Technical University of Cluj-Napoca
Faculty of Automation and Computer Science

Programming Techniques

Homework Assignment 4

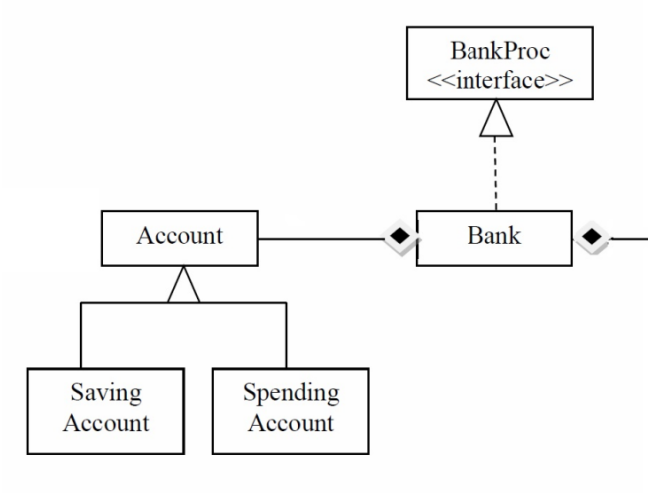
Bank simulation

Student: Crisan Adrian Gabriel
Group: 30433

Table of contents

1. Problem specification

Considering the system of classes in the class diagram specified,



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.
Design 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 about any 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).
Design and 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, modelling, scenarios, use cases

a. Problem analysis, modelling

For this application I used the paradigms of Object Oriented Programming in Java. Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

In computing, a hash table (hash map) is a data structure which implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions must be accommodated in some way. Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Hash tables process workloads to create and update themselves, querying the data they contain and running applications against it.

Hash tables are often needed where big amounts of data are involved, and where data needs to be stored on the long-term, in secure and even encrypted sites and environments.

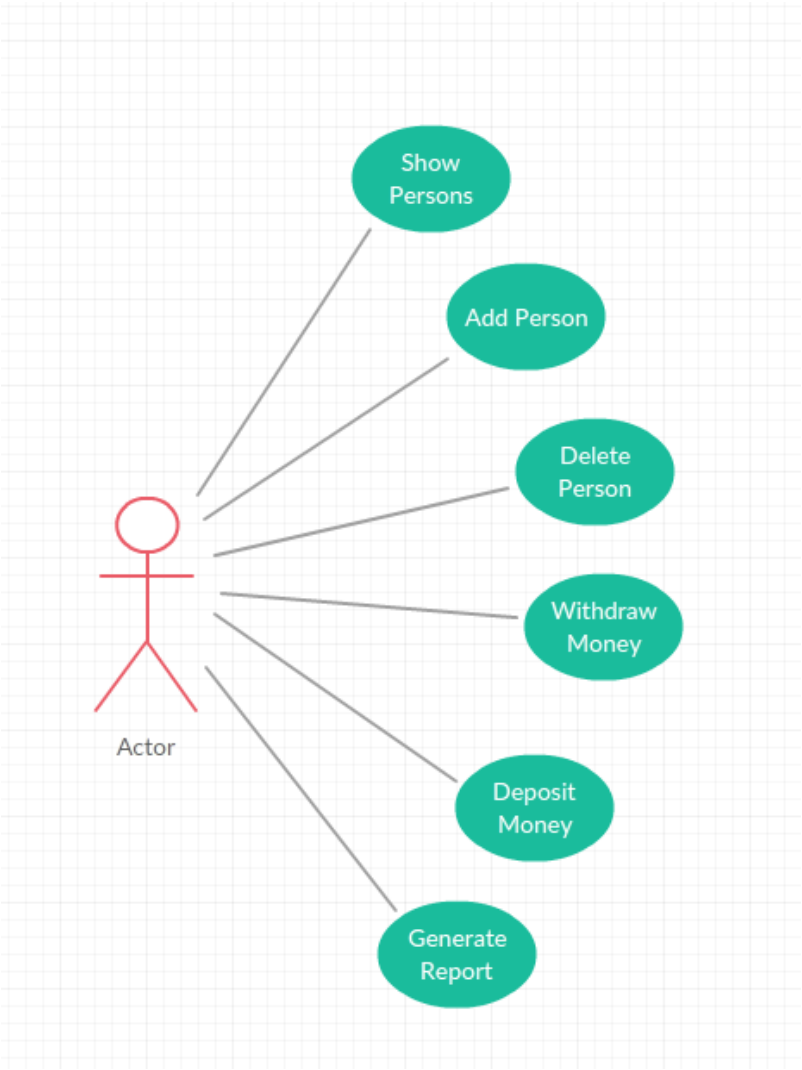
An application like this can be used by small or even medium retail companies that need to keep track of their Persons, Accounts and orders. For larger companies, probably a large-scale solution should be more optimal, something like a dedicated framework, among with several secure servers.

Based on the above, the main concept of the assignment was the HashMap, along with the concepts involved, Persons, Accounts. Also, I tried using the layer – based implementation.

This application should simulate a bank, with a hash table of Accounts, Persons and order details.

b. Scenarios, 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). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.



The program will have a basic input interface with buttons, textboxes, combo boxes and table views. One must provide a valid input, so there will be no non-positive values in the textboxes.

Use case: Add Person
Main Success scenario: open application – select Add Person – enter new person data – select Add – close application

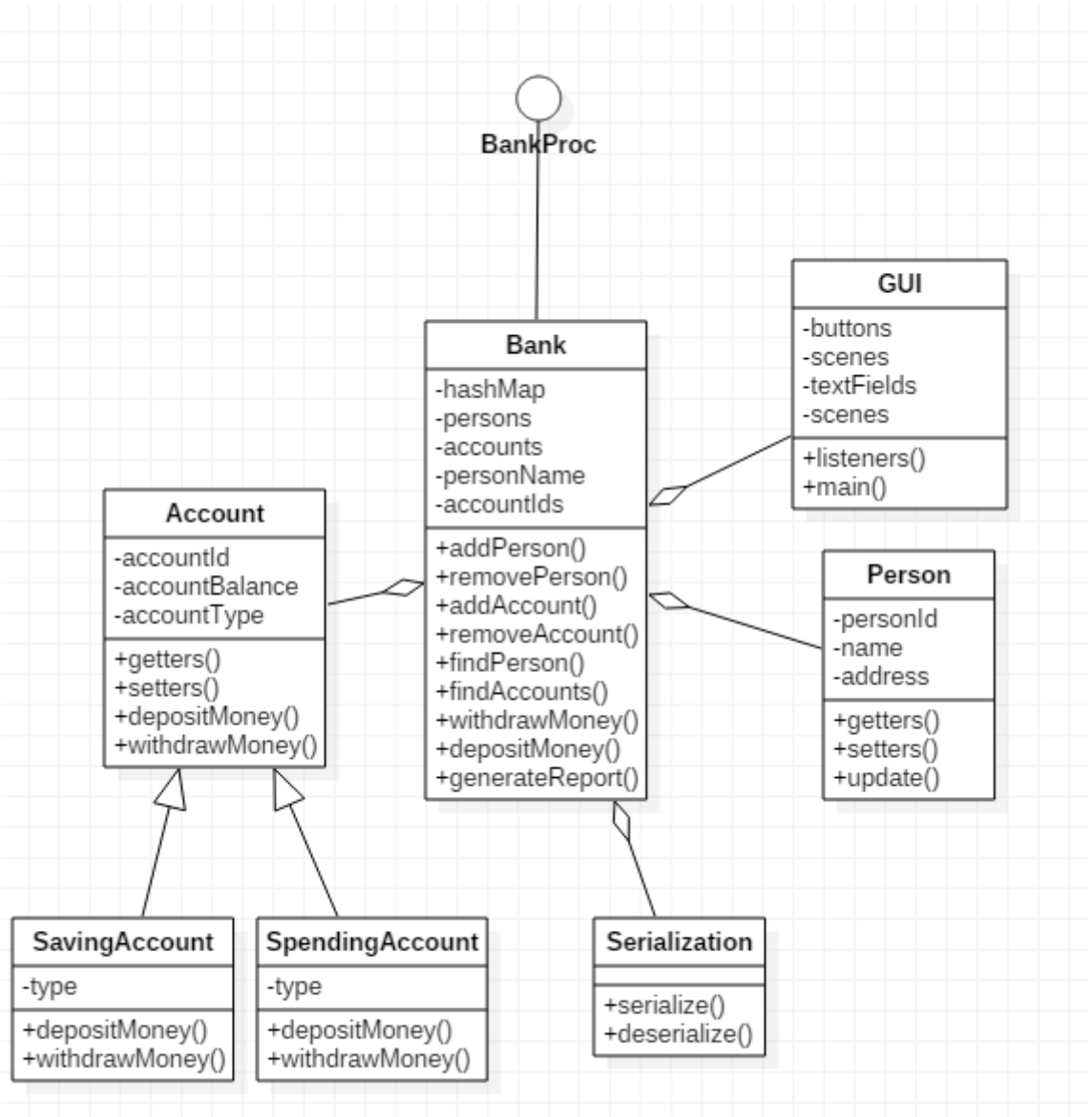
Use case: Deposit Money
Main success scenario: open application – select Deposit – select a Person – select the id of the account in which to deposit – introduce a valid amount - select Deposit – close the application

2. Design

3.1 UML Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

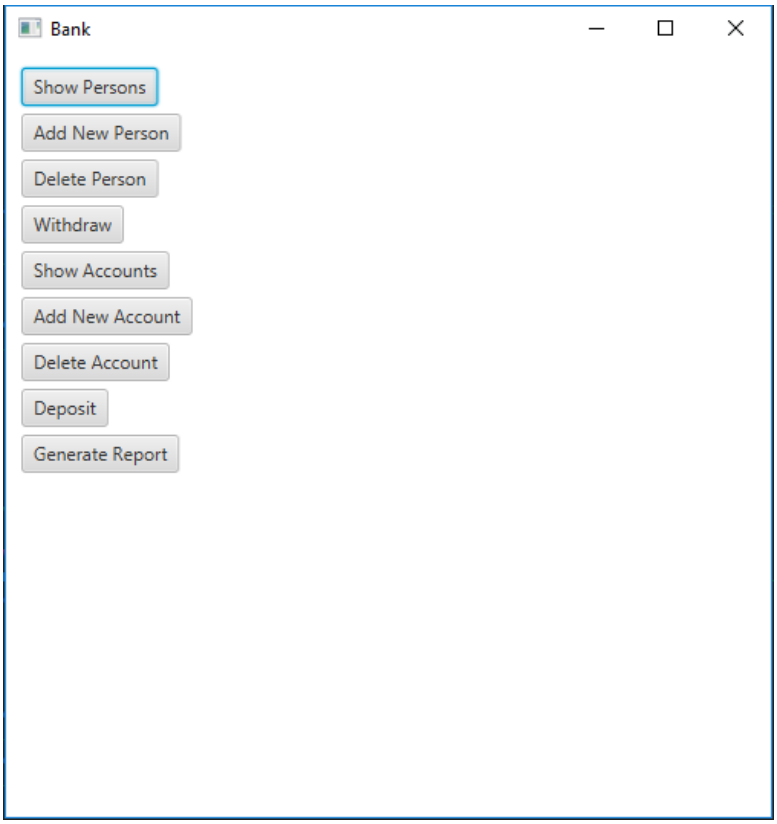
The class diagram is the main building block of object-oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code



3.4 Classes design

1. GUI Class

The “GUI” class implements the interaction with the user, also known as GUI or Graphical User Interface. The window that pops up at runtime:



2. The Person Class: **public class Person implements Observer, java.io.Serializable**

This class is the representation of a person.

The Fields of the Person Class:

- A private int field “personId” which is used to identify the persons.
- A private String field ”name” corresponding to the name of the person.
- A private String field “address” corresponding to the address of the person.

The Constructors of the Person Class: **public Person**(**int** personId, **String** name, **String** address) {}

public Person() {}

- It will initialize the above mentioned fields

The Methods of the Client Class:

- Setters and Getters method;

- update

3. The Account Class: **public class Account extends Observable implements java.io.Serializable**

This class is a representation of an account.

The Fields of the Account Class:

- A private int field “accountId” which is used to identify the accounts.
- A private double field ”accountBalance” corresponding to the sum existent in the account.
- A private String field “accountType” corresponding to the type of the account.

The Constructors of the Account Class: **public Account**(**int** accountId, **double** accountBalance, **String** accountType) {}

public Account() {}

- It will initialize the above mentioned fields

The methods from the Account class:

- Setters and Getters;
- depositMoney
- withdrawMoney

4. The Bank class: **public class Bank implements BankProc, java.io.Serializable**
   This class is the representation of a bank.

   The Fields of the Bank class:
   - Private HashMap<Person, ArrayList<Account>> - used to store the person and the accounts of the person in a hash map
   - Transient private ObservableList<Person> persons – used to store all the persons

- Transient provate ObservableList<Account> accounts – used to store all the accounts

  The methods of the Bank class:
- addPerson(Person person) – adds a new person in the hash map
- removePerson(Person person) – deletes a person from the hash map
- addAccount(Account account, Person person) – adds a new account for the person
- removeAccount(int accountId, Person person) – deletes an account from the person
- findPerson(String name) – finds the person in the hash map
- findAccounts(String name) – finds all the accounts of a person
- getAllPersons() – returns all the persons in the hash map
- getAllAccounts() – return all the accounts in the hash map
- withdrawMoney(int accountId, double sum, Person person) – withdraws money from a certain account of the person
- depositMoney(int accountId, double sum, Person person) – deposits a certain sum of money from an account of the person
- generateReport() – generates a PDF report of the bank
- isWellFormed() – check is the hash map is null or not

5. The SavingAccount class: **public class SavingAccount extends Account implements java.io.Serializable**
   Represents an account for saving money.
   The Fields of the class:
   - String type – used to store the type of the account

   The methods of the class:
   - depositMoney(double sum) – adds the sum in the account
   - withdrawMoney(double sum() – withdraws the sum from the account

6. The SpendingAccount class: **public class SpendingAccount extends Account implements java.io.Serializable**
   Represents an account for saving and spending money.
   The Fields of the class:
   - String type – used to store the type of the account

The methods of the class:

- depositMoney(double sum) – adds the sum in the account
- withdrawMoney(double sum() – withdraws the sum from the account

7. The Serialization class: **public class Serialization**
   An utility class that contains the serialization and deserialization.

   The methods of the Serialization class:

   - Serialize(Bank bank) – serializes the bank into a binary file
   - Deserialize() – deserializes the bank from the binary file

3.5 User Interface

As I've previously mentioned, the user interface is pretty much basic and easily-understood, even by non-familiarized people. When running the application, a window will open and it will provide to the user different options. This window is constructed in the GUI class using some predefined classes and instructions.

3. Using and testing the application

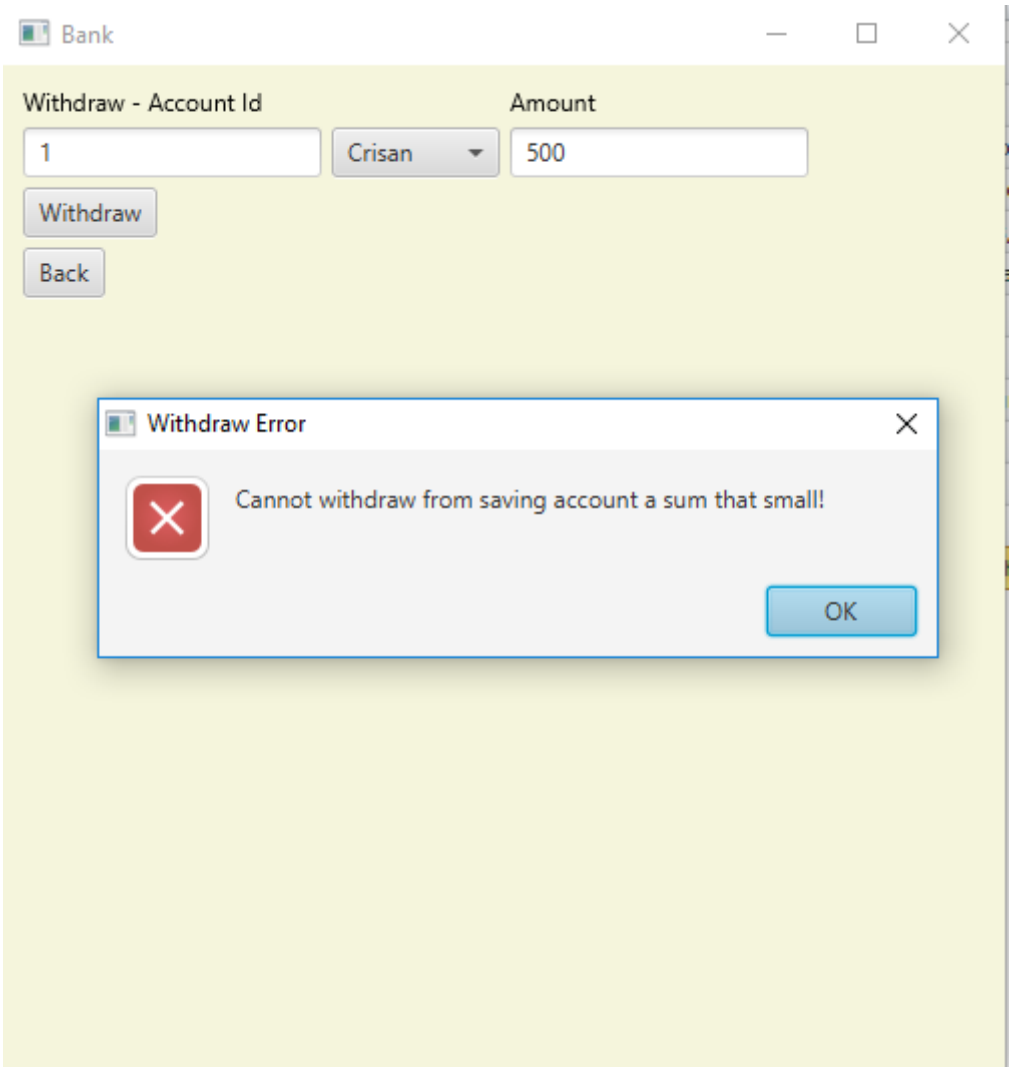
Rules for the input data:

- The input ids and sums must be given as positive integers (or doubles for the sums), respecting the intuitive labels existent in the graphical user interface.

Testing example:

1. The view of a table

2. Withdraw from saving account error



4. Conclusions

In the end, being able to implement something so dynamically and simulating such an often encountered situation of everyday life while combining everything with the concepts and rules of Object-oriented programming was a challenge with a self-rewarding result. As a further development, the first thing that should be a better exception system, especially at the input.

The problem specification presented itself as an interesting subject with many possibilities. Considering that it was the first program that used threads, the process of making was mostly based on a trial-error method and a lot of additional research which proved to be successful in the end.

Updating the graphical user interface in real time was also a challenge that needed extra attention and studying. Working with hashtables turned out to

impose a lot of additional concepts and ways of implementation on top of the basic Object-oriented paradigms.

There is still a lot of space for improvements in order to make the program more efficient. Some of the algorithms could be optimized and maybe implemented in a different way. Some constrains can be imposed on the input, meaning that more exceptions can be handled with and there can be created more statistics for the simulation. Moreover, the animation can be done using some figures and more advanced graphic. The graphical user interface can be improved. There could be added some colors or given a different layout of the components in the frame.