Documentation

Bank

Homework number: 4

Due date: week 12

# Purpose

The purpose of this project is to create a project using the “Design by contract” design pattern. The project should simulate the behavior of a bank, where users can deposit or withdraw money. Each user can have several accounts. We are required to create a good design and to implement it in Java.

# Problem analysis

<Modelling, scenarios, usage>

Modelling the problem is an abstraction of the every-day usage of banking systems. We can identify users or clients and their accounts. It is the duty of the bank to store this information and to allow users to interact with their accounts.

The most common operations are demonstrated below:

A frequent use-case would be the following:

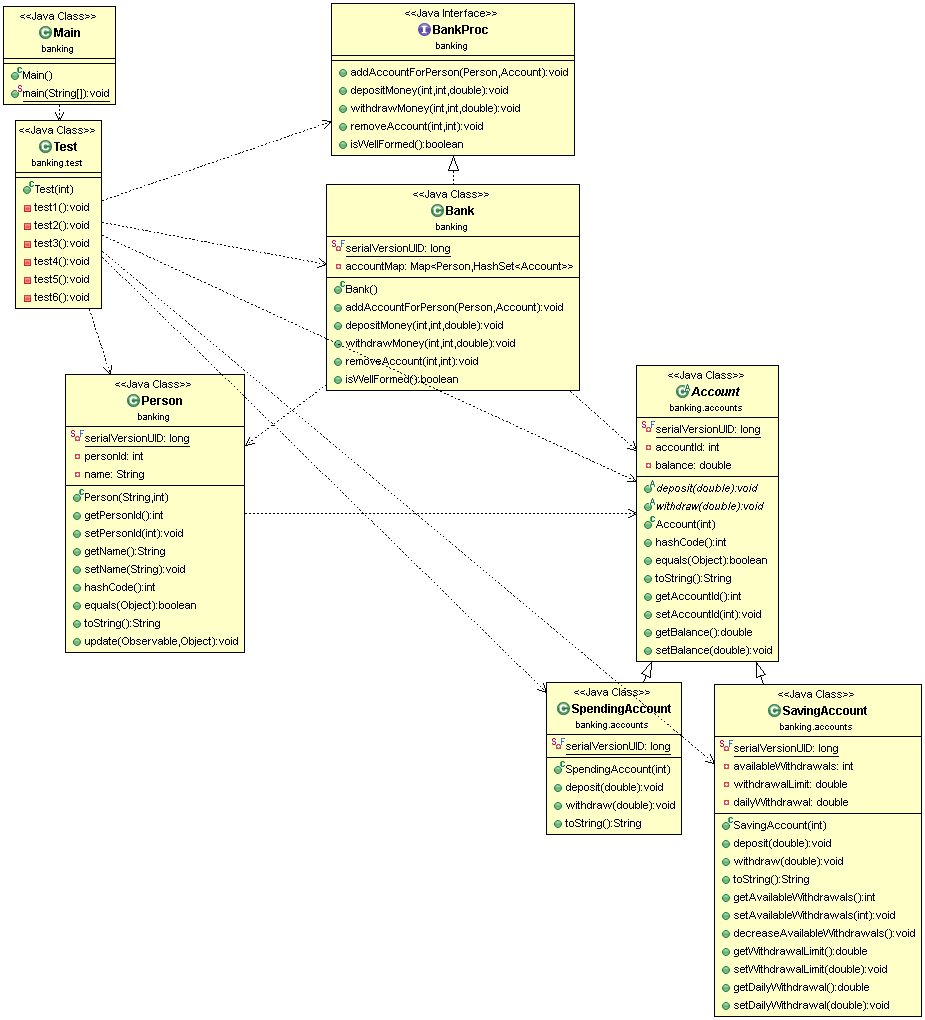
The last two operations can be performed multiple times. A user can also have more than one account.

# Design

<UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface>

Each model abstracted from real-life objects (bank, account, person) will have its own class.

The class diagram is presented below:



I followed the Design by contract design pattern and created the *BancProc* interface which defines the methods that a bank must implement. Each method is well documented and includes pre- and post-conditions, that have to be checked or fulfilled by the class that implements the interface (Bank).

An example of providing a rich JavaDoc comment is presented below:

/\*\*

\* The fields of all the parameters should be initialized.

\* Searches for an already existing Person and adds the account

\* to it. Otherwise stores the new Person.

\*

\* **@pre** isWellFormed(); person.personId != null; account != null &&

\* should not already exist

\* **@post** isWellFormed(); number of persons same or greater with one

\* then before; number of accounts greater than before

\*

\* **@param** person

\* the Person to be added

\* **@param** account

\* the Account to be added

\*/

**void** **addAccountForPerson**(**Person** person, **Account** account);

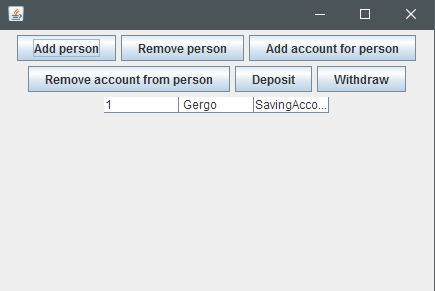
Sometimes not all conditions can be met, because errors an occur or some of the input information is incorrect. For these cases, I’ve implemented Exceptions, which indicate transaction failures.

The “super” exception class is called WithdrawalException and the following other exceptions extend it:

* NotEnoughFunds
* nrOfWithdrawalsExceeded
* WithdrawalLimitExceeded

These classes have nothing to do except to indicate the reason of the error and to print the error message.

The user interface is simple, yet powerful. All the required actions are included in form of buttons. When pressing a button, a new window is opened and we are required to enter the needed information (eg. account ID or sum). The user can also list all accounts and persons in a table. See the example below:



# Implementation and testing

To test my program, I’ve implemented a test class, that performs various operations (including illegal ones). I tried to include all the cases that might occur during the usage of such a program. These are the following:

* Add accounts and persons
* Deposit money in different accounts
* Withdraw money from different accounts – the program should limit the user and check for illegal operations

# Results

# Conclusions

<What have I learned, further improvements & development>

# Bibliography

* <http://www.tutorialspoint.com/java/java_serialization.htm>
* <https://examples.javacodegeeks.com/core-java/util/observer/java-util-observer-example/>