*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department of Computers*

*2nd Year, 2nd Semester 2015-2016*

Programming Techniques

Homework 1

Student: Robert Văcăreanu

Group: 30424

CONTENTS

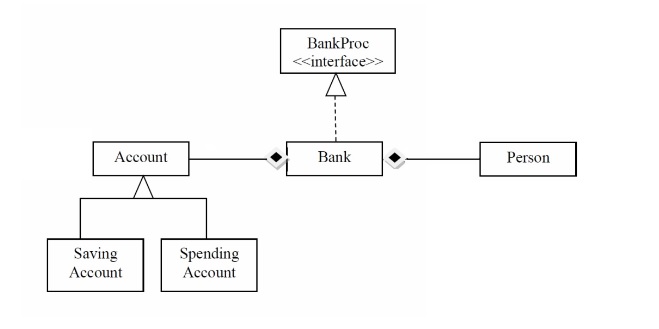
1. Project specification
2. Problem analysis, model, testing and usage
3. Design
4. Results
5. Conclusions
6. Bibliography
7. Project Specification

**Objective**

Design by Contract Programming Techniques

**Description**

Consider the system of classes in the class diagram below.



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 (ro. titularul contului). A person may act asmain 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. Implement a test driver for the system.

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

1. Problem Analysis, Model, Testing and Usage
2. Problem Analysis

In order to fulfill completely the requirements and, at the same time, obey to the object oriented programming paradigm we choose an architectural design pattern, namely the layered architecture design pattern. Since we do not have any persistence layer due to the fact that the application is not connected to any type of database , there will be only 3 layers, namely: presentation, business and model,

For the bank class, we choose to represent the person and the accounts in a generic hash map: HashMap<Person, Set<Account> accounts>. In this way, we ensure not only a fast access but also a consistency due to the fact that no person will be able to add the same account twice.

1. Model

The package **model** contains 4 classes: Account, Person, SavingAccount and SpendingAccount.

The class Person models the generic person that can hold an account to an bank. Therefore, some fields of this class includes: long id and String name. There could be also more fields, such as: age, gender and so on, but the one included are the minimum data requirements. Besides the usual getters and setters, there are overridden the methods hashCode and equals. Since this class is implementing the Observer interface provided inside the jdk, we also overridden the method update. This class implements Serializable as well since it is needed when we save to file the class bank, that contains persons.

The class Account is an abstract one that extends Observable and implements Serializable. Since we do not need to extend anything else, we decided to straightforward extend Observable instead of creating a new class and using composition. Serializable is needed in order to be able to save to disk the data. It contains three fields of crucial importance for modeling accounts: genericId used for keeping track of the number of accounts we created, id used to uniquely identify an account and amount used for identifying the amount of money that account have.

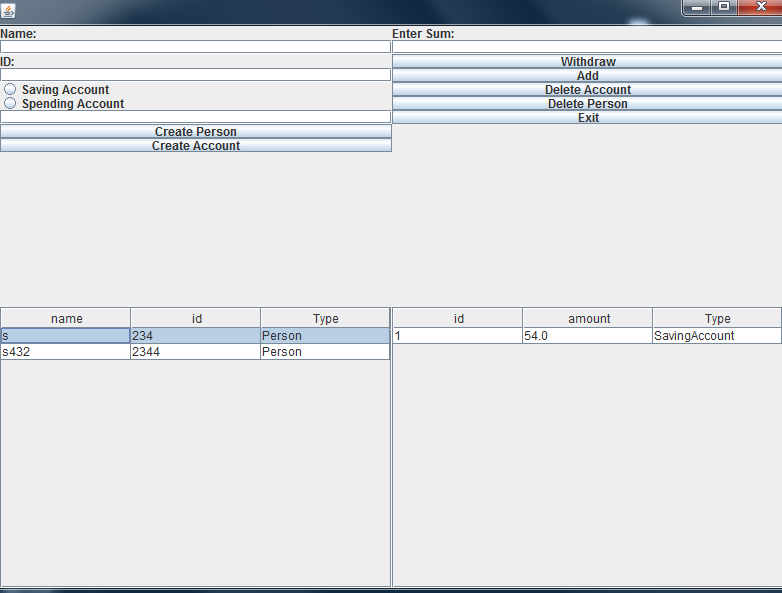
Further, we differentiate two types of accounts: a saving one and a spending one. The difference between them is when we try to add or withdraw money to or from them. This mimics the real life scenarios, where you don’t withdraw money from a saving account until the very end of the contract.

1. Testing

The correct functionality of the application was proven by multiple tests that covers most of the situation a consumer will encounter in mostly any situations. Also, the testing was done by making use of the J unit test cases. In this way we tested the operations that are the most performed inside our application, namely: add person, delete person, add account to the specified person and delete account of the specified person.

1. Usage

The application comes with a UI that is divided into 4 sections of interest: Top Left, where you add accounts and persons, Top Right where you make operations on the accounts, bottom left where you see the persons and bottom right where you see the accounts



1. Design

As mentioned above, we will split the implementation into three packages, following a **three layers** type of architecture. This favors the re-usability of the modules and keeps everything well organized

The whole application is split into three big packages, along with the default one that is used only to keep the main method in it. However, this can be easily changed since that package doesn’t provide any functionality at all.

Those packaged are:

Presentation

Here is the class MainWindow where we have put all the GUI stuff. It is only a class because we decided to keep the UI as compact as possible. Hence there is only one window with all the functionality there. This is due to the fact that the application is not complex enough to need more screens or classes.

Besides some anonymous inner classes to trigger some functionality that makes a lot of sense to keep there, all the action listeners are added from the business, namely from the Manager class. This is done in order to obey to the three layers architecture specifications.

In order to populate the tables, we used reflection as specified in the requirements. The whole job is divided in the methods: populate table, retrieve data and get fields. The job of populate table, as the name implies is to take the data given in the argument list and to make a table out of it, without prior knowledge. Retrieve data, on the other hand receives as arguments the generic collection and give back a matrix of data. Get fields use reflection in order to get the name of all fields . To note that without the getters would not be possible with this approach to correctly populate the table. This is due to the fact that we explicitly invoke the methods.

The package: **model** was already discussed above.

The package **business** contains the logic that drives the application. There are 3 classes, of which one is an interface that is implemented by the class Bank. This is done in order to fulfill the requirements of a bank: add persons, delete persons, add money to account and withdraw money from account. There is no checking done to see if there is enough money because is not unusual to withdraw more than you have in the account. In this way, you have a debt to the bank that you have to pay. The correctness of the class bank is enforced by preconditions and postconditions as well as a methd that ensures that every actions maintains the well formed property of the data.

The class Manager is used to control the bank object as well as to add the action listeners to the main window class. Also, here is done the saving and the loading implemented by using the serializable interface. To note the fact that while we implement the serializable interface to all the needed classes, we are not overriding any method. This is because the serializable interface is actually a marker one. At run-time the java virtual machine sees the classes that implemented the interface and allows the saving/loading.

1. Results

The end result is an application capable of simulating some bank transactions as well as saving and loading from file by using the java serializable interface.

1. Conclusions

While the difficulty depends on the model and the approach, it also depends on how many features the developer wants to add into its application. However, I evaluate the difficulty of this project somewhere between easy and medium. This is because due to the Serializable interface the whole saving part is trivial. The Observer design pattern is pretty easy to comprehend and it is supported by the jdk. While there are some bad approaches such as making the Observable a class instead of an interface. This is bad due to the fact that because java doesn’t support multiple inheritance you lose your only extends. This can be easily overcome by composition. However, I feel like this should be done in the first place. If there are problems that anyone would run into, probably would be due to the bad design choices of the Observer implementation. For example, the class does not implement the serializable interface and in order to save the list of observers you must: either design from scratch ( best ) the observer design pattern and implement it or override some methods.

Hence, lots of possible problems encounter by a developer on this task are partially or totally solved by Java.

By implementing this project, I revisit my knowledge about hash maps and observer design patern.

1. Bibliography
2. Materials from last year course
3. <http://docs.oracle.com/javase/tutorial/java/>