Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

Year of study: 2017 – 2018

Design by Contract Programming Techniques

-Assignment 4-

Ionuț Matiș

Group: 30421

Table of contents

1. Objective ……………………………………………………………………….. 2
2. Problem analysis, modelling, use cases ………………………………………... 3
3. Analysis ……………………………………………………………………. 3
4. Assumptions …………………………………………………………………4
5. Use case diagram …………………………………………………………….4
6. Scenarios …………………………………………………………………….4
7. Errors …………………………………………………………………………5
8. Design ……………………………………………………………………………5
9. Decisions …………………………………………………………………….5
10. Class diagram ………………………………………………………………..6
11. Data structures ……………………………………………………………….6
12. Class design ………………………………………………………………….6
13. User Interface ………………………………………………………………..7
14. Error handling ………………………………………………………………..7
15. Implementation ………………………………………………………………….8
16. Testing …………………………………………………………………………..13
17. Results ………………………………………………………………………….14
18. Conclusions ………………………………………………………………………14

1. Bibliography ………………………………………………………………………14

# 1.Objective

The main objective of this project is to create an application which simulates a bank. The application must have a specific system of classes.

The secondary objectives are:

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.
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).
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 (in RO. “titularul contului”). A person may act as main holder for many accounts. Use JTable to display Bank related information.
   1. Define a method of type “well formed” for the class Bank.
   2. Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions).
5. Implement a test driver for the system.
6. The account data for populating the Bank object will be loaded/saved from/to a file.

# 2. Problem analysis, assumptions, use cases

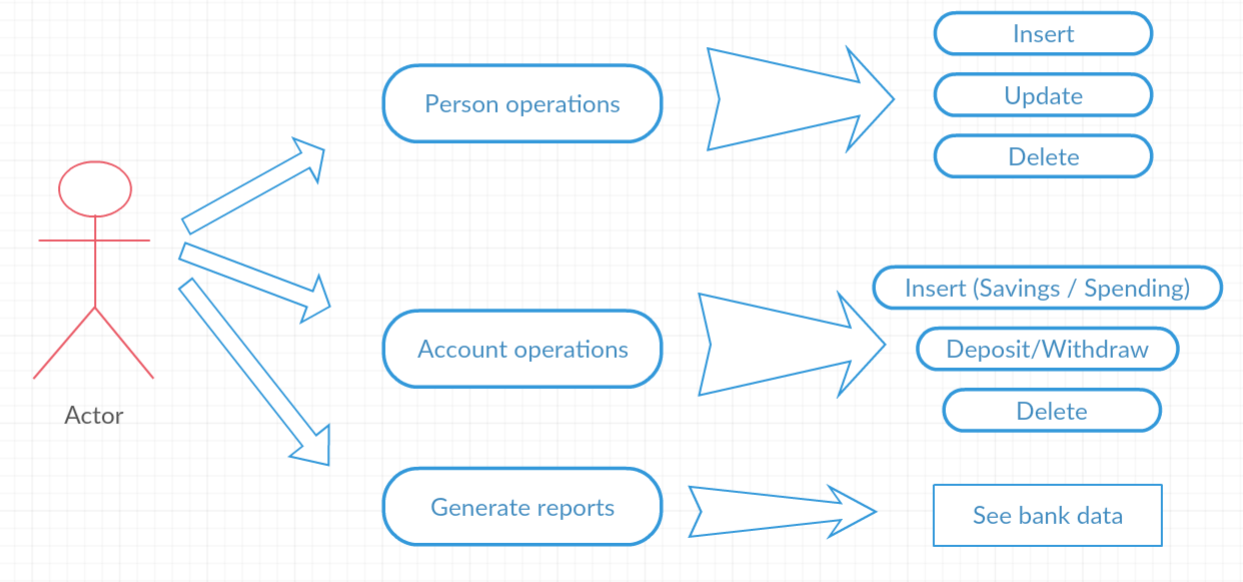
2.1 Analysis

When we think about banks, we think people which have several accounts (either for savings or spending) in which they store money. This seems rather easy to do, but the task becomes more complicated, because the bank needs to keep track of all people, all their accounts and all their virtual money they own. Moreover, the bank allows people to perform various actions upon their accounts, such as depositing or withdrawing a specific amount of money.

2.2 Assumptions

We assume that the input is correct. This means that the name of the user contains contain letters, the email and the date of birth have a specific structure (ex: [name@email.com](mailto:name@email.com)) and (ex: dd/mm/yyyy) respectively.

2.3 Use Case Diagram



From the use case diagram, we can see that the user can selects 2 types of operations (for persons and for accounts) and can generate reports of the bank, which will display all the data from it, every person which their associated accounts.

If the user chooses to work on persons, he can insert, update or delete a person from the bank (provided he insert a valid input for insertion and update).

If the user chooses to work on accounts, he can add a new account (savings or spending) to a specific person, he can delete an account from a specific person and he can deposit or withdraw a sum of money (if the sum is valid)

2.4 Scenarios

First scenario (**success**)

1. The user selects “Person operation” and “Account operations”.
2. He inserts in the bank a person with valid name, email and a date of birth.
3. He decides to create an account for this person with an initial sum equal to 0.
4. He deposits a valid sum of money (greater than 0).
5. He generates reports based on the data he inserted.

Second scenario (**fail**)

1. The user selects “Person operation” and “Account operations”.
2. He inserts in the bank a person with valid name, email and a date of birth.
3. He updates the email of the previously entered person.
4. He decides to create an account for this person with an initial sum equal to 200.
5. He tries to withdraw 300 from the account (which represents a greater sum than he has in the account) and he will get an error message.

2.5 Errors

Errors may appear in 2 places in my implementation. Firstly, when you work on persons, you will get an error if you try to insert duplicate data (persons with the same name and the same date of birth) or try to update / delete a person that is not in the database.

Secondly, you will get an error if you try to create an account (savings / spending) with a negative starting sum, try to deposit / withdraw a negative amount of money or try to withdraw more money than you actually have in the account). The difference between the Savings and Spending account is the fact that the Saving account allows a single large sum deposit and withdrawal and computes an interest during the deposit period whereas the Spending account allows several deposits and withdrawals but does not compute any interest. An error will appear if you want to perform several deposits / withdrawals on a Savings account.

# 3. Design

3.1 Decisions

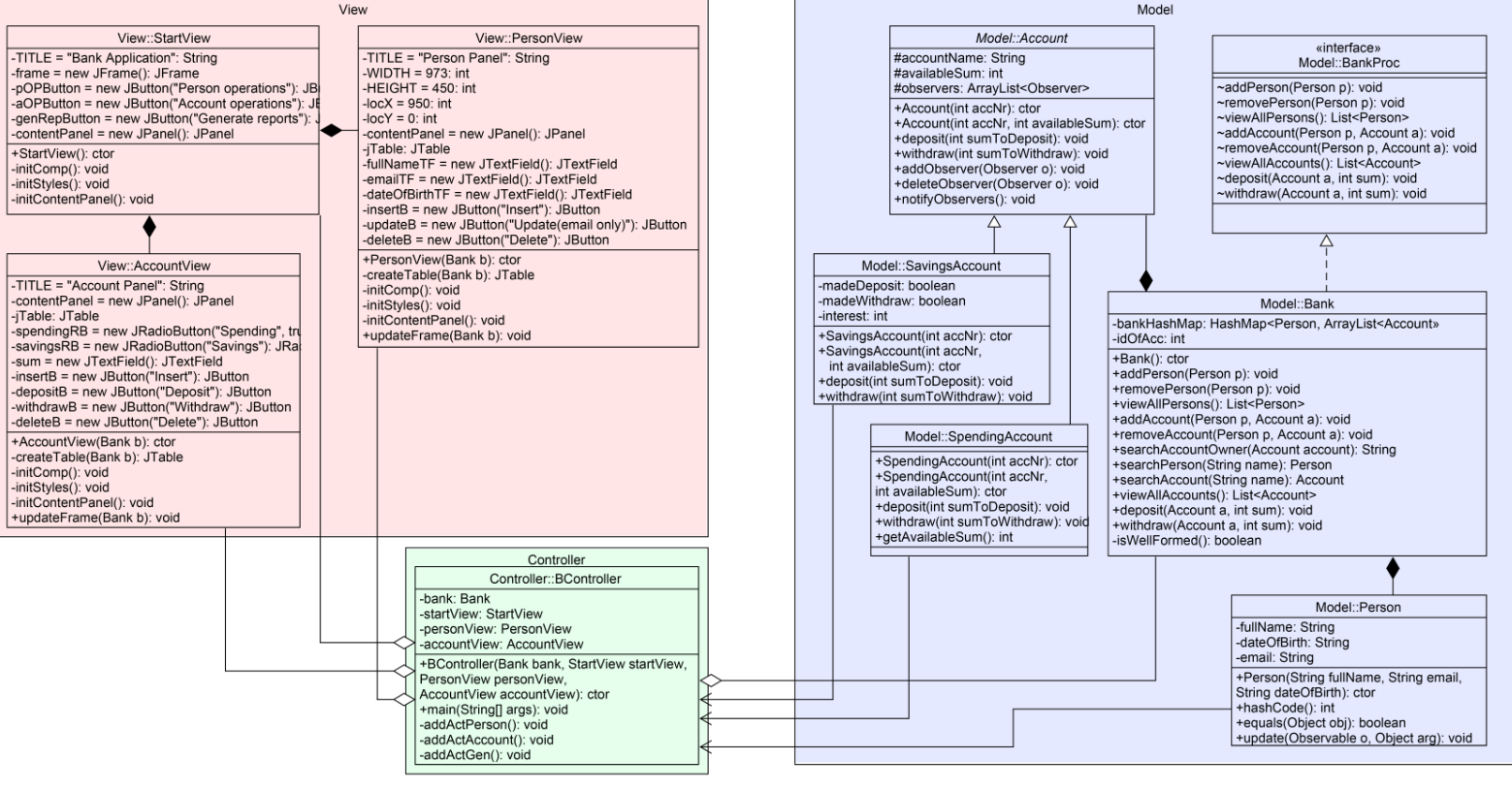
The application has a Model View Controller pattern, because it separates the logic (Back-end) from the visualization (Front-end) and then combines them harmoniously.

The Model represents the logic of the application and the way objects interact with each other. It can also have logic to update controller if its data changes.

The View represents the visualization of the data that model contains.

The Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

3.2 Class diagram



3.3 Data structures

In the bank class I have used as data structure the HashMap (Hash table based implementation of the Map interface). This implementation provides all of the optional map operations and permits null values and the null key. (The HashMap class is roughly equivalent to HashTable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map, so it does not guarantee that the order will remain constant over time. I used this data structure, because I had to associate keys (Persons) to Values (Accounts), so a HashMap was perfect for this operation.

ArrayList was also used intensively, because a Person may hold multiple accounts, so I decided to store them in such a data structure.

3.4 Class design

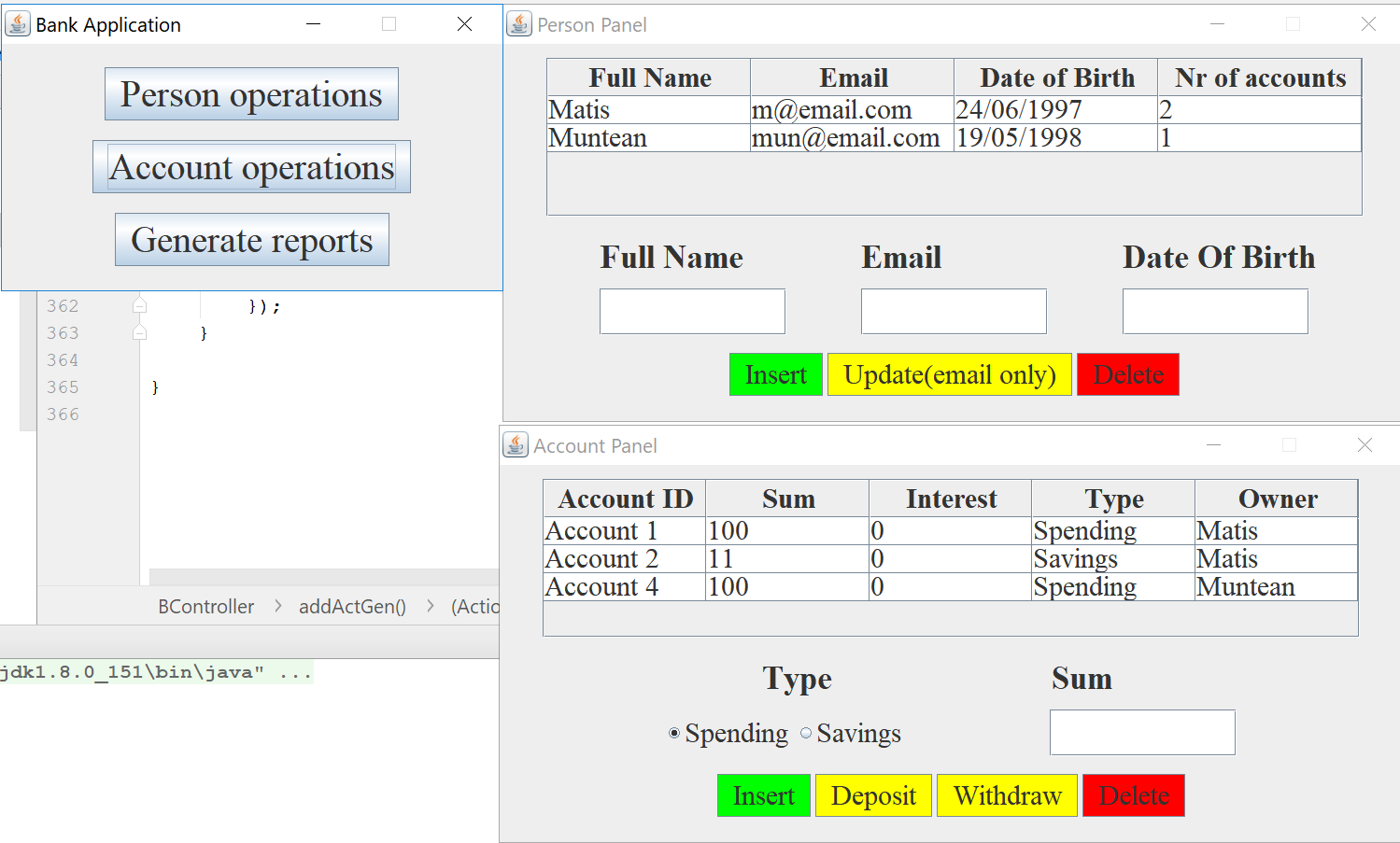
For the design, an OOP implementation was chosen with a Model View Controller pattern.

**Model package:** contains the Person class which overrides the methods hashcode and equals. It also contains the abstract class Account which will be extended by SavingsAccount and SpendingAccount (and also implement its abstract methods). This class has protected fields (accountName, availableSum), so the classes that extend this can use the variables directly. The interface BankProc and the class that implements it (Bank) are also found here. The bank class provides methods for BankProc which are essential for this project (ex: addPerson, removePerson, addAccount, deposit, withdraw etc)

**View package:** contains the graphical user interface classes (3 in total). The main GUI class is the StartView, which opens as the user start the application. From it, the user can select 3 actions – to open the PersonView, to open the AccountView or to generate reports of the bank. After the user chooses an option, he can perform several actions (add or remove a person, add or remove an account, deposit or withdraw some money from a specific account).

**Controller package:** contains the controller of the project. It has object from model package as well as object of the GUI package. In this class I add action listeners to all my buttons and serialize, deserialize the information held in the bank.

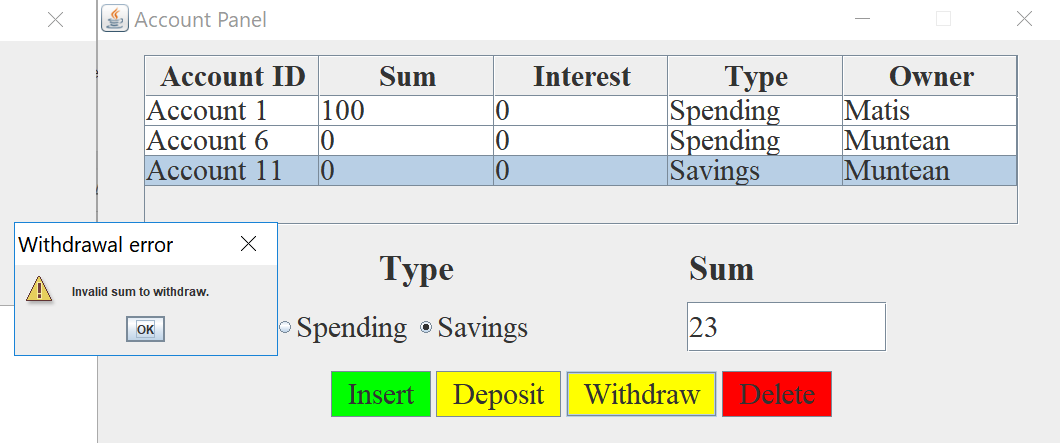
3.5 User interface



This picture shows the User Interface. The left panel has 3 buttons. The first 2 open the left panels. They have similar structure – table to display the information and fields / buttons to insert specific data. The account panel allows the user to select the type of account they want and provide buttons for deposit / withdraw.

3.6 Error handling

Error handling is done using JOptionPanes which display a message in case the user inserts invalid data or tries to do a forbidden action. This method of reporting errors is very user friendly and very efficient.

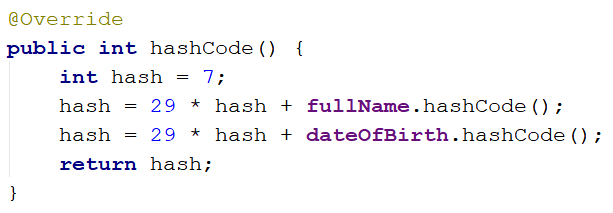
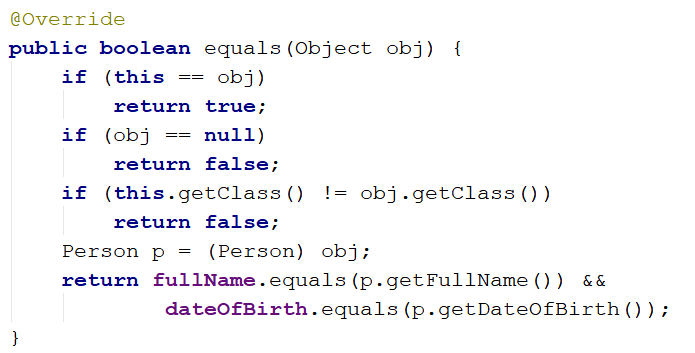


The above error is reported because the user tried to withdraw 23 from the Account 1, which has an amount of 0.

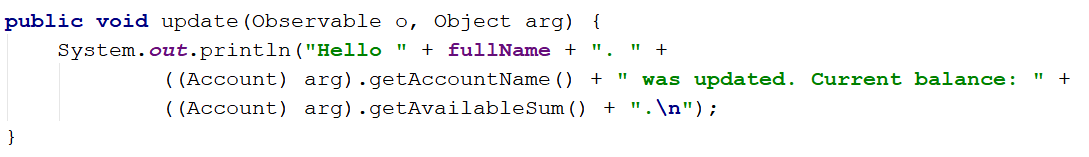
# 4. Implementation

1. **Person**

This class has 3 fields (fullName, email, dateOfBirth) and implements 2 interfaces: Serializable and Observer. What is interesting about this class is the fact that overrides the hashcode and equals methods.

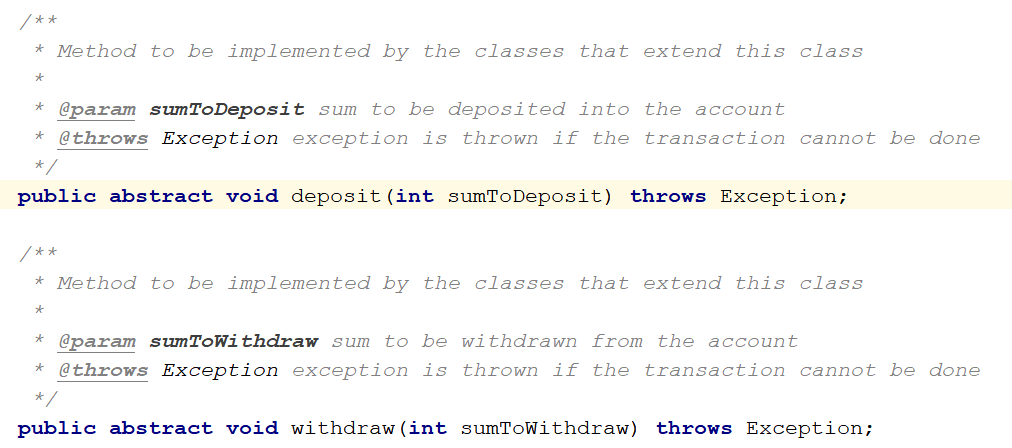
The hashcode of the object is obtained by a formula which takes into consideration only the fullName and the dateOfBirth (because the email can be updated and doing so will break the hash table). Equals method performs the same – it takes into consideration only the name and the email.



Update method for the Observer which will be notified from a class that extends Observable.

1. **Account**

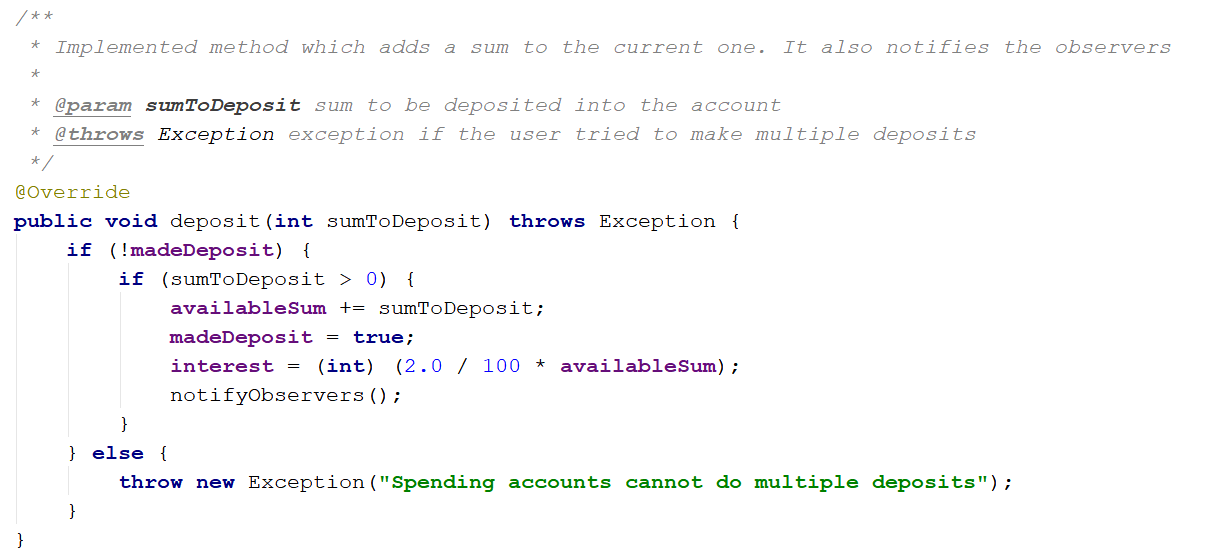
This abstract class has 3 protected fields (to be available also for the classes that extend this one): accountName, availableSum and an Array List of observers.



It has 2 abstract methods to be implemented by the subclass.

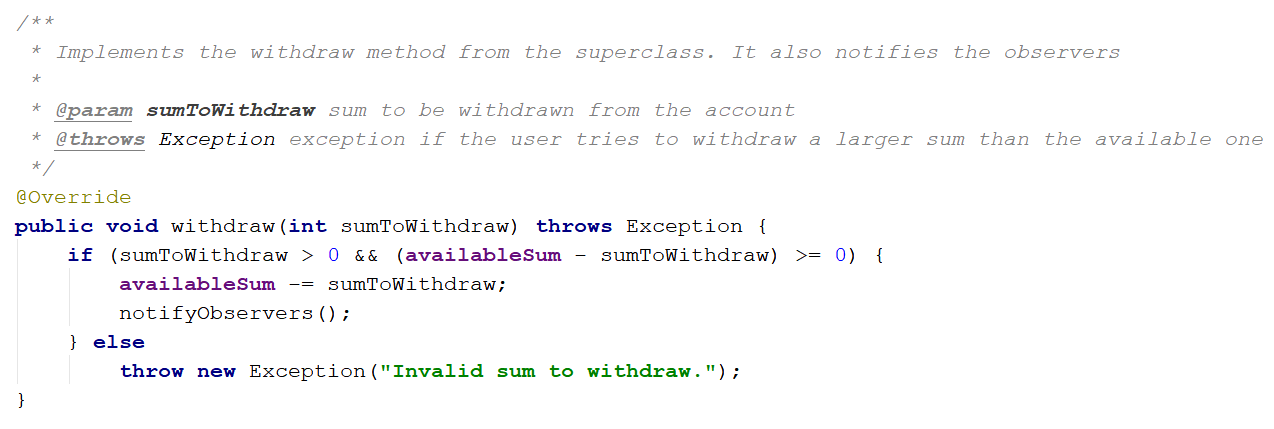
1. **Savings account**

Class that extends the abstract class Account. It provides methods for deposit and withdraw. It has a unique feature - allows the user to perform only one deposit and only one withdrawal. This is done through the boolean variables madeDeposit and madeWithdraw. An implementation of the deposit method can be seen beneath.



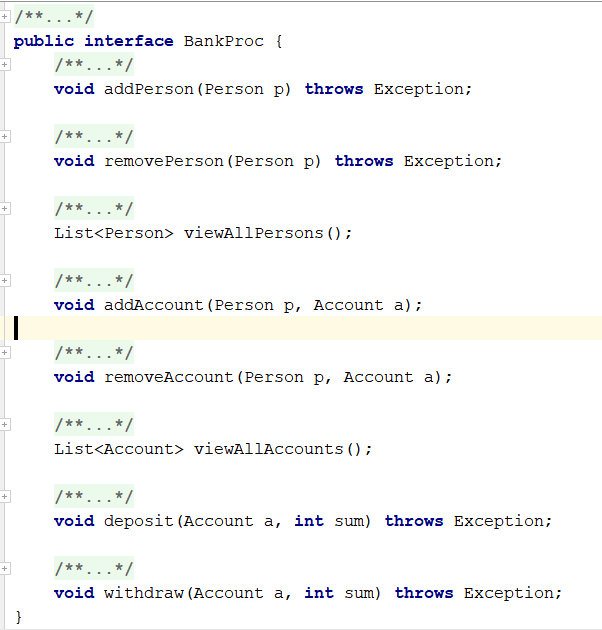
1. **SpendingAccount**

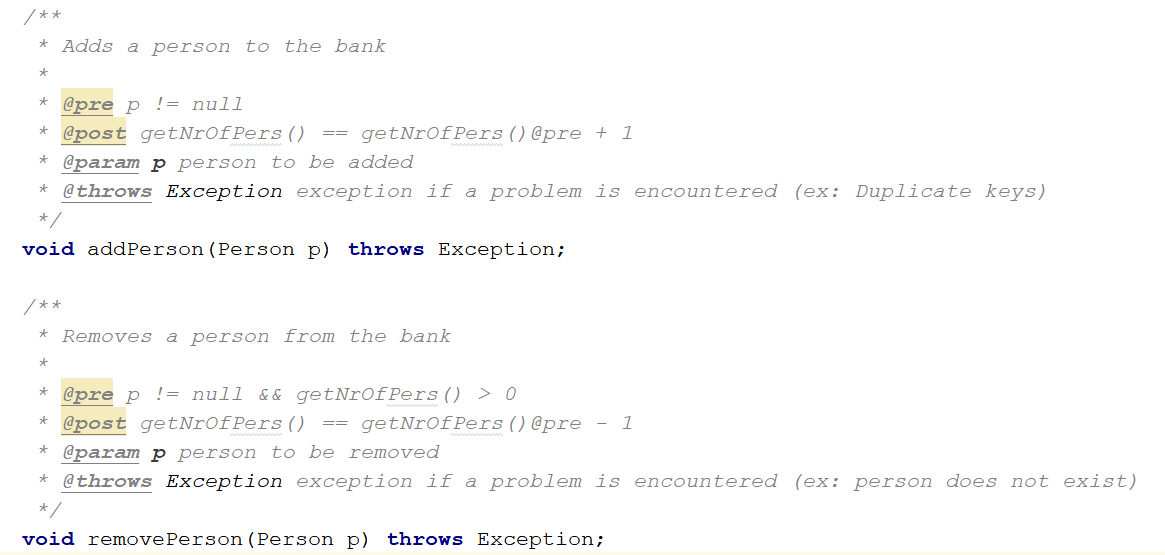
Class that extends the abstract class Account. It provides methods for deposit and withdraw. It allows the user to perform multiple deposits and withdrawals. An implementation of the withdraw method can be seen beneath.



1. **BankProc**

Interface which proposes methods to be implemented by a Bank class. There are also specified the pre and post conditions for the interface methods.

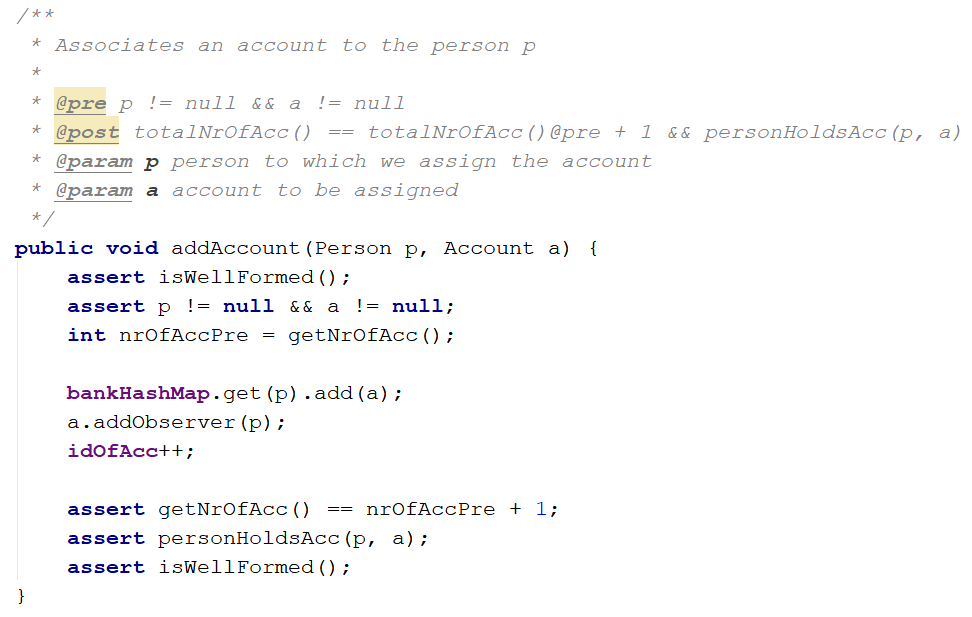


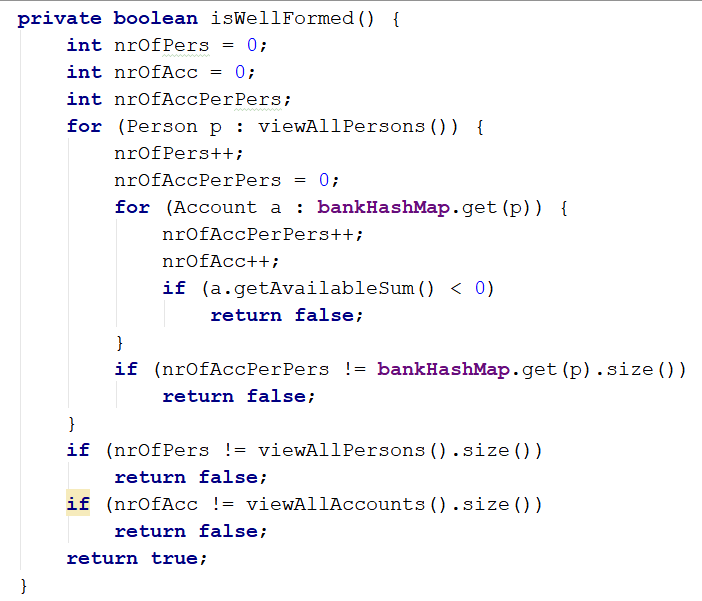


Pre and post conditions for every method to be implemented.

1. **Bank**

Class that implements the BankProc and Serializable interface. It provides methods for the interface, has an invariant and uses assertions. As an important field, it has a HashMap (the keys are the Persons and the values are the Accounts). Example of method that adds an account to a Person p and uses asserts based on the pre and post conditions.

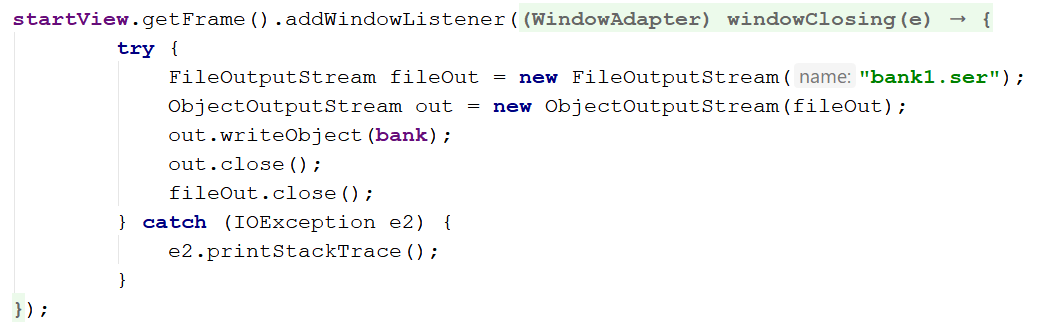




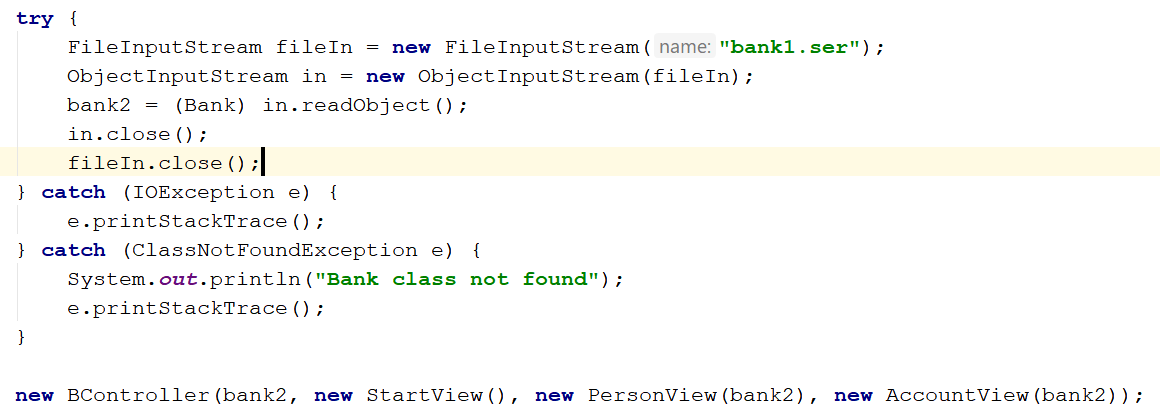
This method represents the invariant of the class. It iterates through the HashMap and checks whether the number of persons / accounts correspond to the actual number of person / accounts. It also checks if every account has an available sum greater than 0 (cannot have negative balance).

1. **BController**

This class represents the controller of the project. The serialization and deserialization processes are done in this class. It also adds action listeners to all buttons from all panels of the GUI.



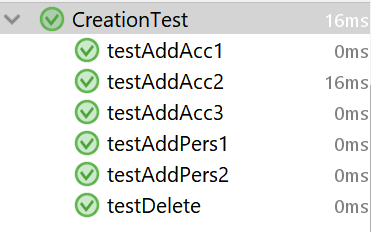
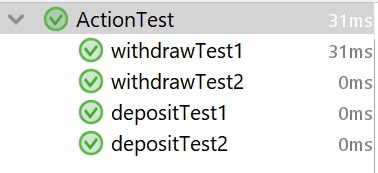
The serialization process is represented above. After the client closes the application (presses x button), the changes made in the bank are stored in a file, so the user does not lose any data.



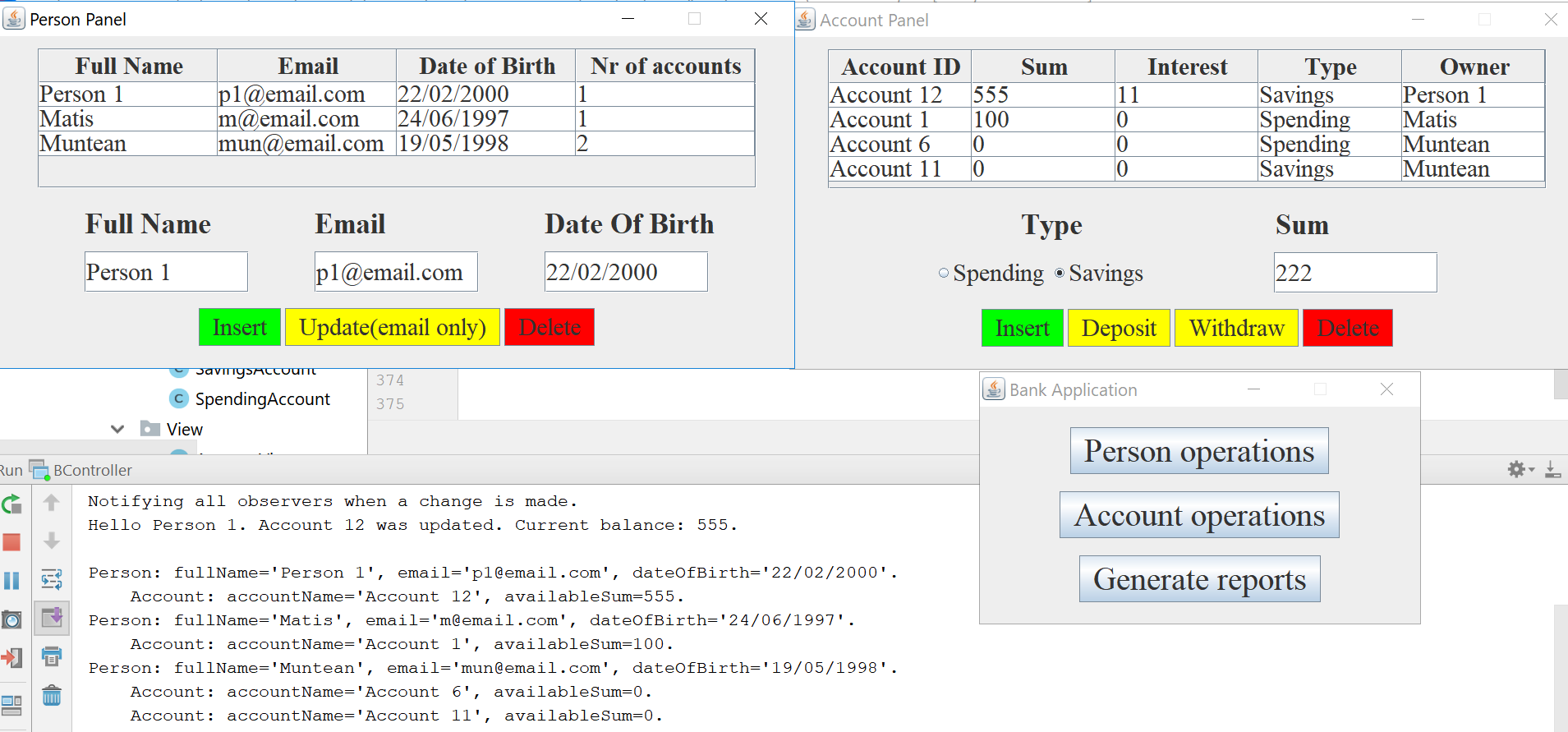
The deserialization process is represented above. This is done in main, just before launching the application.

# 5. Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method to be tested | Input | Expected result | Actual result | Pass/  Fail |
| addPerson | P1: Person 1, [p1@email.com](mailto:p1@email.com), 22/10/1990 | P1 was added to the bank | P1 was added to the bank | PASS |
| addAccount | P1: Person 1, [p1@email.com](mailto:p1@email.com), 22/10/1990  A1: SpendingAccount 1 | P1 holds A1 | P1 holds A1 | PASS |
| deletePerson | P1: Person 1, [p1@email.com](mailto:p1@email.com), 22/10/1990 | P1 was deleted from the bank and the nr of Persons decreased | P1 was deleted from the bank and the nr of Persons decreased | PASS |
| deposit | A1: Spending Account 1, sum = 200;  Deposit 200 | A1 holds 400 | A1 holds 400 | PASS |
| withdraw | A2: Savings Account 2, sum = 100;  Withdraw 50 | A1 holds 50 | A1 holds 50 | PASS |

# 6.Results



Add a new Person (Person 1), create an account with an initial sum of 333 for it (Account 12). Deposit 222 in it -> the observer will be notified because a change has been made. Then generate reports for the bank.

# 7.Conclusions and further developments

In conclusion, the main and secondary objectives were met. The application works as intended. I have learnt a lot of new things working on this assignment, such as how to design by contract (using pre, post condition and assert), how to use the Observer design pattern, how to use a HashMap and override hashcode and last but not least how to use serialization. I think it would be of great help in the future.

Further developments: make the generated reports display a larger amount of information and better statistics.

# 8.Bibliography

<https://stackoverflow.com/questions/1026723/how-to-convert-a-map-to-list-in-java>

<http://www.codejava.net/java-se/swing/jradiobutton-basic-tutorial-and-examples>

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JRadioButton.html>

<https://stackoverflow.com/questions/32051657/how-to-perform-action-after-jframe-is-closed>

<https://www.tutorialspoint.com/java/java_serialization.htm>

<https://creately.com/>