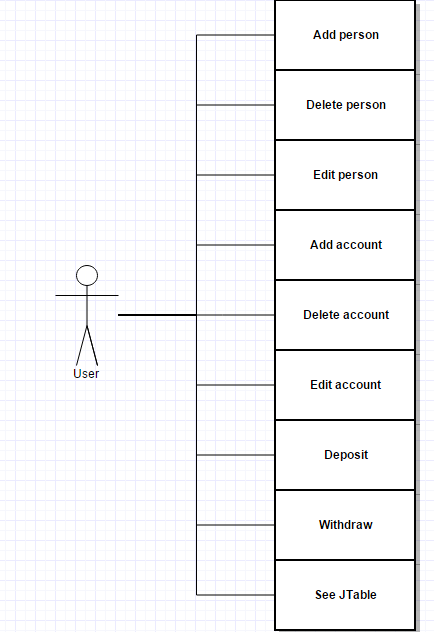
A success scenario looks like this:

* Chooses an operation, for example person operations
* From the 3 options of add, delete and edit, let’s assume add is chosen
* The required fields are introduced correctly
* The table is updated and the new person can be seen

While a failure scenario looks like this:

* Chooses an operation, for example person operations
* From the 3 options of add, delete and edit, let’s assume add is chosen
* The required fields are not introduced correctly
* An error appears, for example if the amount of money introduced in the account is smaller than zero, it will generate an error that tells you that money has to be a positive number.
  1. **Use Cases**

Use case diagrams are usually referred to as behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors).



1. **Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface)**
   1. **Design decisions**

There have been many decisions in the implementation of this project. Just for simplicity, I have chosen to have 2 fields in both the person (id and name) and account class (id and money).

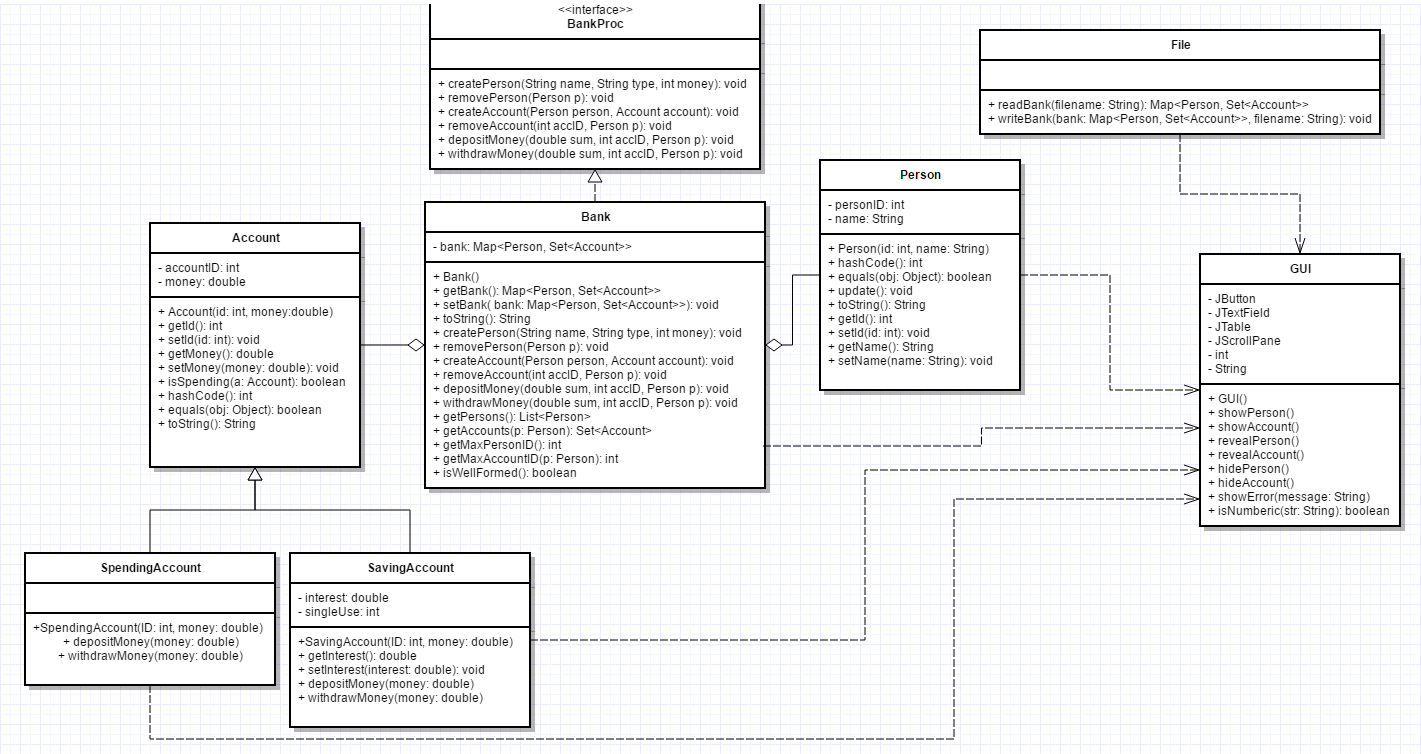
While in the previous project we used a database that had been connected to our application, in this project we do not have this, so the id cannot be autoincremented. I have implemented this by creating a method that looks in the bank for the highest id, and when inserting the next person or account, the given id is this maximum plus one.

Also for creating the JTables (person and account), it was necessary implement methods that return all persons from the bank and all accounts of a certain person (person sent as parameter). This is used when creating a JTable as the fields of one of the persons or accounts is a row of the table.

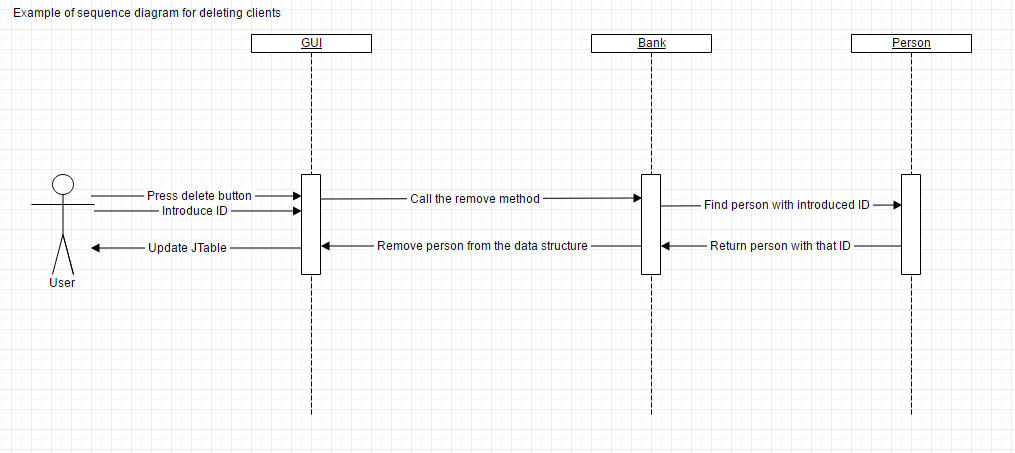
* 1. **UML diagrams**

We prepare UML diagrams to understand a system in better and simple way. A single diagram is not enough to cover all aspects of the system. So, UML defines various kinds of diagrams to cover most of the aspects of a system.

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations and collaboration. Class diagrams basically represent the object-oriented view of a system which is static in nature.



A sequence diagram is a diagram that shows the interaction between the different objects of the application. For this application it would look something like this:



* 1. **Data Structures**

We are asked to use hash tables in this project but the data structure with which we implement them is left to our choice.

I chose to implement the bank by using the Map data structure, a data structure that maps keys to values (a map cannot contain duplicate keys; each key can map to at most one value).

**private** Map<Person, Set<Account>> bank;

And for the hashing part of the project, we use the constructor in the following way:

**public** Bank()

{

bank = **new** HashMap<Person, Set<Account>>();

}

* 1. **Class Design**

For this project the implementation of the following 5 classes has been mandatory: bank, person, account, saving account and spending account with the possibility of adding extra classes where we consider necessary. I have chosen to implement 2 more classes beside the required five, a class called File, which implements the generation of a file required by the project and another class called GUI which is the graphical user interface of the project.

* 1. **Interfaces**

For the implementation of this project, as required there was one interface used in the creation of the Bank class, called BankProc. Here you can find the methods of the class Bank. When it comes to interfaces, this project is special, because it is the first project in which the implementation of an interface is mandatory.

* 1. **Relationships**

Between the required classes, there are also assigned relationships that have to be respected when implementing the classes. In the implementation of the project, we can find the following relationships:

* inheritance, between the account class and its children saving account and spending account
* aggregation, between the class bank and the classes it is formed of person and account, in this case we have aggregation because if we were to destroy bank, each could still exist
* implementation, between the interface BankProc and the class Bank
* dependency, between the GUI class and almost all of the other classes, because the user interface is dependent upon them
  1. **Packages**

For the current implementation, all classes have been saved in one package, just because of simplicity. The grouping of classes can be done easily in the case of this project, one option would be: the graphical user interface package that contains the user interface classes, the file generating package which contains the classes used to load and store the bank and the bank itself which consists of the 5 classes discussed in the above sections.

* 1. **Algorithms**

One of the best examples of how an algorithm looks like in this project is the deposit money method. Here you can see the start where it tries to prevent errors from happening, then the actual algorithm when you know that the introduced values actually exist, then the modification required and the final test.

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

{

// verify if the bank is well formed

**boolean** wellFormed = isWellFormed();

**assert** wellFormed : "Bank not well formed";

// verify if the person exists in the bank and if the sum is positive

**assert** p!= **null** : "The person must not be NULL";

**assert** bank.containsKey(p) : "The person must be from the database";

**assert** sum >= 0 : "The sum must be positive.";

d**ouble** initialMoney = 0, finalMoney = 0;

**boolean** isSavingAccount = **false**;

**double** interest = 0;

// if the person exists in the bank

**if** (bank.containsKey(p))

{

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

// the for goes through all the accounts of the person

**for** (Account a: accounts)

{

// find the account you want to deposit into

**if** (a.getId() == accID)

{

initialMoney = a.getMoney(); // get the money before deposit

a.depositMoney(sum); // sets the new sum with interest

finalMoney = a.getMoney(); // get the money after deposit

**if** (a **instanceof** SavingAccount)

{

isSavingAccount = **true**;

interest = ((SavingAccount) a).getInterest();

}

}

}

}

//if it is a saving account we compare it with an interest as well, otherwise just with the depositated money

**if** (isSavingAccount)

**assert** initialMoney == ( finalMoney - sum \* (1 + interest) ): "The money was not deposited";

**else**

**assert** initialMoney == ( finalMoney - sum ) : "The money was not deposited";

//just to be sure we verify again if the bank is well formed

wellFormed = isWellFormed();

**assert** wellFormed : "Bank not well formed";

}

* 1. **User Interface**

The User Interface was built using Java Swing elements (buttons, panels, frames, labels, textfields). The interface can be called user-friendly because it needs only the most basic of actions and knowledge to make it work, the inputs will be read from textfields that have labels with their individual meaning positioned so that they cannot be mistaken.

The user interface works in the following manner, when opened it will show 4 buttons and the person and account tables. The 4 buttons are the 4 main operations that can be done: person operations, account operations, withdraw and deposit. Let’s assume we choose the person operations button, this will reveal 3 more buttons: add, delete and edit. When clicking one of the 3 buttons, the other 2 will disappear and some text fields will appear, where you have to introduce values, after introducing the desired values, you have to click the button again which will either update the tables with the new person or it will generate an error because of the introduced values and will tell the user how to correct them.

1. **Implementation and testing** 
   1. **Implementation**

The application has been implemented in Java Programming language, using Eclipse. For the GUI, all the buttons, textfields and other components were added by code without any “drag and drop” plugins. Listeners are placed for specific component to catch events (usually a simple button press), and respond accordingly.

All the implemented classes, including their methods and attributes were documented with comments to be easier to understand.

One of the best examples on which you can see the implementation of the project is the method for creating a new account:

**public** **void** createAccount(Person person, Account account)

{

**boolean** wellFormed = isWellFormed(); // isWellFormed is a method that returns false if the bank is empty

**assert** wellFormed : "Bank not well formed"; // asserts that the bank is well formed

**assert** person != **null**: "The person must not be NULL"; // asserts that the person is not a null value

**int** initialSize;

// if the bank contains the person, it adds the account to the list of accounts of that person

**if** (bank.containsKey(person))

{

initialSize = bank.get(person).size(); // gets the number of accounts of that person before adding a new one

bank.get(person).add(account);

}

// if the bank does not contain that person, it puts the person in the bank

**else**

{

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

initialSize = 0; //if the person does not exist he cannot have any accounts

accounts.add(account);

bank.put(person, accounts);

}

// the account observers the person to see if any changes are made

account.addObserver(person);

// finalSize is the number of accounts of that person after adding a new one

**int** finalSize = bank.get(person).size();

**assert** initialSize == finalSize-1: "Account could not be added"; // if initialSize != finalSize-1 then the account could not be added

wellFormed = isWellFormed();

**assert** wellFormed : "Bank not well formed"; // just to be sure we did not ruin the bank, we verify again if it is well formed

}

* 1. **Testing**

The first way of testing, the more rudimentary type was done using try-catch structures and by printing exceptions and by printing errors if any occur during the input or execution part of the application.

Errors will appear for leaving certain textfields as their initial values, there also some constraints on the fields, for example ID has to be integer as does money. Also there are some particular fields with exceptions of their own, for example money has to be always positive as in the real world you would not be able to deposit -500 units.

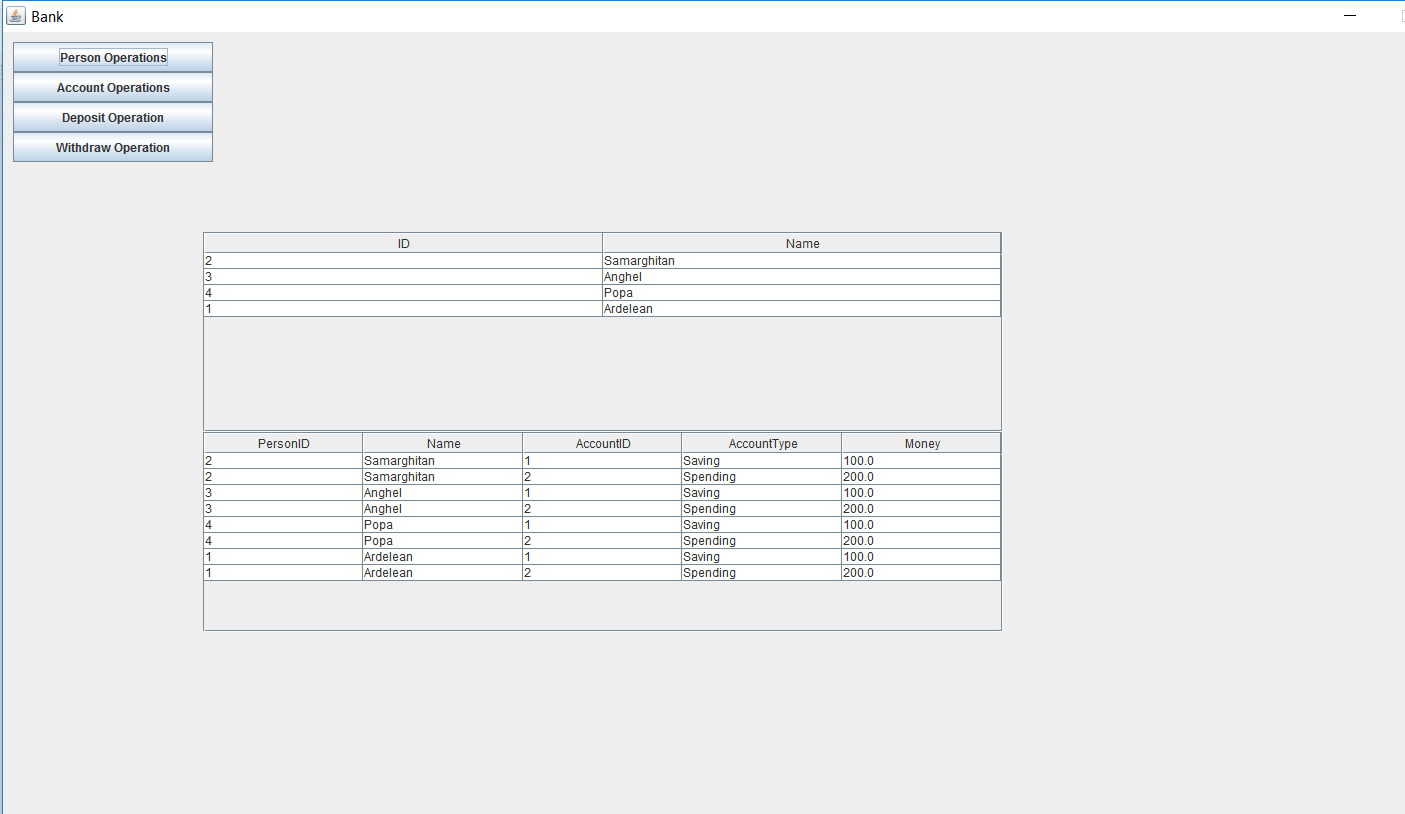
For extra points it is asked to create a Junit test, which is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks.

I have introduced 5 tests, that test if the add and delete of persons and accounts work. Fortunately, all of them are working.

1. **Results**

Through hard work and intensive testing, I have been able to create an application for simulating a bank. In the end the application has the following aspects: operations on persons (add, delete and edit), operations on accounts (add, delete and edit), withdraw and deposit and the persons and their accounts can be seen in 2 JTables. The JTables have click listeners that if you click one of the cells it will show you a message with what is in the cell you clicked. I have also been able to Design by contract, this can be seen in the interface where the user can view the preconditions and postconditions and in the bank class where the methods have assertions to ensure that they work correctly. There exists a file from which it can be both read from and written into, this is the save and load of information.

The final result looks something like this:



1. **Conclusion and future developments**
   1. **Conclusion**

This homework will get the student familiar with new concepts that will be useful to him later in his career. Using the observer interface and the observable class, we learn how we are able to be informed of changes in observable objects. As required the saving account allows only one deposit and withdrawal, because of this at the moment of the withdrawal, whatever the sum asked, all of the money from that saving account is withdrawn, because otherwise the account will have money left without being able to withdraw it. Even though the saving account is not very complex, I still believe that this homework has helped me learn quite a few things.

* 1. **Future developments**

As for most projects, there is a big number of improvements that could be brought to the project:

* First one and easiest one would be implementing more fields in the person and account classes
* There could be implemented a timer and allow the saving account to be used only after a certain amount of time, to allow it to be more realistic
* There could be an operation of merging accounts of the same type
* Such a bank could register the date in which transactions were made as well
* The user interface could be improved because it is rather rudimentary

1. **Bibliography**

* To draw the diagrams:

<https://www.gliffy.com/>

* Answers to questions regarding JAVA syntax:

<https://stackoverflow.com>

* Explanations for diagrams:

<https://www.tutorialspoint.com/uml/uml_standard_diagrams.htm>

* Used when choosing the data structure with which to implement the bank

<https://docs.oracle.com/javase/7/docs/api/>