



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

BankManagement

Student: Maxim Bogdan-Gheorghe



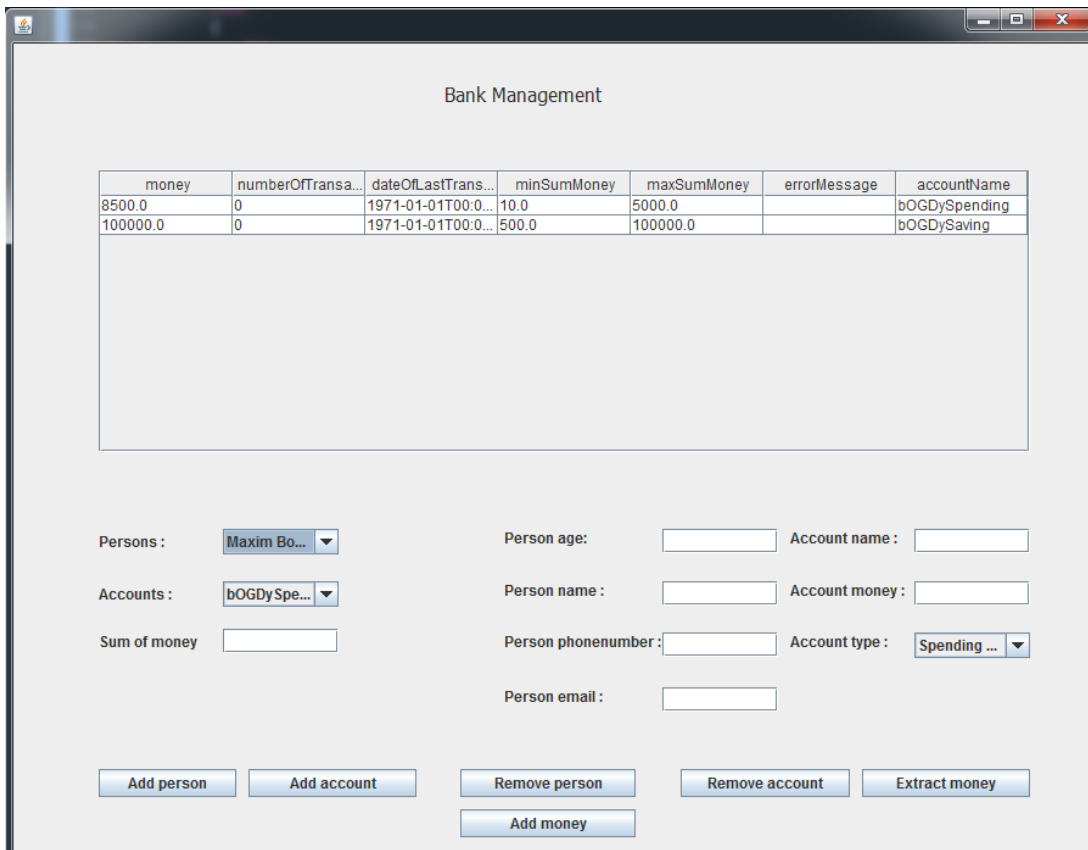
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Cuprins

1. Obiectivul temei.....	3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare	4
3. Proiectare.....	5
4. Implementare și testare.....	18
5. Rezultate	19
6. Concluzii și dezvoltării ulterioare	19
7. Bibliografie.....	20

1. 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.



The screenshot shows a "Bank Management" application window. It features a table with account data, a list of persons and accounts, and various input fields for user information. At the bottom, there are buttons for adding and removing persons and accounts, as well as extracting money.

money	numberOfTransa...	dateOfLastTrans...	minSumMoney	maxSumMoney	errorMessage	accountName
8500.0	0	1971-01-01T00:0...	10.0	5000.0		bOGDySpending
100000.0	0	1971-01-01T00:0...	500.0	100000.0		bOGDySaving

Persons : Maxim Bo...
Accounts : bOGDySpe...
Sum of money :
Person age:
Account name :
Person name :
Account money :
Person phonenumber :
Account type : Spending ...
Person email :

Add person Add account Remove person Remove account Extract money
Add money

2. Analiza problemei, modelare, scenarii, cazuri de utilizare

Primul pas efectuat în analiza problemei curente este studierea și analizarea diagramei propuse pentru realizare. Acest pas înlocuiește analiza 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 adăugate, 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 minimă 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 : Adăugare/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 de 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 și stările în care se poate afla programul - aceste restricții sunt date sub forma unor:
 - a) precondiții: reprezintă obligațiile pe care datele de intrare ale unei metode trebuie să le respecte pentru ca metoda să funcționeze corect
 - b) postcondiții: reprezintă garanțiile pe care datele de ieșire ale unei metode le oferă
 - c) invariante: reprezintă condiții impuse stărilor în care programul se poate afla la un moment dat

Cel care a implementat clasa sau metoda respectivă îi spune utilizatorului ce date sunt considerate valide ca date de intrare. Aceasta îl scuteste să folosească teste de validare a datelor, care în unele cazuri doar ar încetini algoritmul, ele fiind redundante cu teste care deja sunt făcute pentru validarea datelor, de exemplu imediat ce au fost introduse de către utilizator.

În același timp utilizatorul știe din contract la ce date posibile de ieșire sau stări intermediare să se aștepte și atunci poate să-și optimizeze propriul cod în funcție 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 semnatura 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 *simplic de utilizat* pentru salvarea și restaurarea a datelor.

Permite *persistența obiectelor*, ceea ce înseamnă 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 și starea lui refăcută. Acest tip de persistență a obiectelor se numește *persistența ușoară*, întrucât ea trebuie efectuată explicit de către programator și 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"



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

(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 recursiv 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 întregă rețea, ceea ce înseamnă că un algoritm general de salvare a stării unui obiect nu este tocmai facil. În 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 văzut, 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 încapsulate într-un 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 o asemenea abordare nu este în general suficientă, deoarece pot apărea probleme cum ar fi :

- datele obiectului pot fi instanțe ale altor obiecte
- în unele cazuri, este necesară și salvarea tipului datei
- unele câmpuri fac referință la același obiect

Asadar, prin serializare sunt surprinse atât datele, semnatura clasei (numele metodelor și definiția lor - nu și implementarea) precum și starea obiectului.

Pentru a putea fi serializat un obiect trebuie să fie instanța a unei clase care implementează una din interfețele :

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

Interfața `Serialize` nu are nici o metodă, ea da doar posibilitatea de a specifica faptul că se dorește ca o anumită clasă să poată fi serializată. Declarația unei astfel de clase ar fi :

```
class ClasaSerializabila implements Serializable { ... }
```

În urma serializării obiectele pot fi salvate într-un fișier, în același fișier putând fi salvate și mai multe obiecte. Operațiile de intrare/ieșire la nivelul obiectelor se realizează prin intermediul unor fluxuri de obiecte, implementate de clasele `ObjectInputStream` și `ObjectOutputStream`. Salvarea unui obiect într-un fișier se realizează astfel :

```
MyObject o = new MyObject();  
FileOutputStream fout = new FileOutputStream("fisier");  
ObjectOutputStream sout = new ObjectOutputStream(fout);  
sout.writeObject(o);
```

Restaurarea unui obiect salvat într-un fișier se face într-o manieră asemănătoare:

```
FileInputStream fin = new FileInputStream("fisier");  
ObjectInputStream sin = new ObjectInputStream(fin);  
o = (MyObject) sin.readObject();
```

Pe lângă metodele de scriere/citire a obiectelor cele două clase pun la dispoziție și metode pentru scrierea tipurilor de date primare, astfel încât apelurile 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();
```




UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

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

```
istream.close();
```

ObjectInputStream și ObjectOutputStream implementează indirect interfețele DataInput, respectiv DataOutput, interfețe ce declară metode atât pentru scrierea/citirea datelor primitive, cât și 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 apelează la rândul lor metodele implicite de transfer defaultWriteObject și defaultReadObject (având aceleași semnături ca mai sus).

Personalizarea serializării se realizează prin supradefinirea (într-o clasă serializabilă!) a metodelor writeObject și readObject, modificând astfel acțiunea lor implicită.

Metoda writeObject controlează ce date sunt salvate și este uzual folosită pentru a adăuga informații suplimentare la cele scrise implicit de metoda defaultWriteObject.

Metoda readObject controlează modul în care sunt restaurate obiectele, citind informațiile salvate și, eventual, modificând starea obiectelor citite astfel încât ele să corespundă anumitor cerințe.

Aceste metode trebuie obligatoriu să aibă următorul format:

```
private void writeObject(ObjectOutputStream stream)
```

```
    throws IOException
```

```
private void readObject(ObjectInputStream stream)
```

```
    throws IOException, ClassNotFoundException
```

De asemenea, uzual, primul lucru pe care trebuie să îl facă aceste metode este apelul la metodele standard de serializare a obiectelor defaultWriteObject, respectiv defaultReadObject și abia apoi să execute diverse operațiuni suplimentare. Forma lor generală este:

```
private void writeObject(ObjectOutputStream s)
```

```
    throws IOException {
```

```
    s.defaultWriteObject();
```

```
    // personalizarea serializării
```

```
}
```



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

```
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 clasă serializabilă derivă dintr-o altă clasă, salvarea câmpurilor clasei părinte se va face doar dacă și aceasta este serializabilă. În caz contrar, subclasa trebuie să salveze explicit și câmpurile moștenite.

Reflecția

Reflecția reprezintă capacitatea unui program de a se observa pe sine însuși, de a se uita la propria structură, de a se auto-observa în ceea ce privește modul de funcționare și posibil chiar de a se auto-modifica pe sine însuși.

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

Reflecția comportamentală este capacitatea unui program de a-și analiza propriul comportament sau de a vedea ce se întâmplă atunci când, de exemplu, este executată o anumită comandă sau instrucțiune. Adesea un program nu cunoaște ce se întâmplă atunci când o instrucțiune este executată. Programatorii, cel mult, (speră că) știu care este efectul execuției respectivei instrucțiuni, dar în cele din urmă rezultatul final al execuției depinde de compilator și poate de interpretorul ce rulează respectivul program.

Există două operații de bază ce sunt folosite în cadrul programului pentru a suporta reflecția. Una este *introspecția* (eng. introspection), ce înseamnă că programul se poate uita la propria structură și 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-și modifice chiar propria structură, propriul cod sau chiar propriul comportament. E similar modificării definiției rolului pe care îl au anumite instrucțiuni, a efectului execuției respectivelor instrucțiuni în cadrul limbajului, similar redefinirii limbajului.

Reflecția poate apărea fie înainte ca programul să se execute, când se numește *reflecție statică*, sau în timpul execuției, și se numește *reflecție dinamică*. Reflecția statică apare de exemplu atunci când avem părți ce sunt executate înainte de corpul



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

principal, înainte de execuția funcției *main*. Ea poate conduce la adăugarea de exemplu de noi clase. Reflecția dinamică apare la runtime, când aplicația se poate uita la propriile structuri de date, posibil la propriul cod.

Reflecția poate fi periculoasă dacă nu este folosită corect. Dacă avem un program care se poate modifica pe sine însuși atunci când rulează înseamnă că el poate introduce ușor și 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 reflecției este numit *protocol meta-obiect* (eng. metaobject protocol). Pentru a înțelege însă conceptul trebuie să vedem ce înseamnă un *metaobiect*. Să presupunem că avem un program format din diverse părți, interne, pe care de obicei nu le putem vedea/accesa din cadrul programului. Ca programator cunoaștem doar că programul e format din clase de exemplu, dar programul însuși, în timp ce rulează, nu-și vede propriile clase sau propriul cod. El doar rulează prin cod, dar aplicația nu poate de exemplu inspecta codul respectiv în timp ce rulează și să spună „hmm, instrucțiunea aceasta parcă nu îmi place, parcă mai bine aș sări-o și aș executa următoarea instrucțiune mai bine”. Pentru a face detaliile de implementare accesibile programului se recurge adesea la reprezentarea detaliilor interne de implementare sub formă de instrucțiuni de program, clase, și toate acestea sunt formă de obiecte accesibile în program. Acest artificiu este folosit pentru reprezentarea detaliilor interne, cunoscute ca *metadata* - date ce reflectă mai mult decât lucrurile obișnuite cu care lucrează programul, de care un program în mod normal nu este conștient, date despre program. Deci metadatale sunt date ce descriu ceva precum structura datelor.

În concordanță și 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 și așa mai departe, chiar cum arată interpretorul sau compilatorul pe care le folosim pentru execuția codului.

Dacă avem acces la detaliile interne ale programului și la utilizarea de operații de introspecție a respectivelor detalii interne putem în continuare să ne referim la un detaliu de implementare internă a aplicației pentru care să obținem 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 însuși la runtime prin modificarea metaobiectelor iar respectivele modificări ajung să se reflecte înapoi în structura programului. De exemplu, dacă modificăm numele unei clase. Putem obține un obiect descriind structura clasei și 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 introspecție și intercesiune, prin folosirea metaobiectelor. Modalitatea prin care putem folosi aceste



UNIVERSITATEA TEHNICĂ

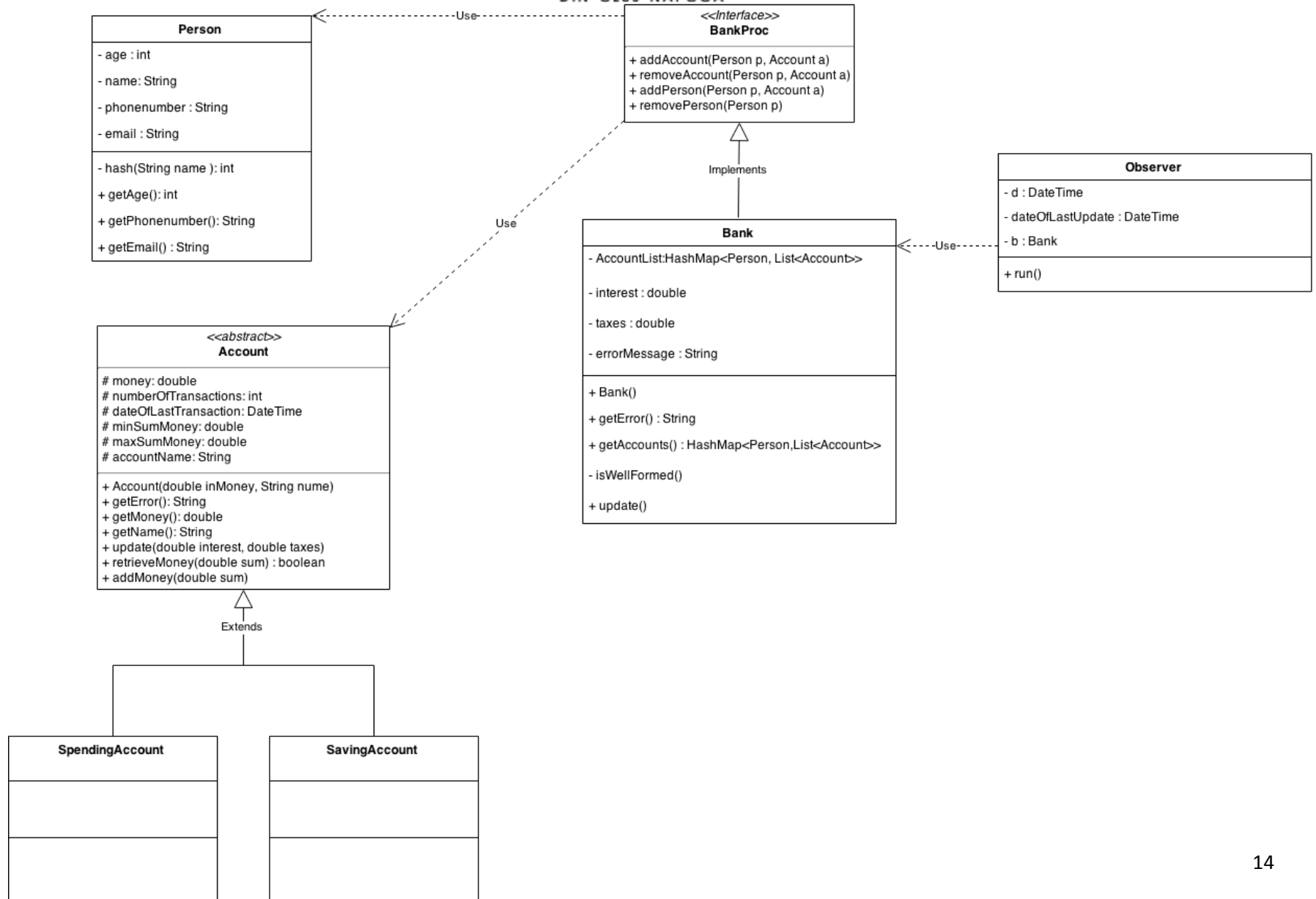
DIN CLUJ-NAPOCA

mecanisme, prin care putem obține metaobiecte, se numește *protocol de metaobiecte*. Altfel spus, reprezintă API-ul reflecției într-un program orientat pe obiecte.

Prezentăm în continuare diagramele realizate pentru program :

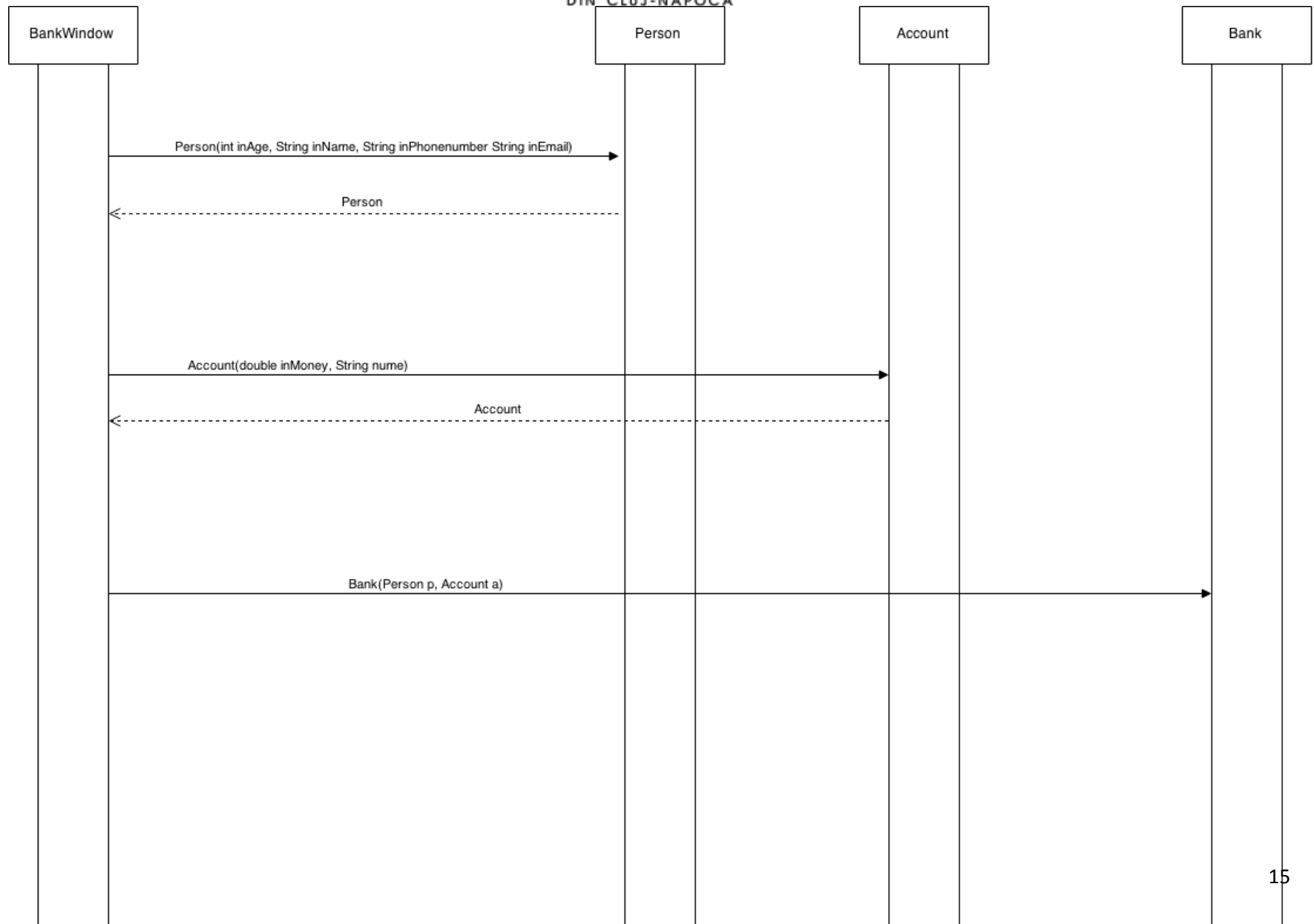


UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA



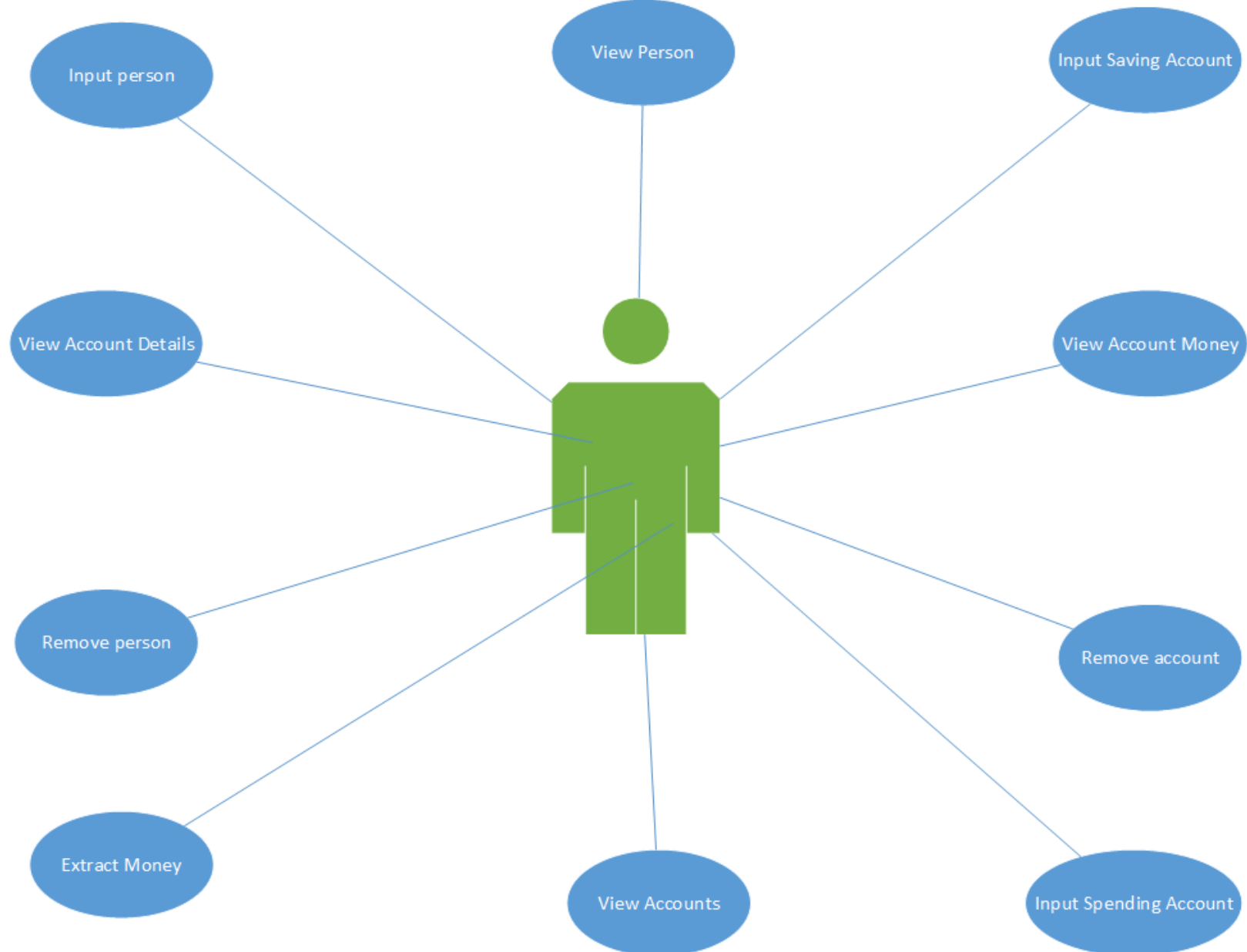


UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA



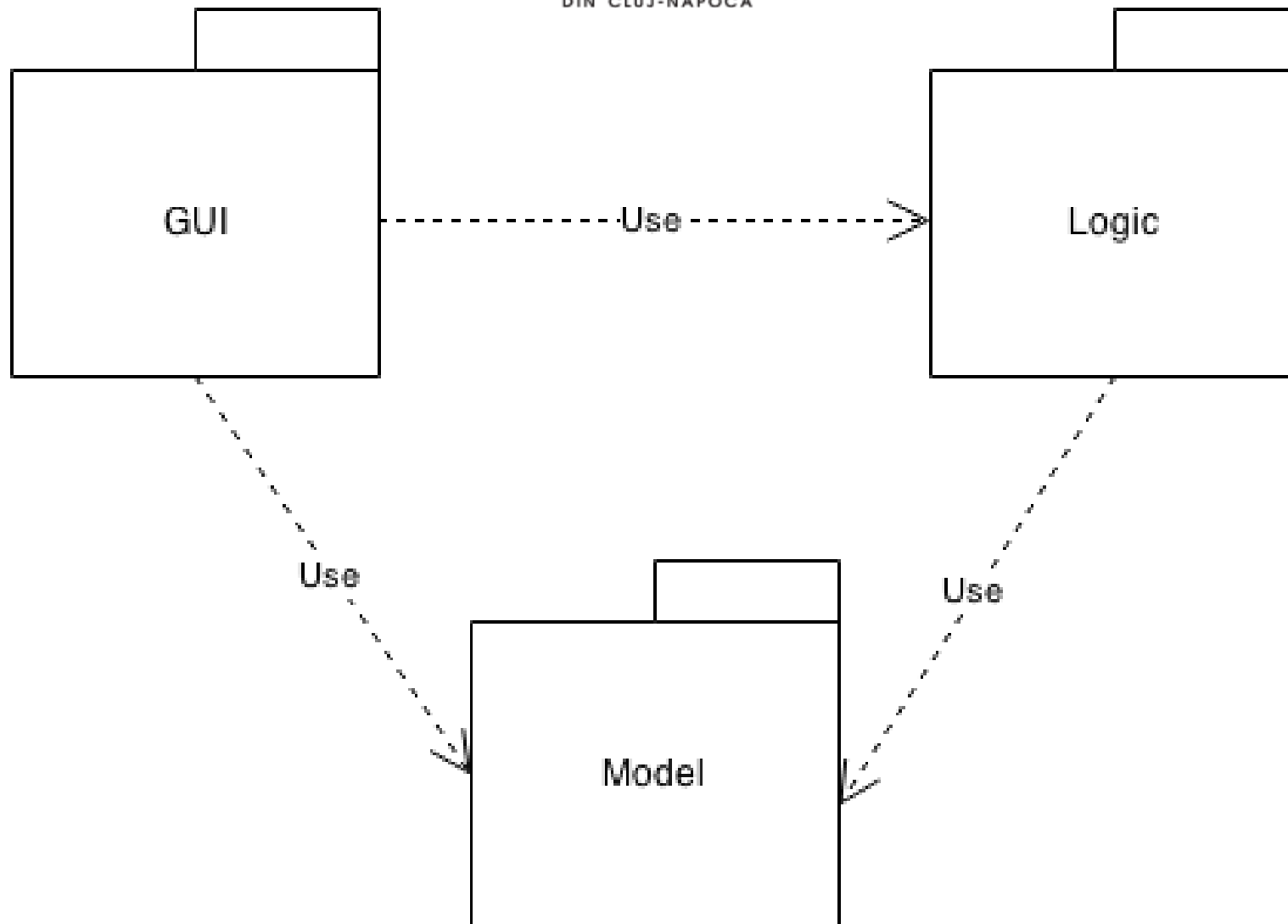


UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA





UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA



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 crea 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. Dacă 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 crea noi tipuri de conturi în bancă pentru a facilita nevoia utilizatorului și centralizarea datelor(trimiterăa fișierelor salvate prin serializare către un server central din care vor putea citi toate terminalele legate și le vor afișa).



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

7. Bibliografie

1. Cristian Frăsinaru – Curs practic de Java
2. Laborator Design după contract
3. Curs reflexie
4. Curs serializare