March 2016

Laboratory Assignment 4

Bank

Farcas Silviu Vlad

30425/1

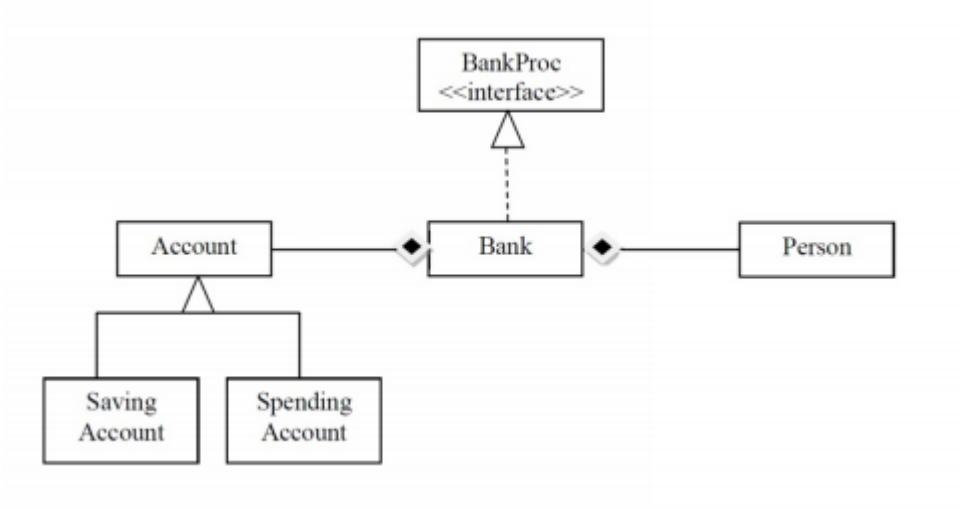
Contents

1. Introduction

1.1 Task Objective

The task of this assignment is defined as follows:

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

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. Implement a test driver for the system.

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

Our bank application should empower the administrator of the bank to add persons, delete persons, add accounts, delete accounts and generate reports about bank activity. The client (or person) should be able to safely deposit or withdraw money, according to its type of account (Savings or Spendings Account). Also, it should be able to view info about the account and to access reports about the account.

1.2. Personal Approach

The aim of this paper is to present one way of implementing the required system, based on a simple and minimal GUI that provides the user with all possibilities described above. The UI will provide the user with

logging based on its name (no password required). If the user is admin, it will have at their discretion the management of all bank related information: person (add/remove), accounts of each individual person (add/remove), and generation of report. In our implementation, the report shows all operations ever to be done on the bank. If the user is a client (person), it will be able to deposit/withdraw from his account (provided the funds permit withdraw operation) or generate reports based on account history.

2. Problem Description

2.1. Problem Analysis

The bank – one of the most evil institutions man has ever known (some may say…). Either we like it or not, the bank is present everywhere, in our daily lives, in the world we live in. Every transaction possibly passes through a bank. The bank is also used for tax-evasion by tycoons flushing capital in Cayman Islands or something, but we let that part to the competent institutions…

The basic operations present in our bank system will be differentiated on whether the user logs in as administrator or as regular client. The logging is by their name. These operations will be:

- for admin: add person (followed necessarily by add account, since a person becomes a client not just by registering, but also by creating an account), remove person (and by default all associated accounts), add account, remove account, generate reports of global bank operations.
- for user: deposit money, withdraw money (if funds permit), generate report based on account history

The account operations will be differentiated based on whether the account is a Spending Account or a Savings Account. For a Spending Account, the owner is allowed to withdraw any amount of money (provided they have the funds). For a Savings Account, it is necessary that the balance be kept above the minimum defined in the Bank class. Also, on a daily basis, the interest will be calculated based on the current balance (the interest rate is defined in the Bank class). Also, when withdrawing, a commission rate will be applied, which is also defined in the bank class.

2.2 Modeling

The Bank has information about persons and a list of accounts for each person (savings account/spending account). Also, the bank decides upon the commission rate for withdrawing, the interest rate, the minimum balance amount and the administrator name of the system (in our example, there is only one administrator). Also, the Bank implements the interface BankProc, which defines the base procedures a bank is responsible for: add/remove persons, add/remove account.

The Person has name, and an age. It is all we know (in this implementation) about the person.

The Account has a unique id, an opening date, an expiring date, and the money effectively. There are several operations we can perform on an account: deposit, withdraw and generateReportHolder. The Account can be a Savings Account or a Spending Account. The Savings Account has an additional operation of calculateInterest, which is called whenever several days pass.

The operations on Bank and on Customer can throw several exceptions: NotEnoughFundsException or IllegalOperationException.

2.3 Scenarios

When on initial screen, the user can log in either as an admin or as a client. If they log in as

- an admin: they will be able to visualize information about persons. They will also be able to generate reports about general bank info. If they select a person, they will be able to visualize account info related to that person (View Accounts) or delete that person. If they want to add a person, they will press Add after they entered the data. They will be redirected to creating a new account for that person, since no person can be added without an account. When adding an account, they will provide the type of account and the initial balance. Provided the operation is successful, they will see the results immediately in the table.

- a user: they will be able to view basic info about accounts in the form of a table. They will be able to deposit or withdraw on the selected account, depending on the account type. They will also be able to generate pdf reports based on selected account.

2.4 Use cases

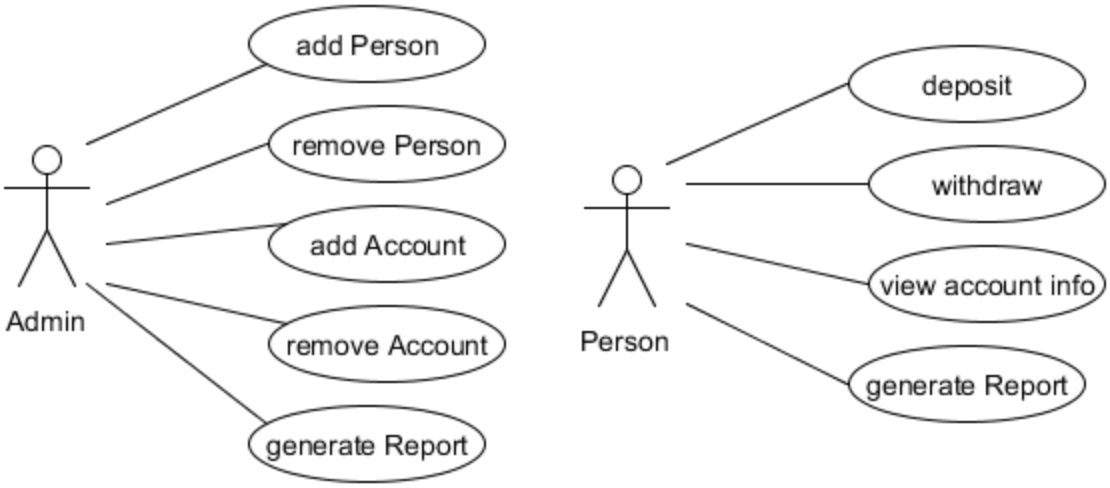
The user can deposit or withdraw, or can view info about the account. He can also generate reports of account history. The admin can add and delete persons, add and remove accounts and generate report of bank history. He can also view person info and person's account related info.

3. Design

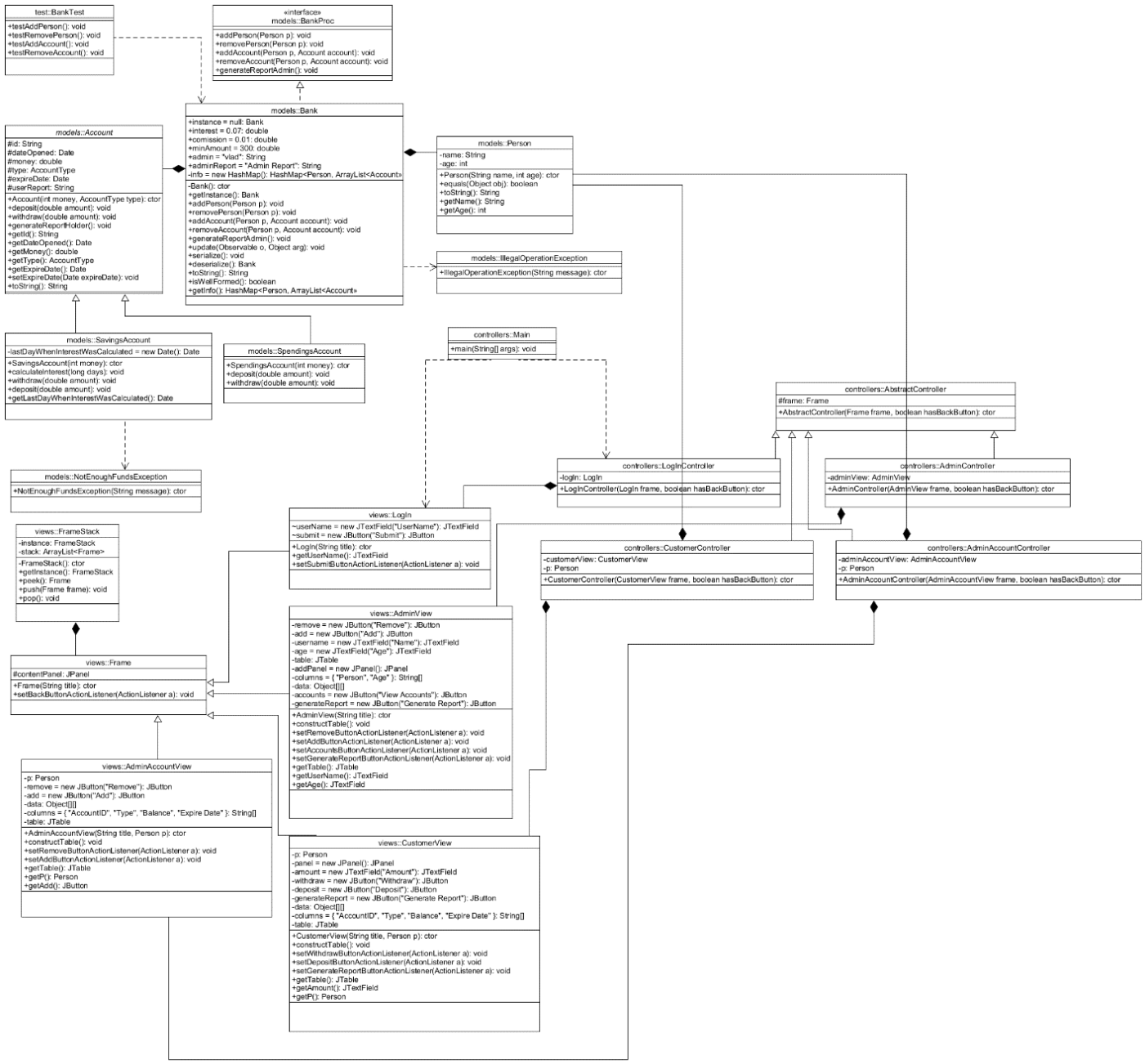
3.1. UML Diagrams

In the following we will present Use Case diagrams, Class Diagram, Sequence Diagram.

3.1.1. Use Case Diagrams (next page)



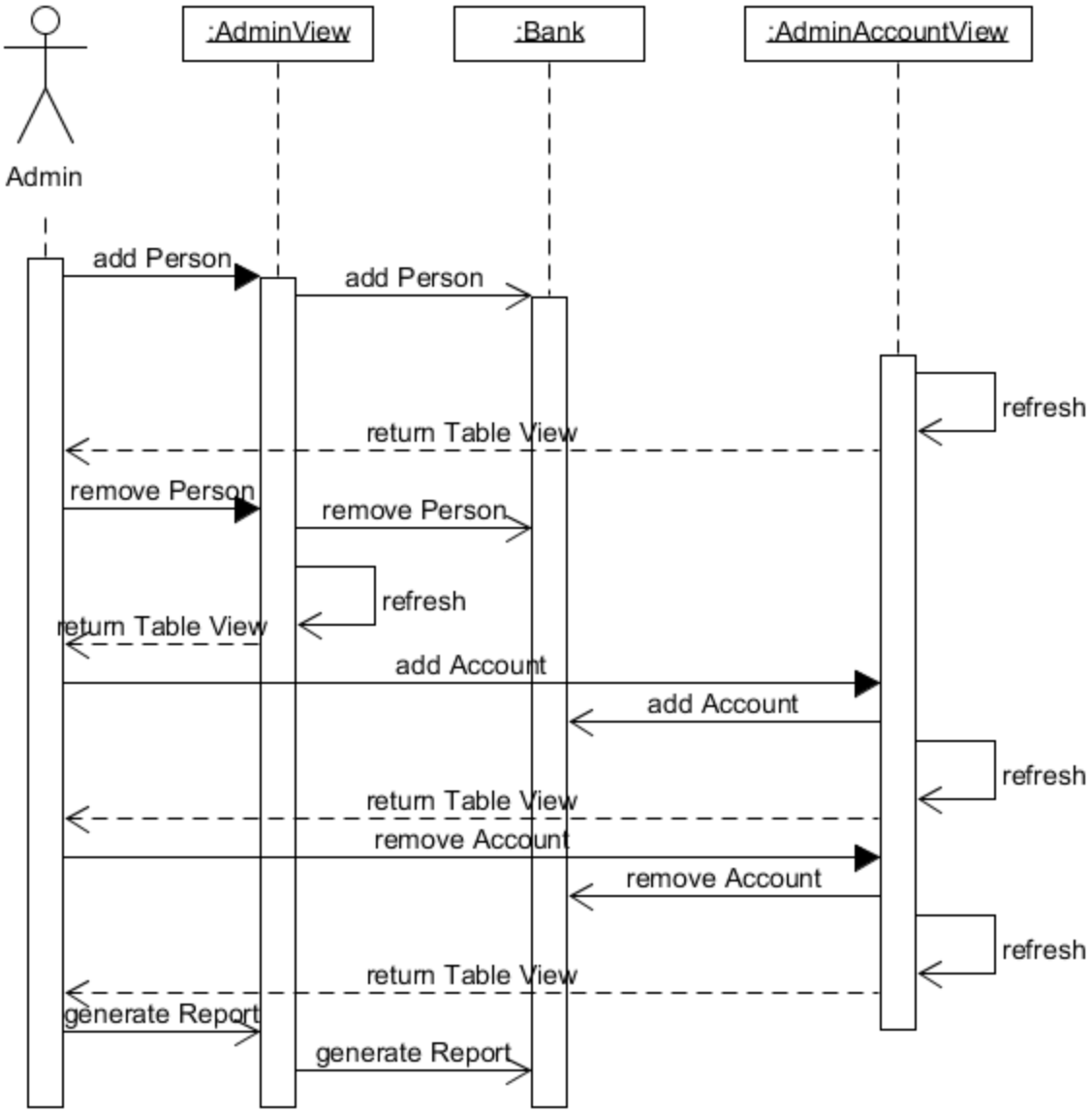
3.1.2.Class Diagram (next page)

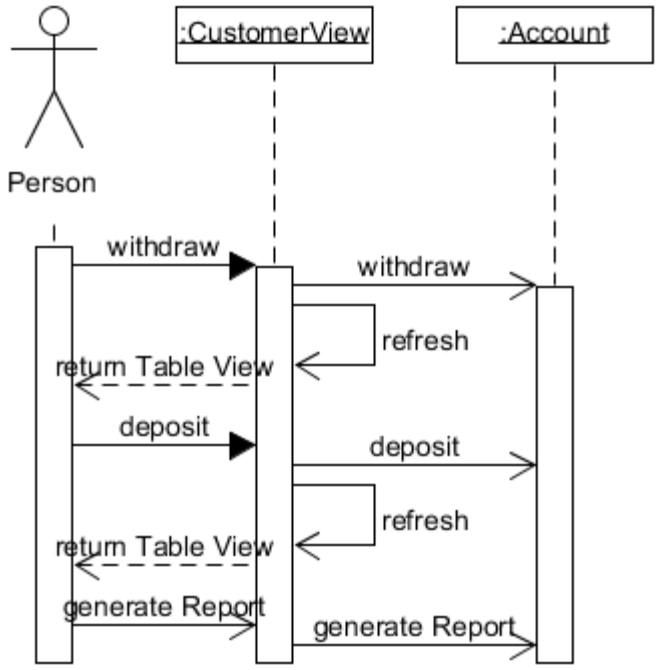


The class diagram shows the modelling of our problem into classes. There are 20 classes. One must know that not all fields and methods were represented on the diagram, only the ones that are important to our implementation (for example, getters/setters are not shown). Also,

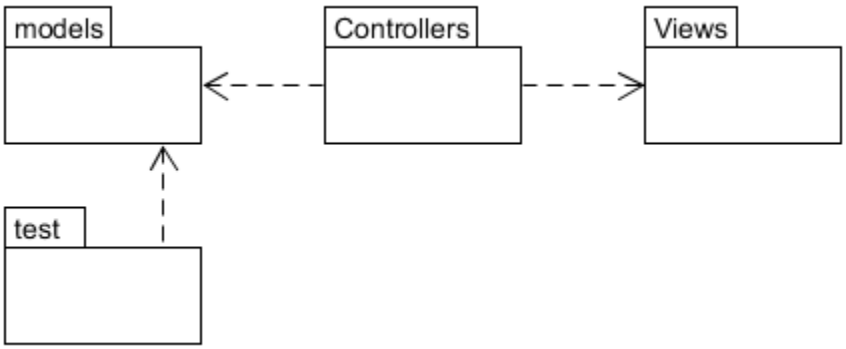
not all dependencies have been drawn, otherwise it would make the diagram unreadable.

3.1.3 **Sequence Diagrams**





3.1.3. Package diagram



3.2. Data Structures

Besides primitive data types like int, long, double and besides String, our implementation makes use of an implementation of Map called HashMap. The HashMap class uses a hashtable to implement the Map

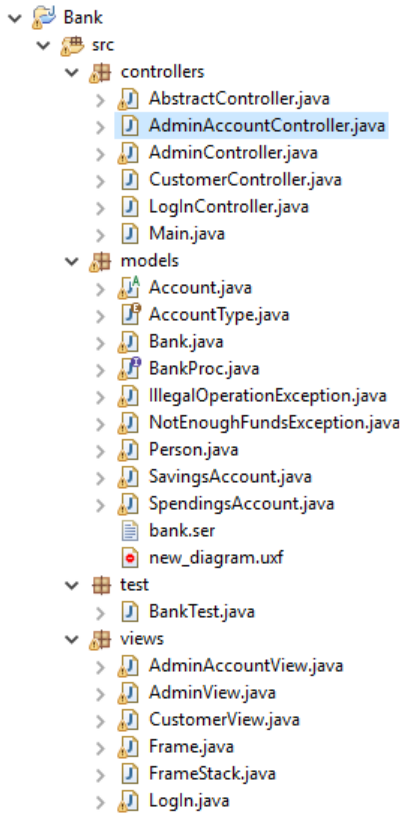
interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets. As a key, we will store persons, and as values, an ArrayList of Accounts. We need to make sure that no ArrayList is empty and that no multiple values of Accounts exist.

3.3. **Class Design**

The problem domain was decomposed into sub-problems, and for each sub-problem a solution was build. By assembling the solutions we obtain the general solution to our problem. The sub-problems were:

- how to correctly model a bank system?
- how will be implemented the user interface?
- how can we assure user interaction with our models?

For each of the above mentioned, we came up with a package that would support the features of the bank system. These packages follow the Model-View-Controller software architecture and they are as follows:



This concludes a quick introduction into the aspect of our class design.

3.4. Interfaces

The Interface BankProc defines the Bank related operations. It follow the design by contract approach, with preconditions and postconditions for each method. The BankProc interface is fully reproduced below:

```
public interface BankProc {

	/**
	 *
	 * @pre p != null
	 * @post size() == size()@pre + 1
	 * @post get(p) != null
	 */
	public void addPerson(Person p) throws IllegalOperationException;

	/**
	 * @pre p != null
	 * @pre containsKey(p)
	 * @post !containsKey(p)
	 * @post size() == size()@pre - 1;
	 *
	 */
	public void removePerson(Person p) throws IllegalOperationException;

	/**
	 * @pre p != null
	 * @pre a != null
	 * @pre get(p) != null
	 * @post get(p).size() == get(p).size()@pre + 1;
	 * @post !get(p).isEmpty()
	 *
	 */
	public void addAccount (Person p, Account account) throws
IllegalOperationException;

	/**
	 * @pre p != null
	 * @pre account != null
	 * @pre get(p) != null
	 * @post get(p).size() == get(p).size()@pre - 1
	 */
	public void removeAccount(Person p, Account account) throws
IllegalOperationException;

	public void generateReportAdmin();
}
```

3.5. Relations

In Object Oriented Programming, it is a good habit to design classes as loosely coupled as possible, in order to avoid error propagation. I tried to stick with this principle and, with few exceptions, I think I managed to fulfill this requirement. In this design, one can found the following relations: aggregation, dependency and innerness.

3.6. Packages

The design follows the Model-View-Controller (MVC) architecture. It has three packages: models, views, controllers. There is also an additional Test package, that contains a BankTest class, which is used for unit testing. The contents of the BankTest class is fully reproduced below:

```
public class BankTest {

	@Test
	public void testAddPerson() {
		Bank.deserialize();
		Person p = new Person("Test", 20);
		boolean flag = false;
		try {
			Bank.getInstance().addPerson(p);
		} catch (IllegalOperationException e) {
			e.printStackTrace();
			assertTrue(true);
			return;
		}
		for(Person pp: Bank.getInstance().getInfo().keySet())
			if(p.equals(pp)){
				flag = true;
				break;
			}
		assertTrue(flag);
	}

	@Test
	public void testRemovePerson() {
		Bank.deserialize();
		Person p = (Person)
Bank.getInstance().getInfo().keySet().toArray()[0];
		try {
			Bank.getInstance().removePerson(p);
		} catch (IllegalOperationException e) {
			e.printStackTrace();
			assertTrue(true);
			return;
```

```
			}
			assertFalse(Bank.getInstance().getInfo().containsKey(p));
		}

		@Test
		public void testAddAccount() {
			Bank.deserialize();
			Person p = (Person)
Bank.getInstance().getInfo().keySet().toArray()[0];
			Account a = null;
			try {
				a = new SavingsAccount(400);
			} catch (NotEnoughFundsException e1) {
				e1.printStackTrace();
				assertTrue(true);
				return;
			}
			try {
				Bank.getInstance().addAccount(p, a);
			} catch (IllegalOperationException e) {
				e.printStackTrace();
				assertTrue(true);
				return;
			}
			assertTrue(Bank.getInstance().getInfo().get(p).contains(a));
		}

		@Test
		public void testRemoveAccount() {
			Bank.deserialize();
			Person p = (Person)
Bank.getInstance().getInfo().keySet().toArray()[0];
			Account a = Bank.getInstance().getInfo().get(p).get(0);
			try {
				Bank.getInstance().removeAccount(p, a);
			} catch (IllegalOperationException e) {
				e.printStackTrace();
				assertTrue(true);
				return;
			}
			assertFalse(Bank.getInstance().getInfo().get(p).contains(a));
		}

}
```

3.7. Algorithms

In the following we will detail only the more intriguing algorithms used in our program.

The Bank class has a class invariant which is called isWellFormed(), which is tested before and after each operation in the Bank class. The isWellFormed method is asserted and tests the consistency of the Bank class. The isWellFormed method is fully reproduced below:

```
public boolean isWellFormed() {
            for (Person p : info.keySet()) {
                  if (info.get(p).isEmpty())
                        return false;
                  for (Account a : info.get(p)) {
                        if (a.getDateOpened().after(a.getExpireDate()))
                              return false;
                        if (a.getType() == AccountType.SAVINGSACCOUNT &&
a.getMoney() < 300)
                              return false;
                        if (a.getMoney() < 0)
                              return false;
                        for (Person pp : info.keySet()) {
                              for (Account aa : info.get(pp))
                                    if (aa.getId().equals(a.getId()) &&
!(aa.equals(a)))// check

                                    // for

                                    // uniqueness
                                          return false;
                        }
                  }
            }
```

When computing the interest, we will borrow the renowned formula from statistics. In our case the interest is computed per day. The caluclateInterest(long days) method is applied eaxh time the customer enters his account. The number of days past since the last updating is done using the formula:

```
long days = (System.currentTimeMillis() - ((SavingsAccount)
a).getLastDayWhenInterestWasCalculated().getTime())
                                    / (1000 * 60 * 60 * 24);
```

The calculateInterest method is fully reproduced below:

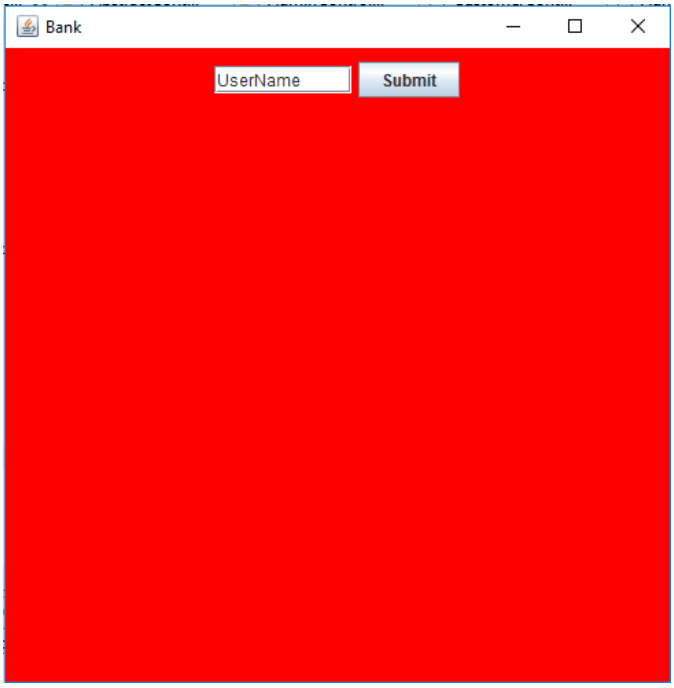
```
public void calculateInterest(long days) {
            money = money * Math.pow((1 + Bank.interest), days);
            lastDayWhenInterestWasCalculated = new Date();
            setChanged();
            notifyObservers(this);
            userReport += "\nInterest calculated. Now, the balance is: " +
money + ", on" + new Date();
      }
```

3.8. Patterns

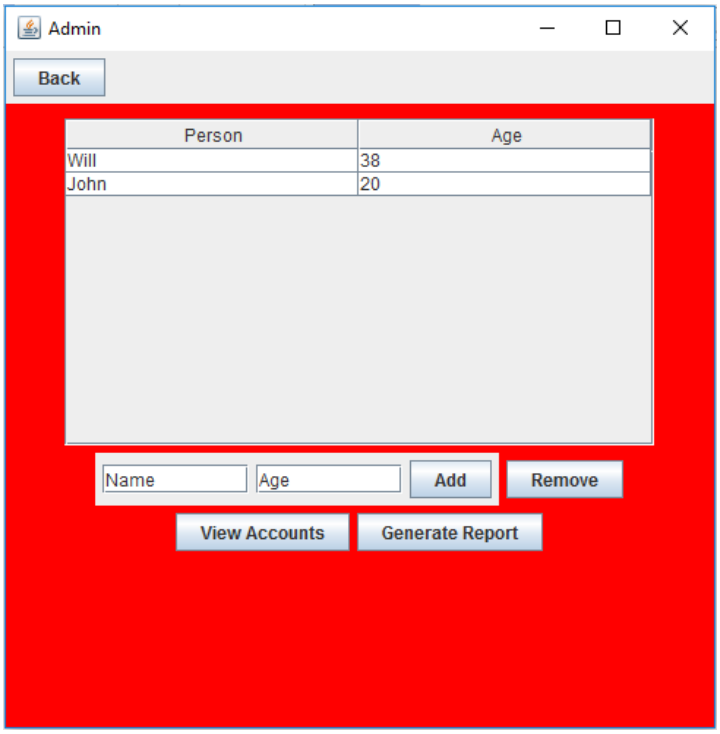
We have used the Singleton Pattern for implementing the FrameStack class and the Bank class. The Observer Pattern is also used: the Account extends the Observable class and the Bank implements the Observer interface. Whenever a new Account is created, it is added to it as an observer the Bank class. Also, whenever a method is performed on the Account class, the Observer is notified by:

```
setChanged();
notifyObservers(this);
```

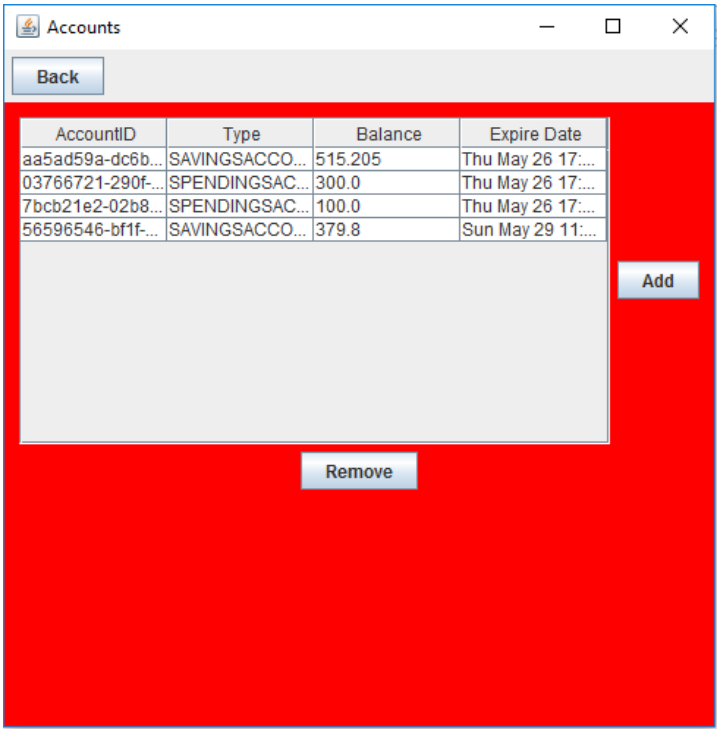
3.9. User Interface



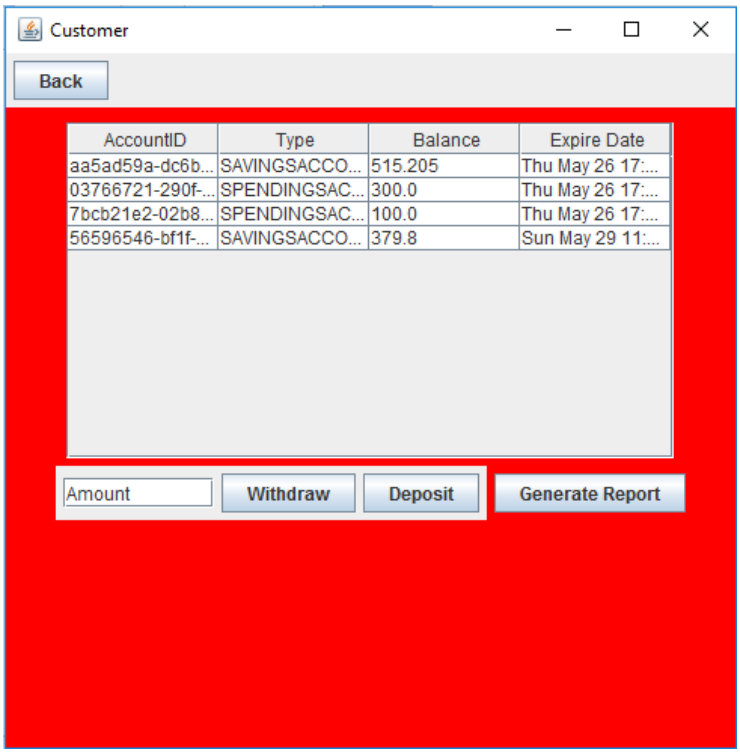
admin View:



View Account for admin:



user:



4. Implementation and Testing

The project was developed in Eclipse IDE using Java 8, on a Windows 10 platform. It should maintain its portability on any platform that has installed the SDK. It was heavily tested, but new bugs could be discovered in the near future. One of the main inconveniences are that the program is not protected against all kinds of illegal input data, only to some. This could be the subject of future development.

5. Results

The application is a user-friendly, helpful app that can perform bank systems simulations. It is a good tool in a hands of a potential corporate client that would like to simulate bank related applications

6. Conclusions

6.1. What I've Learned

TIME IS PRECIOUS! I had to solve complicated problems of time management which taught me good organization skills. I learned that a good model is always a key to a successful project and that a bad model could ruin your project in the end. I learned never to get stuck on one little bug, and if I do, ask for help, either from the TA or from colleagues or on different forums. I am looking forward to apply what I have learned in future projects.

6.2. Future Developments

The app could be made more user-proof. That is, if the user enters a String at a numerical field, an error dialog pops-up.

7. Bibliography

[1] http://www.uml.org/

[2] Barry Burd, *Java For Dummies*, 2014

[3] Kathy Sierra, Bert Bates, *Head First Java*, 2005

[4] Steven Gutz, Matthew Robinson, Pavel Vorobiev, *Up to Speed with Swing*, 1999

[5] Joshua Bloch, *Effective Java*, 2008

[6] http://www.stackoverflow.com/

[7] http://docs.oracle.com/

[8] http://www.coderanch.com/forums

[9] Derek Banas' Channel on YouTube
https://www.youtube.com/channel/UCwRXb5dUK4cvsHbx-rGzSgw

[10] Cave of Programming Channel on YouTube

https://www.youtube.com/user/caveofprogramming/playlists

[11] https://www.tutorilaspoint.com