**DOCUMENTATIE TEMA 4**

**Sistem de gestionare**

**a unei banci**

**Nume prenume Man Larisa Silvana**

**Grupa 30224**

**Profesor Laborator Assist Pop Claudia**

Contents

[1. Cerinte Functionale 3](#_Toc482948573)

[2. Obiective 3](#_Toc482948574)

[2.1. Obiectiv Principal: 3](#_Toc482948575)

[2.2. Obiective Secundare: 4](#_Toc482948576)

[3. Analiza Problemei 4](#_Toc482948577)

[4. Proiectare 6](#_Toc482948578)

[4.1. Structuri de date 6](#_Toc482948579)

[4.2. Diagrama de clase 10](#_Toc482948580)

[4.3. Algoritmi 12](#_Toc482948581)

[5. Implementare 12](#_Toc482948582)

[6. Testare 15](#_Toc482948583)

[7. Concluzii si Dezvoltari Ulterioare 16](#_Toc482948584)

[8. Bibliografie 16](#_Toc482948585)

# Cerinte Functionale

**Cerinta** proiectului este:

*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.*

# Obiective

## Obiectiv Principal:

Obiectivul acestei teme este acela de a crea o aplicație cu interfața grafica utilizator care se bazează pe procesarea conturilor unei bănci. Implementarea presupune adăugarea tuturor elementelor pentru a putea rezolva aceasta sarcina: adăugarea clienților, crearea si asocierea conturilor la clienții corespunzători, extragere/depunere numerar in cont. Trebuie sa facem diferența intre mai multe tipuri de conturi: Saving account si Spending account. Conturile păstrate in sistem, într-o structura de data tip HashMap, identificate unic printr-un ID, conturile pot si gestionate cu ușurință.

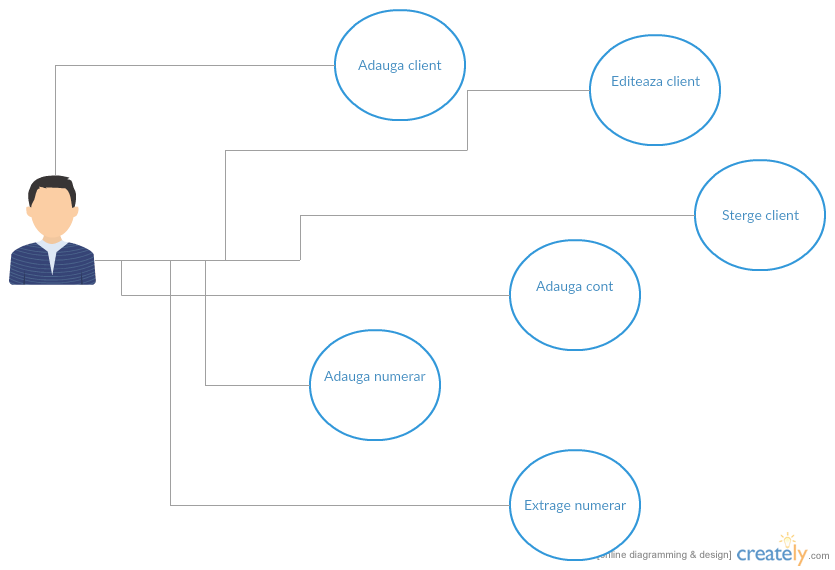
Pentru dezvoltarea acestei aplicații am folosit limbajul de programare Java, mediul de dezvoltare Eclipse si conceptele programării orientate pe obiect.

## Obiective Secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Alegerea operatiilor care se pot face asupra clientilor si al conturilor | 3 |
| Alegerea structurilor de date | Am ales folosirea HashMap-ului pentru o reprezentare mai usoara a structurii bancii. | 4 |
| Impartirea pe clase | Impartirea pe clase s-a facut dupa diagrama de clase data in cerinta | 4 |
| Dezvoltarea algoritmilor | Crearea algoritmilor pentru retragere si depunere numerar, de asemenea diferentiindu-se prin tipul de cont: *saving* sau *spending*. Fiecare dintre acestea vor gestiona bani din cont intr-un mod diferit | 4 |
| Implementarea solutiei | Crearea unei aplicatii care simuleaza modul de gestionare a conturilor din banca | 5 |
| Testare | Testarea programului cu JUnit | 6 |

# Analiza Problemei

Am creat diagrama de USE CASE pentru a fi mai clara structura aplicatiei si ce poate ea sa faca:



O banca pastreaza o evidenta a clientilor, care au in prezent sau au avut in trecut cel putin un cont asociat. Astfel, inainte de a crea un cont propriu zis, este nevoie de introducere a informatiilor despre persoana in baza de date a bancii: am ales sa include urmatoarele informatii despre o persoana: numele, codul numeric personal (CNP), adresa. Dupa inregistrarea persoanei este posibil sa cream un cont care va fi asociat unei persoane. O persoana poate avea mai multe conturi si de tipuri diferite.

In ceea ce priveste conturile, acestea se disting intre Saving account si Spending account. In acest program exemplu diferenta intre cele doua conturi este ca la saving account se poate extrage o singura data si la adaugarea de numerar se adauga o dobanda, in functie de suma depusa. Astfel, pt o suma mai mica de 100 RON, dobanda va fi de 0,1%, intre 100 RON si 500 RON, dobanda va fi 0,2% si peste 500 RON, 0,3 %. In cazul spending accountului, la fiecare retragere se va percepe un comision de 0,3%. Toate aceste taskuri trebuie implementate si puse la dispozitie sub forma unei interfete utilizator intuitiva, usoara de folosit atat pentru utilizatori specialisti, cat si pentru non-specialisti.

# Proiectare

## Structuri de date

Ca si structura de date folosita in proiect, am ales HashMap. Hashing este un bun exemplu de un compromis spațiu-timp. În cazul în care nu este nici o limitare de memorie, atunci am putea face orice căutare cu doar un acces la memorie pur și simplu folosind cheia ca o adresă de memorie, la fel ca în căutarea indexată. Acest ideal de multe ori nu poate fi atins deoarece cantitatea de memorie necesară este prohibitivă atunci când cheile sunt lungi. Pe de altă parte, în cazul în care nu ar exista nici o limitare de timp, atunci am putea obține răspunsul cu doar o cantitate minimă de memorie cu ajutorul unei metode de căutare secvențiale. Hashing oferă o modalitate de a folosi atât o cantitate rezonabilă de memorie și cât şi de timp care să ţină un echilibru între aceste două extreme. În particular, putem atinge orice echilibru alegem, pur și simplu prin ajustarea dimensiunii tabelei de hash, nu prin rescrierea codului sau alegerea de algoritmi diferiţi. Avantajul principal al acestei colecţii este că inserarea într-o astfel de structură este făcută în timp O(1) . Nu contează cât de mare este structura, va lua cam acelaşi timp, pentru inserarea unui element. Cum se poate acest lucru? Atunci când folosim perechi chei-valoare, cheile sunt convertite în cod hash, folosind un algoritm de hash. Apoi acest cod este redus la o structură internă, pentru a fi folosit ca un index. Pentru două elemente egale, codul va fi acelaşi iar pentru două elemente diferite codul este diferit. Dacă hashcode-ul este acelaşi pentru elemente diferite, discutăm de coliziune. Dacă sunt multe coliziuni, timpul de inserţie şi căutare, creşte şi se pierde astfel avantajul principal. Dacă există elemente multiple cu aceeaşi cheie, va trebui traversată o listă înlănţuită pentru a ajunge la valori. Mai jos avem un HashTable cu mai multe coliziuni:

**Creearea HashTable**

Există patru constructori pentru un HashTable, dintre care primii trei sunt:

*public Hashtable()*

*public Hashtable(int initialCapacity)*

*public Hashtable(int initialCapacity, float loadFactor)*

Ultimii doi constructori permit specificarea unei capacităţi iniţiale. Rata de mărire a capacităţii implicite, care poate fi specificată în cel de-al doilea constructor este de dublul capacităţii actuale +1. Se poate modifica prin ajustarea acelui procent. Al patrulea constructor iniţializează HashTable prin copierea unei colecţii în tabelul actual.

*public Hashtable(Map t)*

Adăugarea perechilor de chei-valoare Spre deosebire de clasele studiate anterior, trebuie să oferim o cheie dar şi o valoare:

*public Object put(Object key, Object value)*

într-un HashTable, cheile sau valorile nu pot fi null. Încercarea de introducere o cheie null în HashTable va produce eroarea NullPointerException. Introducerea unei perechi în care cheia se afla deja în colecţie, va avea ca efect înlocuirea valorii de la acea cheie cu valoarea nouă. Obiectul returnat va fi vechea valoare, sau dacă obiectul nu era în colecţie, obiectul returnat este null.

Afişarea unui HashTable se face natural pentru ca implementează metoda toString().

**Ştergerea unei perechi**

Se face utilizând funcţia remove():

*public Object remove(Object key).*

Dacă cheia key este prezentă în colecţie, atunci va fi eliminata perechea cheie-valoare specificată prin key şi valoarea este returnată. O altă metodă de a şterge elementele dintr-o colecţie de acest tip, este clear():

*public void clear()*

Această metodă va șterge toate perechile din HashTable.

**Mărimea unui HashTable**

Putem controla mărimea unui HashTable doar după crearea acestuia. Mai jos sunt câteva funcţii pentru controlul mărimii unui HashTable:

*public int size()*

*public boolean isEmpty()*

Aceste funcţii returnează mărimea colecţiei şi verifică dacă HashTable este gol sau nu. Există o funcţie de redimensionare a colecţiei, dar este protected:

*protected void rehash()*

Aceasta permite crearea unui nou şir intern mai mare, inserând toate valorile din şirul deja existent. Operaţii cu HashTable Returnarea obiectelor din HashTable Există câteva moduri de a returna date din HashTable. Cel mai simplu este apelând get:

*public Object get(Object key)*

dacă este găsită cheia dată ca parametru, atunci funcţia va returna valoarea corespunzătoare cheii. Dacă nu este găsită, atunci funcţia va returna null. Dacă vrem să aflăm cheile din colecţie putem folosi:

*public Enumeration keys()*

*public Set keySet()*

diferenţa constă în modul în care sunt returnate cheile. Prima metodă returnează ca un Enumeration. A doua metodă, returnează colecţia sub forma unui Set. Pe lângă chei, se pot returna şi valorile din HashTable:

*public Enumeration elements()*

*public Collection values()*

Metoda elements() returnează o mulţime de valori sub forma unei Enumeration. Metoda values() returnează aceleaşi date, dar sub forma unei Collection. Cea mai complexă metodă de returnare a elementelor dintr-o colecţie HashTable este entrySet()

*public Set entrySet()*

Aceasta returnează o colecţie de tip Set ce conţine perechile de chei-valori. Acest tip de data returnat este Map, şi vom discuta despre el în cele ce urmează. Iată două moduri de a accesa cheile unei astfel de colecţii:

*Enumeration enum = hash.keys();*

*while (enum.hasMoreElements())*

*{*

*String key = (String)enum.nextElement();*

*System.out.println(key + " : " + hash.get(key));*

*}*

Un alt mod, implică folosirea întregului obiect din colecţie:

*Set set = hash.entrySet();*

*Iterator it = set.iterator();*

*while (it.hasNext()) {*

*Map.Entry entry = (Map.Entry)it.next();*

*System.out.println(entry.getKey() + " : " + entry.getValue());*

*}*

Căutarea elementelor Clasa Hashtable conţine trei metode, care ne permit căutarea unei chei sau a unei valori în colecţie. Cea mai simplă rămâne metoda get() discutată mai sus.

*public boolean containsKey(Object key)*

Această funcţie verifică existenţa unei chei în colecţie. Alte două metode vor verifica existenţa unei valori în colecţie:

*public boolean contains(Object value)*

*public boolean containsValue(Object value)*

Ambele realizează acelaşi lucru, însă pentru că HashTable implementează şi interfaţa Map, avem această duplicitate. Este de recomandat să folosim aceste două funcţii cât mai puţin posibil, deoarece se parcurge întreaga colecţie, algoritmul având cel un ordin O(n).

**Verificarea egalităţii între HashTable**

Aceasta se poate realiza folosind funcţia equals():

*public boolean equals(Object o)*

Egalitatea este definită de interfaţa Map, iar nu la nivelul HashTable. În consecinţă, HashTable poate fi egal cu orice alt Map şi nu doar un alt HashTable. Regula de bază este că, două HashMap-uri sunt egale atunci când au aceleaşi perechi cheie-valoare. Ordinea nu contează. HashTable este imutabil. Dacă un HashTable este iniţializat cu un set de elemente, atunci pentru redimensionare va trebui să iniţializăm un nou obiect. Pentru a obţine un Map, ce nu este imutabil avem următoarea funcţie unmodifiableMap:

*Hashtable h = new Hashtable(); // se va umple hashtable*

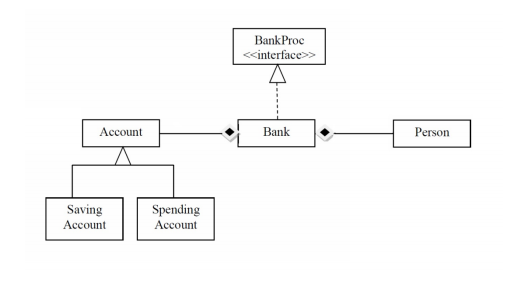
*Map m = Collections.unmodifiableMap(h);*

Un concept nou folosit este cel de „observer”. Sablonul observer este utilizat atunci cand se doreste ca unul sau mai multe obiecte sa fie notificate cu privire la schimbarea starii unui alt obiect. Sablonul implementeaza un mecanism de inregistrare a obiectelor interesate de schimbarea starii obiectului tinta si un mecanism de schimbare a starii obiectului tinta. In momentul schimbarii starii obiectului se poate transmite si o informatie suplimentara catre observatori prin intermediul unui obiect event.

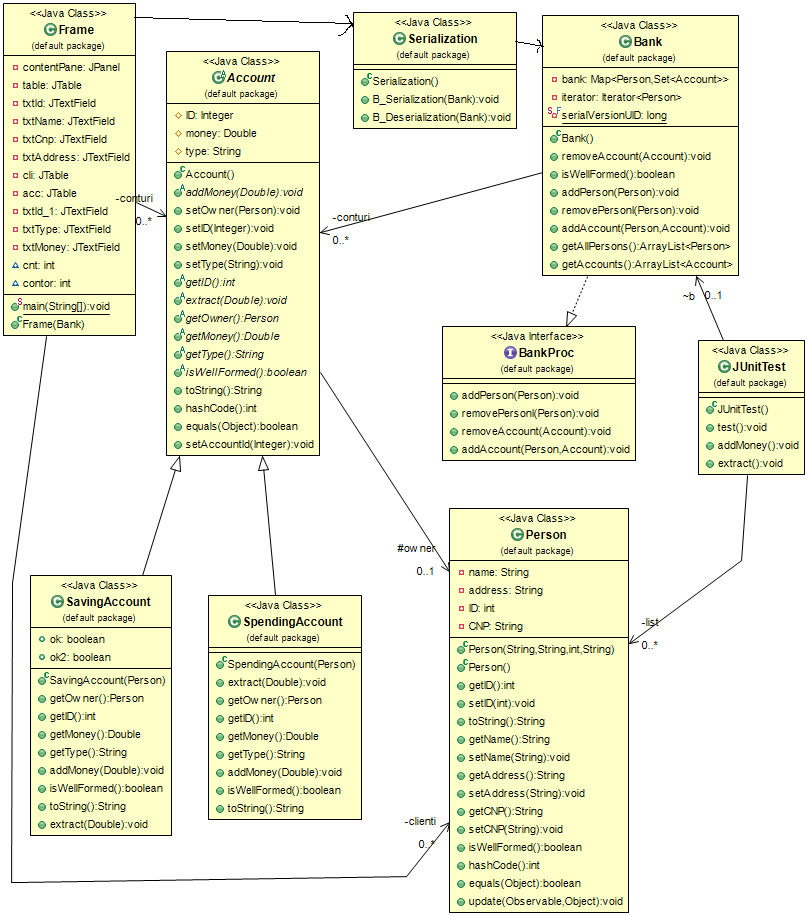
Java standard implementeaza sablonul Observer prin intermediul claselor Observer si Observable localizate in pachetul java.util. Primul exemplu de mai jos prezinta o implementare proprie a acestui sablon. Al doilea exemplu implementare utilizeaza clasele Observer si Observable din pachetul java.

## Diagrama de clase

Pornind de la diagrama de clasa data in cerinta:



Am ajuns la urmatoarea diagrama de clase:



## Algoritmi

Operatiile implementate pe banca sunt de adaugare clienti, stergere clienti, adaugare cont, stergere cont, iar pentru fiecare cont in parte, in functie de tipul lui (saving sau spending), se pot face operatiile de extragere si depunere numerar. In cazul in care contul este de tip „saving” , se vor putea efectua o singura data operatiile de depunere si extragere, iar la depunere se va acorda o dobanda in functie de suma depusa. Astfel, dobanda poate fi de 0.1% , 0.2% sau 0.3%. Daca contul este de tip „spending”, la fiecare retragere se va prelua un comision de 0.3%.

# Implementare

Structura acestui proiect a fost facuta dupa diagrama de clase din carinta. Astfel avem clasa Person, Account, SavingAccount, SpendingAccount, Bank, BankProc, Frame, JUnitTEst si Serialization.

**Clasa Person** implemeanteaza clasele Serializable si Observer. Persoana va di observer-ul care va stii cand se face o modificare in contul lui. El este caracterizat de un ID unic, numit si nr de Cont, un nume, o adresa si un CNP. Aceasta clasa prezinta gettere, settere, metoda indispensabila : hashCode(), care se face dupa ID, metoda equals care face verificari, metoda update(), folosita pentru Observable, si metoda IsWellFormed() care verifica daca datele sunt corecte.

**Clasa abstracta Account** extinde Observable, adica ea influeteaza informatia transmisă persoanei. Un Account este caracterizat de owner-ul lui, persoaa care detine acel cont, de un ID unic, o suma de bani si un tip. Un account poate fi de doua feluri: saving account sau spending account. Despre acestea vom detalia in urmatoarele randuri.

Aceasta clasa contine gettere, settere, metodele hashCode(), equals() si antetul metodelor pentru operatiile ce se pot face pe un cont.

**Clasa SavingAccount** extinde clasa Account, deci vom avea aceiași parametri. Pe lângă acestea, gettere, settere, o metoda isWellFormed(), care verifica corectitudinea datelor si operațiile pe cont, respectiv adăugare si extragere numerar. In acest caz, la o depunere in cont se oferă o dobânda de 0.1% daca suma introdusa e mai mica decât 100 RON, 0.2% daca e intre 100 si 500 RON, iar pentru sume mai mari, dobânda este de 0.3%. o alta particularitate a aceste clase este faptul ca operațiile se pot executa o sigura data pe ea.

**Clasa SpendingAccount** extinde clasa Account, deci vom avea aceiași parametri. Pe lângă acestea, gettere, settere, o metoda isWellFormed(), care verifica corectitudinea datelor si operațiile pe cont, respectiv adăugare si extragere numerar. In acest caz la retragerea din cont se percepe un comision de 0.3%, indiferent de suma aleasa.

**Interfata BancProc** contine antetele operatiilor care se aplica pe banca, adica adaugarea, stergerea de persoane din banca si adaugarea, stergerea de conturi. Totodata, aici gasim stlul design-by-contract, adica prezenta preconditiilor, postconditiilor si a claselor invariant. Inainte de fiecare metoda am pus preconditiile, are trebuie sa fie adevarate cand se apeleaza metoda, postconditiile, care trebuie sa fie adevarate dupa ce metoda s-a executat cu succses si parametri metodei.

**Clasa Bank** implementeaza Serializable si clasa BankProc. Banca este reprezentata sub forma unu Map cu persoane, carora li se atribuie conturi : **Map<Person,Set<Account>>.** Aceasta clasa implementeaza metodele regasite in BankProc, si nu numai. Am adaugat diferite metode de care avem nevoiecum ar fi: isWellFormed() – verifica daca avem clienti in banca, si daca putem apela operatiile, un get pentru clienti, unul pentru conturi, fiecare din ele returnand un ArrayList cu obiectele respective.

*Metoda addPerson( Person p ),* care adauga o persoana in banca, verifica cu ajutorul asserturilor daca persoana introdusa este valida, daca nu cumva ea exista deja in banca, si daca ttul este bine, o adauga, marind dimensiunea bancii si verificand acest lucru.

*Metoda removePerson( Person p )* sterge un client din baza de date a bancii, verificand daca ea chiar exista sau nu.

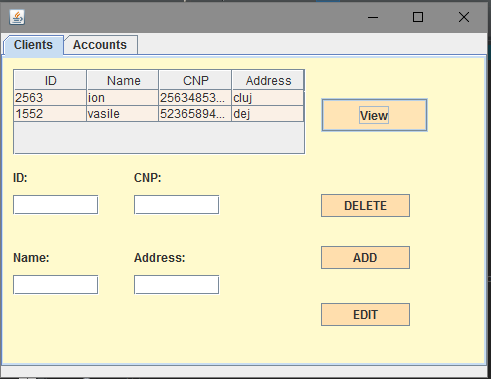
*Metoda addAccount( Person person, Account account)* adauga un cont unei persoane, verificand daca exista persoana respectiva in cont, daca datele sunt valide apoi daca persoana mai are vreun cont creat si se adauga contul. Tot aici este adaugat observerul, adica persoana care aprimit contul in banca. In momentul in care se face vreo modificare in contul sau, el va fi anuntat.

*Metoda removeAccount( Account a )* parcurge lista clientilor si cand gaseste contul pe care vrem sa il stergem, face acest lucru si actualizeaza banca.

**Clasa Serialization** contine doua metode pentru a realiza serializarea bancii, adica pastrarea datelor: B\_Serialization, metoda care scrie si B\_Deserialization, metoda care citeste Obiectele. Se creeaza un fisier „bank.ser” , unde se retin datele.

**Clasa JUnitTest** este cea care testeaza daca operatiile functioneaza si va fi desrisa in capitolul 6.

**Clasa Frame** contine interfata grafica GUI a aplicatiei si main-ul programului, deci aceasta clasa este creierul aplicatiei bancare. Pentru crearea interfetei grafice am folosit tool-ul pus la dispozitie de Eclipse, WindowBuilder. Astfel, am creat o fereastra cu doua tab-uri. Un tab este pentru operatiile efectuate pe client si al doilea pentru operatiile efectuate pe conturi.

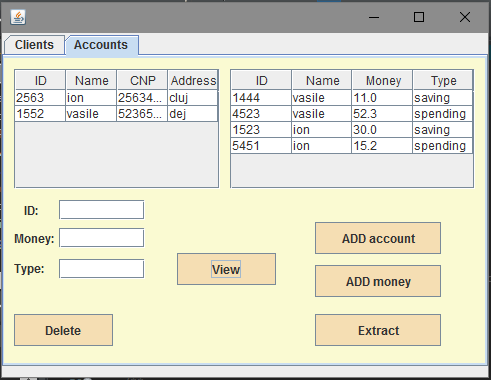
Primul tab contine o lista cu clientii din banca si datele lor si patru butoane. 

Primul este View, pentru vizualizarea in tabel a clientilor, al doilea – Delete, pentru stergerea unei persoane din lista,cu tot cu account-urile sale. Stergerea se face introducand ID-ul persoane care dorim sa fie stearsa in casuta respectiva. Butonul Add adauga o persoana in banca, daca datele de intrare ale persoanei sunt corecte, iar butonul Edit, editeaza datele unei persoane, in afara de ID. Dupa ID se face cautarea persoane dorite.

JTextField-urile sunt pentru a introduce datele de care avem nevoie in efectuarea operatiilor.

Al doilea tab, denumit Accounts, conține doua tabele si șase butoane. Primul tabel afiseaza persoanele din banca, iar al doilea toate conturile din banca, impreuna cu posesorii lor.

Aici se efectueaza operatiile pe conturi. Astfel, primul buton este View, care afiseaza in tabelul pentru conturi, noile modificari, butonul Delete sterge un account, introducandu-se ID-ul lui in textField-ul reprezentativ. Butonul Add account adauga un cont personei a carui ID este specificat in JtextField-ul creat in acest sens. La introducere se va specifica suma pe care o vom regasi in cont si tipul contului – saving sau spending. butonul Add money adauga o suma dorita de noi in contul al carui ID se specifica, in functie de tipul lui, iar butonul Extract, are rolul de a extrage din contul specificat suma dorita.



# Testare

In clasa JUnitTest, am verificat daca se fac corect adaugarea de persoane, conturi, depunerea de bani si retragerea de numerar. Pentru fiecare operatie am creat o metoda, de exemplu pentru a verifica depunerea, am creat metoda addMoney(), in care am introdus o persoana cu un cont in banca, am adaugat o suma si cu functia assertEquals, am verificat daca datele dorite sunt aceleasi cu rezultatul.

***@Test***

**public** **void** **addMoney**() **throws** **Exception** {

**Person** **p1** = **new** Person("ion", "cluj", 1256, "2563485326125");

b.addPerson(p1);

**Account** **a2** = **new** SpendingAccount(p1);

a2.setMoney(30.0);

a2.setType("saving");

a2.addMoney(10.0);

*assertEquals*(**Double**.*valueOf*(40.0), a2.getMoney());

}

# Concluzii si Dezvoltari Ulterioare

Rezultatele obtinute in urma simualarii bancii, in conformitate cu datele furnizate de utilizator pot fi observate in interfata grafica si verificate prin urmarirea atenta a operatiilor efectuate asupra conturilor. De asemenea, am inclus un tabel in cadrul caruia se poate observa in timp real starea conturilor incluse in banca. Fiecare account poate fi deci updatat, iar rezultatul se vede in tabel.

Ca o concluzie, acest proiect m-a ajutat sa imi consolidez cunostintele de programare orientata pe obiect dobandite in primul semestru si sa imi organizez munca in baza paradigmelor POO. Totodata, felul in care am structurat proiectul si am creat clasele si metodele m-a ajutat sa eficientizez codul din punct de vedere al lungimii si al usururintei intelegerii acestuia, lucruri vitale in cazul programelor complexe care necesita mai multe clase sau care presupun un numar mai mare de programatori.

In cadrul dezvoltarilor ulterioare se pot aminti urmatoarele: imbunatatirea interfetei grafice (prin folosirea de model view controler), implementarea a diverse metode ce ajuta la securizarea conturilor. Astfel, aplicatia ar avea un impact mult mai potrivit asupra realitatii, deoarece proiectul actual nu asigura o prea mare siguranta clientilor, rezumandu-se strict la partea teoretica a ideii temei. Bineinteles, proiectul e simplist, iar cum o banca in zilele noastre ofera un numar tot mai mare de servicii, este ideal ca anumite metode generale precum: transfer credit, investitii bani. Nu in ultimul rand, proiectul mai poate fi imbunatatit in ceea ce priveste cautarea anumitor informatii, in ideea reducerii timpilor.

# Bibliografie

1. <http://cursuri.cs.pub.ro/~poo/wiki/index.php/Colectii>
2. <https://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html>
3. <http://www.tutorialspoint.com/java/java_serialization.htm>
4. <http://javarevisited.blogspot.ro/2011/02/how-hashmap-works-in-java.html>
5. <http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
6. . <http://javarevisited.blogspot.ro/2012/01/what-is-assertion-in-java-java.html>
7. <http://stackoverflow.com/>