BankManagement

**Student:** Maxim Bogdan-Gheorghe

Cuprins

[1. Obiectivul temei](#_Toc377483546) 3

[2. Analiza problemei, modelare, șcenarii, cazuri de utilizare](#_Toc377483547) 4

[3. Proiectare](#_Toc377483548) 5

[4. Implementare și testare](#_Toc377483554) 18

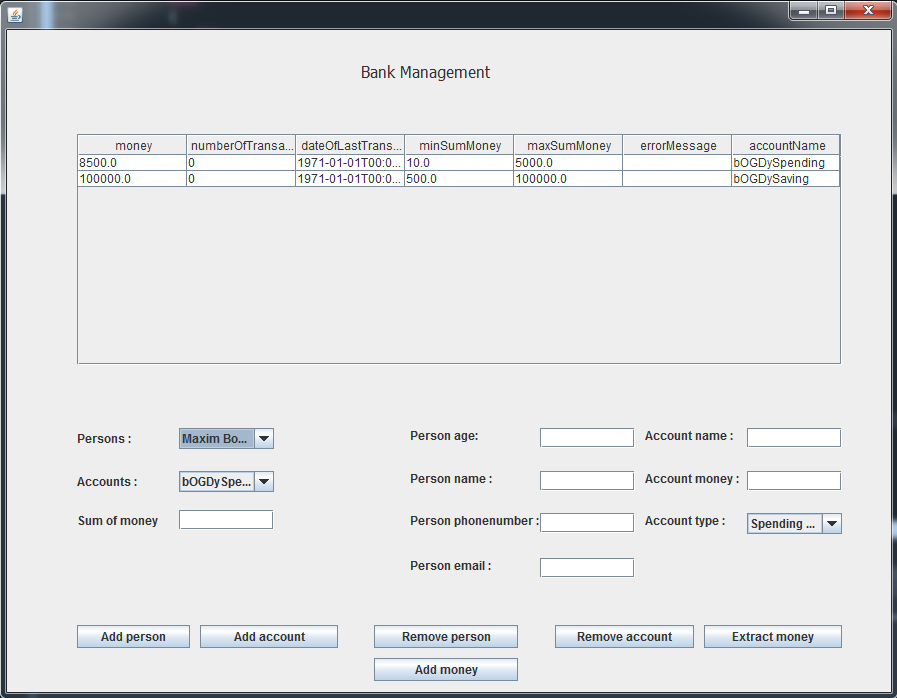
[5. Rezultate](#_Toc377483558) 19

[6. Concluzii și dezvoltării ulterioare](#_Toc377483581) 19

7. [Bibliografie 20](#_Toc377483584)

# Obiectivul temei

Obiectivul temei este proiectarea, implementarea și realizarea unei aplicații menite pentru a realiza gestiunea conturilor într-o bancă. Aplicația implementată poate fi folosită într-o bancă reală, unde accesul la conturi se face de către o persoană fizică angajată ca operator care va facilita accesul la cont unui utilizator în momentul în care acesta va dori extragerea unei anumite sume dintr-un cont deținut.



# 2.Analiza problemei,modelare,șcenarii,cazuri de utilizare

Primul pas efectuat în analiza problemei curente este studierea și analizarea diagramei propuse pentru realizare. Acest pas înlocuiește analizarea și identificarea substantivelor din proiectele propuse precedent. Astfel observăm clasele și interfețele care urmează a fi implementate : Person, Account, SavingAccount, SpendingAccount, BankProc și Bank. Pe lângă clasa interfeței grafice adaugate, avem și clasa Observer care implementează o versiune simplificată a pattern-ului Observer pentru a notifica banca în realizarea efectuării modificărilor necesare lunare asupra banilor din conturi(plata automată de taxe, adăugarea dobânzilor, etc).

Începem a identifica persoana. Ea are un nume, un număr de telefon, o adresă de email și o vârstă care trebuie să fie neapărat peste 18 ani(pentru a avea un cont valid în bancă).

Clasa Account este o clasă abstractă implementată de clasele concrete SavingAccount și SpendingAccount(care poate fi extinsă ușor și către alte clase care implementează alte tipuri de conturi, în funcție de banca ce folosește softul dat). Ea conține pe lângă alte date necesare implementării, suma depusă în cont și suma minima și maximă care poate fi stocată.

Clasa SavingAccount implementează clasa Account și se diferențiază de clasa SpendingAccount prin faptul că acest cont este un cont de salvare de bani. Asta înseamnă că suma extrasă minimă este mare(5000.00 lei) iar extragerea poate fi făcută exact odată pe lună. De asemenea taxele extrase din acest cont sunt mult mai mici(mai exact cu 50% mai puțin). În contrast, SpendingAccount poate efectua tranzacții mult mai mici(de la 10.00 lei până la cel mult 5000

00 lei) și poate efectua un număr maxim de trei tranzacții pe zi. Acest tip de cont este potrivit pentru extrageri de bani zilnice. De asemenea, taxele plătite sunt integrale.

Următoarea componentă importantă este interfața BankProc. În ea sunt definite operațiile principale efectuate de către operator : Adaugare/Ștergere/Listare persoane și respectiv conturi specifice. Clasa Bancă implementează această interfață și conține o metodă specifică pentru a actualiza toate sumele depuse în contă în data de 1 a lunii. Identificăm astfel următoarele scenarii:

-Adăugarea unei persoane

-Ștergerea unei persoane

-Adăugarea unui SavingAccount

-Ștergerea unui SavingAccount

-Adăugarea unui SpendingAccount

-Ștergerea unui SpendingAccount

-Listarea clienților

-Listarea conturilor și a diferitelor informațiilor pe cont

-Extragerea unei anumite sume de bani din cont în funcție de contul selectat

-Adăugarea unei anumite sume da bani într-un cont.

Cazurile de utilizare vizează băncile și diferitele societăți comerciale care doresc păstrarea unui depozit și diferite operații de sume pe acestea.

# 3.Proiectare

Pentru prezentarea proiectării aplicației, continuăm prin a prezenta trei tehnici de programare utilizate :

Design by contract :

- reprezinta un "contract" care specifică restricțiile la care trebuie să se supună datele de intrare ale unei metode, valorile posibile de ieșire si stările în care se poate afla programul - aceste restrictii sunt date sub forma unor:

a) precondiții: reprezintă obligațiile pe care datele de intrare ale unei metode trebuie sa le respecte pentru ca metoda sa functioneze corect

b) postconditii: reprezinta garantiile pe care datele de iesire ale unei metode le ofera

c) invarianti: reprezinta conditii impuse starilor in care programul se poate afla la un moment dat

Cel care a implementat clasa su metoda restectiva ii spune utilizatorului ce date sunt considerate valide ca date de intrare. Aceasta il scuteste sa foloseasca teste de validare a datelor, care in unele cazuri doar ar incetini algoritmul, ele fiind redundante cu teste care deja sunt facute pentru validarea datelor, de exemplu imediat ce au fost introduse de catre utilizator.

În acelasi timp utilizatorul stie din contract la ce date posibile de iesire sau stari intermediare sa se astepte si atunci poate sa-si optimizeze propriul cod in functie de acestea

Contractul unei clase sau al unei metode se specifica literar, printr-un text, inclus in program ca un comentariu sau scris in documentatia aferenta - in faza de dezvoltare a unui program se pot folosi instructiuni care sa testeze indeplinirea contractului, instructiuni care sa fie scoase pe urma (manual sau automat) din codul final. O asertiune genereaza o exceptie speciala cu textul dat dupa ":", in cazul in care conditia ei este falsa. Deoarece asertiunile au fost introduse mai tarziu in limbajul Java trebuie specificat la optiunile de compilare: -source 1.4 - implicit asertiunile nu fac nimic (sunt dezactivate), ca si cum codul este pregatit pentru livrare. Daca dorim sa activam asertiunile trebuie sa specificam la optiuni: -enableassertions (sau -ea) .

Serializare

Serializarea este o metoda prin care se pot salva, într-o maniera unitara, datele împreuna cu signatura unui obiect. Folosind aceasta operatie se poate salva într-un fisier, ca sir de octeti, o instanta a unei clase, în orice moment al executiei. De asemenea, obiectul poate fi restaurat din fisierul în care a fost salvat în urma unei operatii de serializare.

Utilitatea serializarii constă în următoarele aspecte:  
Asigură un mecanism *simplu de utilizat* pentru salvarea și restaurarea  
a datelor.  
Permite *persistența obiectelor*, ceea ce înseamna că durata de viața a unui obiect nu este determinată de execuția unui program în care acesta este definit - obiectul poate exista și între apelurile programelor care îl folosesc. Acest lucru se realizează prin serializarea obiectului și scrierea lui pe disc înainte de terminarea unui program, apoi, la relansarea programului, obiectul va fi citit de pe disc ¸si starea lui refacută. Acest tip de persistență a obiectelor se numește *persistența ușoară*, întrucât ea trebuie efectuată explicit de către programator ¸si nu este realizată automat de către sistem.

*Compensarea diferențelor între sisteme de operare* - transmiterea unor  
informații între platforme de lucru diferite se realizează unitar, independent de formatul de reprezentare a datelor, ordinea octeților sau alte detalii specifice sistemelor repective.

*Transmiterea datelor în rețea* - Aplicațiile ce rulează în rețea pot comunica între ele folosind fluxuri pe care sunt trimise, respectiv recepționate,obiecte serializate

*RMI (Remote Method Invocation)* - este o modalitate prin care metodele  
unor obiecte de pe o altă mașină pot fi apelate ca și cum acestea ar exista local pe mașina pe care rulează aplicația. Atunci când este trimis un mesaj către un obiect ”remote”

(de pe altă mașină), serializarea este utilizată pentru transportul argumentelor prin rețea și pentru returnarea valorilor.

*Java Beans* - sunt componente reutilizabile, de sine stătătoare ce pot fi utilizate în medii vizuale de dezvoltare a aplicațiilor. Orice componentă Bean are o stare definită de valorile implicite ale proprietăților sale, stare care este specificată în etapa de design a aplicației. Mediile  
vizuale folosesc mecanismul serializării pentru asigurarea persistenței componentelor Bean.

Un aspect important al serializării este că nu salvează doar imaginea unui obiect ci și toate referințele la alte obiecte pe care acesta le conține. Acesta este un proces recusiv de salvare a datelor, întrucât celelalte obiectele referite de obiectul care se serializează pot referi la rândul lor alte obiecte, și așa mai departe. Așadar referințele care construiesc starea unui obiect formează o întreagă rețea, ceea ce înseamnă că un algoritm general de salvare a stării unui obiect nu este tocmai facil. In cazul în care starea unui obiect este formată doar din valori ale unor variabile de tip primitiv, atunci salvarea informațiilor încapsulate în acel obiect se poate face și prin salvarea pe rând a datelor, folosind clasa DataOutputStream, pentru ca apoi să fie restaurate prin metode ale clasei DataInputStream, dar, așa cum am vazut, o asemenea abordare nu este în general suficientă, deoarece pot apărea probleme cum ar fi: variabilele membre ale obiectului pot fi instanțe ale altor obiecte, unele câmpuri pot face referință la același obiect, etc.

Salvarea datelor înapsulate într-un obiect se poate face si prin salvarea pe rând a datelor, folosind clasa DataOutputStream, pentru ca apoi sa fie restaurate prin metode ale clasei DataInputStream, dar o asemenea abordare nu este în general suficienta, deoarece pot aparea probleme cum ar fi :

* datele obiectului pot fi instante ale altor obiecte
* în unele cazuri, este necesara si salvarea tipului datei
* unele câmpuri fac referinta la acelasi obiect

Asadar, prin serializare sunt surprinse atât datele, signatura clasei (numele metodelor si definitia lor - nu si implementarea) precum si starea obiectului.

Pentru a putea fi serializat un obiect trebuie sa fie instanta a unei clase care implementeaza una din interfetele :

- java.io.Serializable sau

- java.io.Externalizable (care extinde clasa Serializable)

Interfata Serialize nu are nici o metoda, ea da doar posibilitatea de a specifica faptul ca se doreste ca o anumita clasa sa poata fi serializata. Declaratia unei astfel de clase ar fi :

class ClasaSerializabila implements Serializable {...}

In urma serializarii obiectele sunt pot fi salvatr într-un fisier, în acelasi fisier putând fi salvate si mai multe obiecte. Operatiile de intrare iesire la nivelul obicetelor se realizeaza prin intermediul unor fluxuri de obiecte, implementate de clasele ObjectInputStream si ObjectOutputStream.

Salvarea unui obiect într-un fisier se realizeaza astfel :

MyObject o = new MyObject();

FileOutputStream fout = new FileOutputStream(“fisier”);

ObjectOutputStream sout = new ObjectOutputStream(fout);

sout.writeObject(o);

Restaurarea unui obiect salvat într-un fisier se face într-o maniera asemanatoare:

FileInputStream fin = new FileInputStream(“fisier”);

ObjectInputStream sin = new ObjectInputStream(fin);

o = (MyObject) sin.readObject();

Pe lânga metodele de scriere/citire a obiectelor cele doua clase pun la dispozitie si metode pentru scrierea tipurilor de date primare, astfel încât apeluri ca cele de mai jos sunt permise :

FileOutputStream ostream = new FileOutputStream("t.tmp");

ObjectOutputStream p = new ObjectOutputStream(ostream);

p.writeInt(12345);

p.writeObject("Today");

p.writeObject(new Date());

p.flush();

ostream.close();

FileInputStream istream = new FileInputStream("t.tmp");

ObjectInputStream p = new ObjectInputStream(istream);

int i = p.readInt();

String today = (String)p.readObject();

Date date = (Date)p.readObject();

istream.close();

ObjectInputStream si ObjectOutputStream implementeaza indirect interfetele DataInput, respectiv DataOutput, interfețe ce declara metode atât pentru scrierea/citirea datelor primitive, cât si pentru scrierea/citirea obiectelor. Pentru transferul obiectelor sunt folosite metodele:

final void writeObject( java.lang.Object obj )

throws java.io.IOException

final java.lang.Object readObject( )

throws java.io.OptionalDataException, java.lang.ClassNotFoundException, java.io.IOException

Acestea apeleaza la rândul lor metodele implicte de transfer defaultWriteObject si defaultReadObject (având aceleasi signaturi ca mai sus).

Personalizarea serializarii se realizeaza prin supradefinirea (într-o clasa serializabila!) a metodelor writeObject si readObject, modificând astfel actiunea lor implicita.  
Metoda writeObject controleaza ce date sunt salvate si este uzual folosita pentru a adauga informatii suplimentare la cele scrise implicit de metoda defaultWriteObject.  
Metoda readObject controleaza modul în care sunt restaurate obiectele, citind informatiile salvate si, eventual, modifcând starea obiectelor citite astfel încât ele sa corespunda anumitor cerinte.  
Aceste metode trebuie obligatoriu sa aiba urmatorul format:  
private void writeObject(ObjectOutputStream stream)  
throws IOException  
private void readObject(ObjectInputStream stream)  
throws IOException, ClassNotFoundException  
De asemenea, uzual, primul lucru pe care trebuie sa îl faca aceste metode este apelul la metodele standard de serializare a obiectelor defaultWriteObject, respectiv defaultReadObject si abia apoi sa execute diverse operatiuni suplimentare. Forma lor generala este:  
private void writeObject(ObjectOutputStream s)  
throws IOException {  
s.defaultWriteObject();  
// personalizarea serializarii  
}  
private void readObject(ObjectInputStream s)  
throws IOException,ClassNotFoundException {  
s.defaultReadObject();  
// personalizarea deserializarii  
. . .  
// actualizarea starii obiectului (daca e necesar)  
}  
Metodele writeObject si readObject sunt responsabile cu serializarea clasei în care sunt definite,  
serializarea superclasei sale fiind facuta automat (si implicit). Daca însa o clasa trebuie sa-si coordoneze serializarea proprie cu serializarea superclasei sale, atunci trebuie sa implementeze interfata Externalizable.

Exista cazuri când dorim ca unele variabile membre sau sub-obiecte ale unui obiect sa nu fie salvate automat în procesul de serializare. Acestea sunt cazuri comune atunci când respectivele câmpuri reprezinta informatii confidentiale, cum ar fi parole, sau variabile auxiliare pe care nu are rost sa le salvam. Chiar declarate ca private în cadrul clasei aceste câmpuri participa la serializare. O modalitate de a controla serializare este implementarea interfetei Externalizable, asa cum am vazut anterior. Aceasta metoda este însa incomoda atunci când clasele sunt greu de serializat iar multimea câmpurilor care nu trebuie salvate este redusa.  
Pentru ca un câmp sa nu fie salvat în procesul de serializare atunci el trebuie declarat cu modificatorul **transient** si trebuie sa fie ne-static. De exemplu, declararea unei parole ar trebui facuta astfel:  
transient private String parola; //ignorat la serializare  
<  
**Atentie**  
Modificatorul static anuleaza efectul modificatorului transient;

static transient private String parola; //participa la serializare  
De asemenea, nu participa la serializare sub-obiectele neserializabile ale unui obiect, adica cele  
ale caror clase nu au fost declarate ca implementând interfata Serializable (sau  
Externalizable).  
Exemplu: (câmpurile marcate 'DA' participa la serializare, cele marcate 'NU',  
nu participa)  
class A { ... }  
class B implements Serializable { ... }  
public class Test implements Serializable {  
private int x; // DA  
transient public int y; // NU  
static int var1; // DA  
transient static var2; // **DA**  
A a; // NU  
B b1; // DA  
transient B b2; // NU  
}  
Atunci când o clasa serializabila deriva dintr-o alta clasa, salvarea câmpurilor clasei parinte se va  
face doar daca si aceasta este serializabila. In caz contrar, subclasa trebuie sa salveze explicit si  
câmpurile mostenite.

Reflecția

*Reflecţia* reprezintă capacitatea unui program de a se observa pe sine însuşi, de a se  
uita la propria structură, de a se auto-observa în ceea ce priveşte modul de funcţionare şi  
posibil chiar de a se auto-modifica pe sine însuşi.

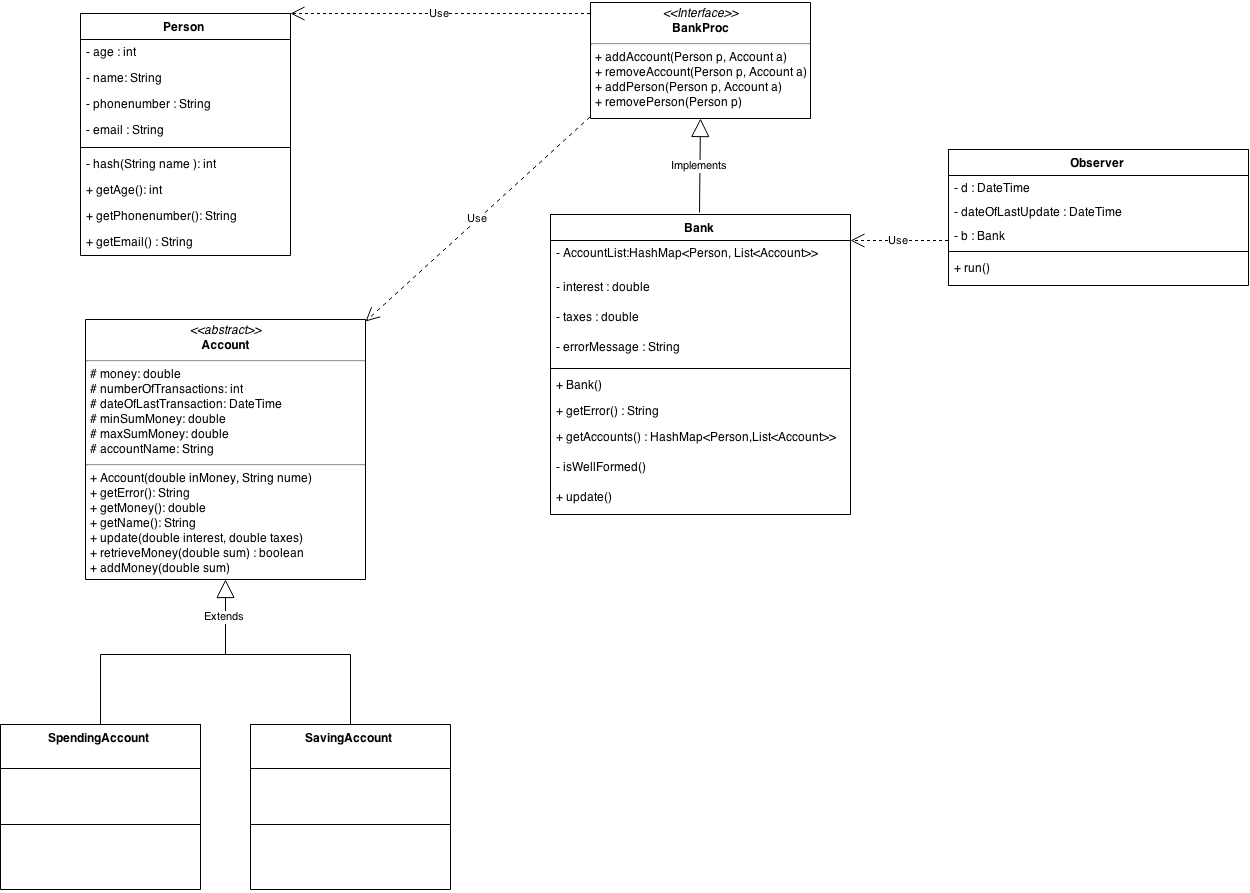
Există două laturi ale reflecției. *Reflecţia de structură* se referă la structuri de date şi  
la cod. Aceasta înseamnă capacitatea unui program de a-şi observa propriile structuri de  
date şi posibil propriul cod.

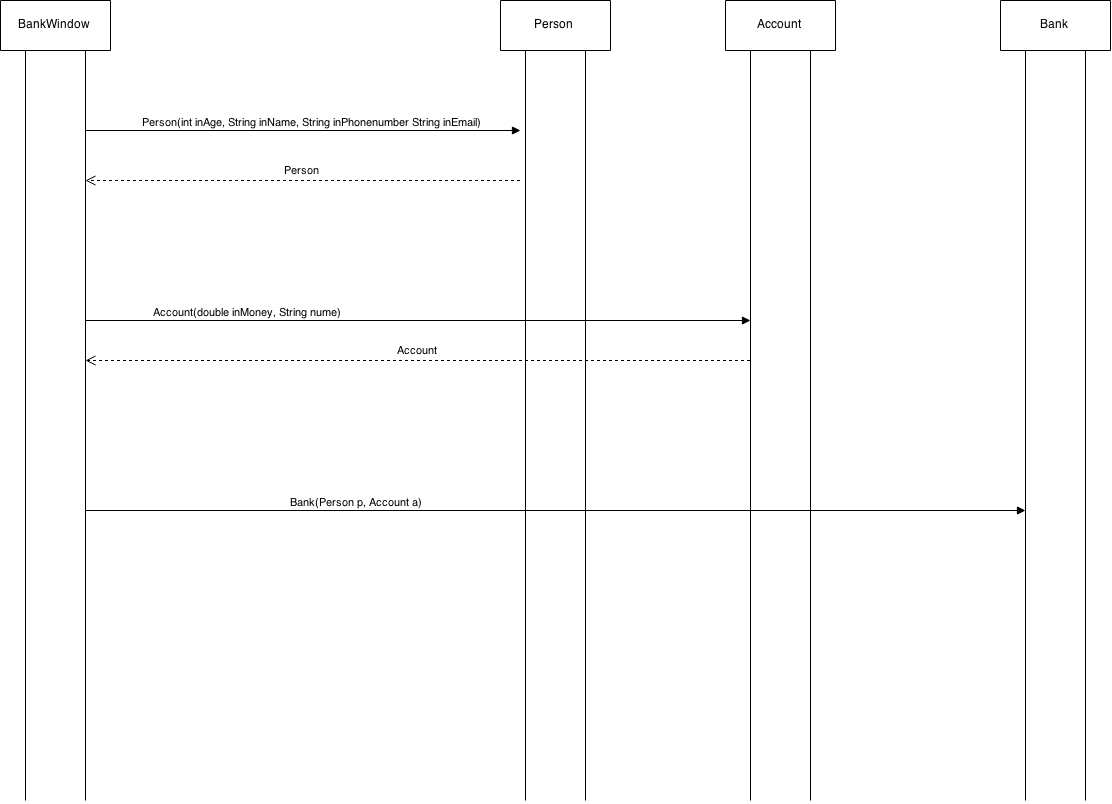
*Reflecţia comportamentală* este capacitatea unui program de a-şi analiza propriul  
comportament sau de a vedea ce se întâmplă atunci când, de exemplu, este executată o  
anumită comandă sau instrucţiune. Adesea un program nu cunoaşte ce se întâmplă atunci  
când o instrucţiune este executată. Programatorii, cel mult, (speră că) ştiu care este  
efectul execuţiei respectivei instrucţiuni, dar în cele din urmă rezultatul final al execuţiei  
depinde de compilator şi poate de interpretorul ce rulează respectivul program.

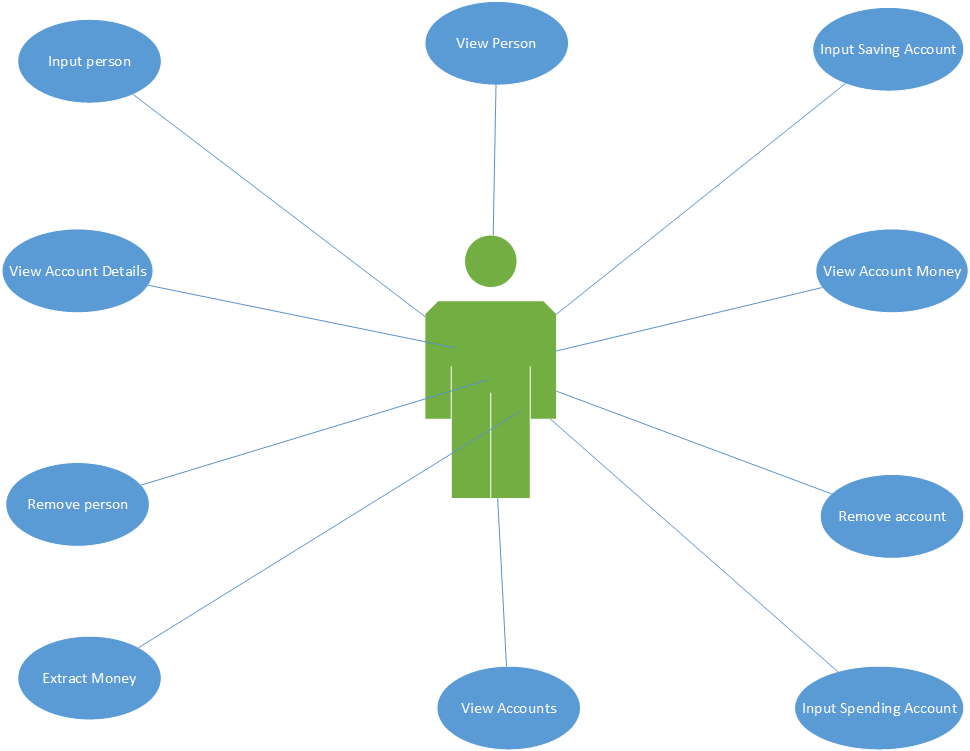
Există două operaţii de bază ce sunt folosite în cadrul programului pentru a suporta  
reflecţia. Una este *introspecţia* (*eng.* introspection), ce înseamnă că programul se poate  
uita la propria structură şi chiar la diverse detalii de implementare sau la comportamentul  
pe care îl are atunci când este executat. Cea de-a doua proprietate este *intercesiunea* (*eng.*  
intercession), ce înseamnă că programul poate să-şi modifice chiar propria structură,  
propriul cod sau chiar propriul comportament. E similar modificării definiţiei rolului pe  
care îl au anumite instrucţiuni, a efectului execuţiei respectivelor instrucţiuni în cadrul  
limbajului, similar redefinirii limbajului.  
Reflecţia poate apărea fie înainte ca programul să se execute, când se numeşte  
*reflecţie statică*, sau în timpul execuţiei, şi se numeşte *reflecţie dinamică*. Reflecţia  
statică apare de exemplu atunci când avem părţi ce sunt executate înainte de corpul  
  
principal, înainte de execuția funcției *main*. Ea poate conduce la adăugarea de exemplu  
de noi clase. Reflecţia dinamică apare la runtime, când aplicaţia se poate uita la propriile  
structuri de date, posibil la propriul cod.  
Reflecţia poate fi periculoasă dacă nu este folosită corect. Dacă avem un program  
care se poate modifica pe sine însuşi atunci când rulează înseamnă că el poate introduce  
uşor şi noi defecte, bug-uri, chiar în timp ce rulează. Astfel de probleme sunt mai dificile,  
dacă nu imposibile, de analizat, depistat și reparat.

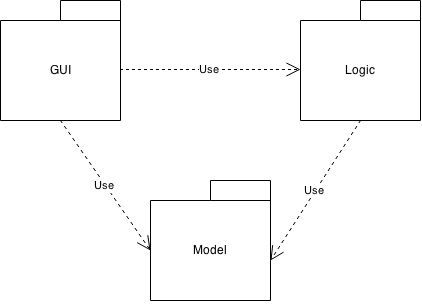
În programarea orientată obiect modul standard de implementare a reflecţiei este  
numit *protocol meta-obiect* (*eng*. metaobject protocol). Pentru a înţelege însă conceptul  
trebuie să vedem ce înseamnă un *metaobiect*. Să presupunem că avem un program format  
din diverse părţi, interne, pe care de obicei nu le putem vedea/accesa din cadrul  
programului. Ca programator cunoaştem doar că programul e format din clase de  
exemplu, dar programul însuşi, în timp ce rulează, nu-şi vede propriile clase sau propriul  
cod. El doar rulează prin cod, dar aplicaţia nu poate de exemplu inspecta codul respectiv  
în timp ce rulează şi să spună „hmm, instrucţiunea aceasta parcă nu îmi place, parcă mai  
bine aş sări-o şi aş executa următoarea instrucţiune mai bine”. Pentru a face detaliile de  
implementare accesibile programului se recurge adesea la reprezentarea detaliilor interne  
de implementare sub formă de instrucţiuni de program, clase, şi toate acestea sunt formă  
de obiecte accesibile în program. Acest artificiu este folosit pentru reprezentarea  
detaliilor interne, cunoscute ca *metadate* - date ce reflectă mai mult decât lucrurile  
obişnuite cu care lucrează programul, de care un program în mod normal nu este  
conştient, date despre program. Deci metadatele sunt date ce descriu ceva precum  
structura datelor.  
În concordanţă şi obiectele ce descriu detalii interne de implementare ale obiectelor se  
numesc *metaobiecte*. Un metaobiect poate descrie de exemplu cum arată, care sunt  
detaliile de implementare ale codului, cum arată clasele şi aşa mai departe, chiar cum  
arată interpretorul sau compilatorul pe care le folosim pentru execuţia codului.  
Dacă avem acces la detaliile interne ale programului şi la utilizarea de operaţii de  
introspecţie a respectivelor detalii interne putem în continuare să ne referim la un detaliu  
de implementare internă a aplicaţiei pentru care să obţinem deci un metaobiect. Acesta  
este de fapt modul în care putem introspecta ceva. Un metaobiect reprezintă un obiect  
normal, doar că este folosit pentru a descrie ceva din structura programului. Pentru a  
modifica programul, pentru a folosi intercesiunea, un program se modifică pe sine însuşi  
la runtime prin modificarea metaobiectelor iar respectivele modficări ajung să se reflecte  
înapoi în structura programului. De exemplu, dacă modificăm numele unei clase. Putem  
obţine un obiect descriind structura clasei şi acesta poate avea o variabilă *name*. Dacă  
modificăm valoarea curentă a respectivei variabile ar putea ca respectiva clasă să nu mai  
fie cunoscută decât prin noul nume. Astfel putem folosi deci mecanismele de introspecţie  
şi intercesiune, prin folosirea metaobiectelor. Modalitatea prin care putem folosi aceste  
mecanisme, prin care putem obţine metaobiecte, se numeşte *protocol de metaobiecte*.  
Altfel spus, reprezintă API-ul reflecţiei într-un program orientat pe obiecte.

Prezentăm în continuare diagramele realizate pentru program :









# 4. Implementare și testare

Clasele care implementează contul sunt : SavingAccount și SpendingAccount. Acestea conțin următoarele câmpuri :

**protected** **double** money;

**protected** **int** numberOfTransactions;//depends on the type of the account

**protected** **DateTime** dateOfLastTransaction;//i need this to see if i may allow the user to retrieve money or not.

**protected** **double** minSumMoney;//we have a minimum sum to retrieve. if the sum is lower than this, we will not give money to the user

**protected** **double** maxSumMoney;//we have a maximum limit for the sum in both cases

**protected** **String** errorMessage;

**protected** **String** accountName;

Câmpul money reprezintă cantitatea de bani stocată în contul respectiv. Ea este modificată dinamic, în funcție de data curentă și de operațiile pe care utilizatorul le efectuează(adăugare, respective extragere).

numberOfTransactions reprezintă numărul de tranzacții efectuate curent asupra contului selectat. Pentru SavingAccount, el numără unu, și se reface 0 doar atunci când revine clientul pentru a mai efectua o extragere luna următoare. La SpendingAccount, după trei extrageri, el blochează extragerile până a doua zi.

dateOfLastTransaction este utilizat pentru a memora data ultimei tranzacții efectuate.

minSumMoney, respective maxSumMoney reprezintă suma minimă(maximă) care poate fi extrasă din cont. Valorile lor depend de tipul de cont.

errorMessage este setat atunci când apare o eroare la extragere(numarul de tranzacții limită depășit, etc.)

accountName – numele contului.

Clasa Person conține date utile despre persoana care deține conturile : Nume, vârstă, număr de telefon, email etc.

Clasa Bank conține :

**private** **HashMap**<Person, List<Account>> AccountList;

**private** **double** interest = 0.25;

**private** **double** taxes = 30.00;

**private** **String** errorMessage;

HashMap-ul asociază persoane la o listă de clienți, interest reprezintă dobânda iar taxes reprezintă taxele platite(valoare fictivă). errorMessage are același rol ca și la Account : în cazul unei erori apărute la adăugare sau ștergere, el este setat.

Testele au fost realizate folosind biblioteca JUnit în care s-au testat cazuri de adăugare, ștergere și diferențe la adăugare.

# 5.Rezultate

În urma realizării aplicației date, s-a obținut un software menit pentru gestiunea depozitelor mari de bani, cum ar fi cele prezente într-o bancă. Se pot creea conturi pentru persoane la care sunt asociate conturi în depozit pentru a facilita extragerea banilor în diverse moduri : salvare sau cheltuire. Aceste tipuri de conturi pot fi foarte ușor extinse și la altele, în funcție de nevoia utilizatorului

# 6.Concluzii și dezvoltări ulterioare

Prin dezvoltarea aplicației date, s-au învățat trei tehnici principale de programare :

-Design by contract

-Serializare

-Reflecție

Prin design by contract se stabilește un contract cu utilizatorul software-ului, astfel încât atât cel care a propus software-ul, cât și cel care îl utilizează trebuie să îl respecte. Daca oricare dintre cele două părți nu îl respectă, atunci contractul este încălcat și apare un bug în software.

Prin serializare se pot salva date într-un fișier sub formă binară, urmând ca mai apoi să fie deserializate pentru a fi citite(atât pe aceeași mașină, cât și pe alta)

Prin reflecție, se pot citi dinamic anumite câmpuri urmate pentru afișare, indiferent de obiectul care urmează a fi deserializat.

Pentru dezvoltări ulterioare, se pot creea noi tipuri de conturi în bancă pentru a facilita nevoia utilizatorului și centralizarea datelor(trimiterea fișierelor salvate prin serializare către un server central din care vor putea citi toate terminalele legate și le vor afișa).

# 7.Bibliografie

1. Cristian Frăsinaru – Curs practic de Java

2.Laborator Design după contract

3. Curs reflexie

4. Curs serializare