Documentatie

**Tema 4**

Ghitun Patricia Roxana

Grupa 30227

2018

1. **Obiectivul Temei**

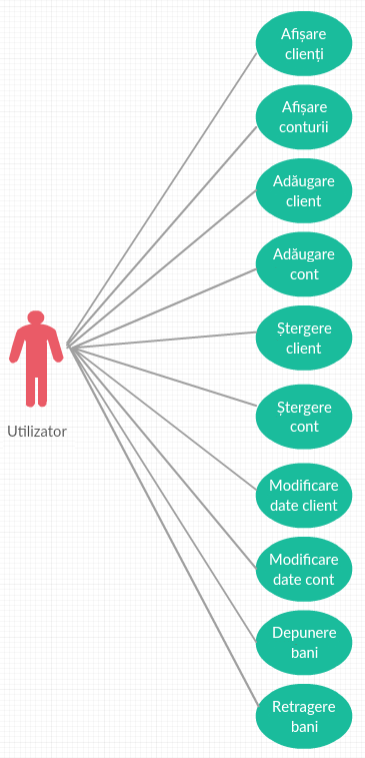
Obiectivul acestei teme este proiectarea si implementarea unei aplicatii pentru managementul unei banci . Aceasta aplicatie implementeaza toate functiile pe care un client intr-o banca cum ar fi adaugarea unui nou cont de cheltuieli sau economii . Deoarece nu a fost folosita o baza de date , de data aceasta folosim un fisier pentru a stoca toate datele. Astfel cand rulam aplicatia preluam toate datele din fisier , iar la finalul ei toate datele din aplicatie le vom scrie inapoi in fisier .

1. **Analiza problemei**

Sistemele de gestiune a conturilor dintr-o banca sunt oferă acces rapid și simplu la informațiile despre un anumit client și conturile asociate acestuia.

In primul rand, pentru rezolvarea temei avem nevoie de o buna intelegere a principiului

Serializare si deserializare. Serializarea obiectelor = salvarea si restaurarea starii obiectelor.

Obiectele oricarei clase care implementeaza interfata Serializable, pot fi salvate intr-un

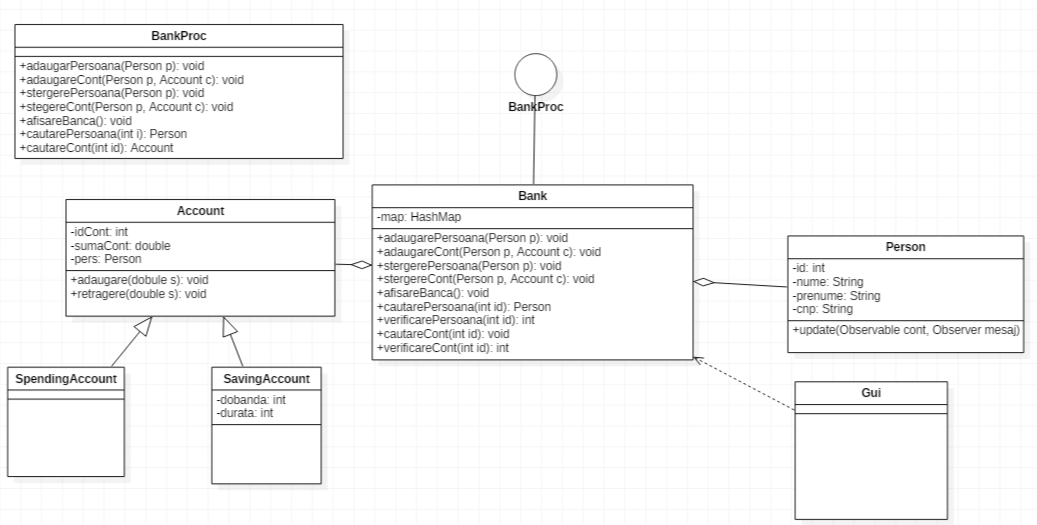
stream si restaurate din acesta. Pachetul java.io contine doua clase speciale

ObjectInputStream respectiv ObjectOutputStream pentru serializarea tipurilor primitive.

Aplicatia trebuie sa realizeze o interfata cu utlilizatorul si banca. Utilizatorul trebuie sa poata sa isi creeze conturi noi ( de cheltuieli sau de economii ), sa stearga conturi existente , sa editeze un cont pe care il detine , etc . Astfel toate datele din aplicatie trebuie retinute intr-un fisier , iar la fiecare deschidere a aplicatiei toate datele din fisier vor fi preluate in aplicatie, similar la iesirea din aplicatie toate datele vor fi scrise inapoi in fisier.

1. **Proiectare**

Diagrama UML :



Aplicatia este impartita in trei pachete :

* Pachetul **model** : pachetul ce contine clasele in care se realizeaza logica aplicatiei ( contine clasele Bank , BankProc , Person , Account etc)
* Pachetul **presentation :** acesta contine clasele de interfata grafica ce creeaza interfata cu utilizatorul (frame-ul principal si frame-urile pentru operatii – fiecare operatie este implementata intr-un nou frame );
* Pachetul **start** : ce contine clasa Main din care se ruleaza aplicatia si de asemenea clasa Teste ce contine testele unitare pentru aplicatie .

Structuri de date folosite :

O structură de date se referă la metode de organizare a unităților de date astfel incat ele sa organizeze datele cat mai eficient . Realizarea și menținerea structurilor de date specifice contribuie la îmbunătățirea algoritmilor implementați.

Pachetul de utilități Java, java.util.\*, conține un număr mare de clase și interfețe pentru a facilita stocarea și manipularea unor grupuri de obiecte. Java 5.0. a reimplementat Java Collections Framework folosind tipuri generice care permit unui programator să specifice un tip pentru obiectele stocate în structură. Pentru această aplicație, clasa Bank folosește o structură de date de tip HashMap<Person, <Set<Account>>.

Interfețele Set și Map sunt similare cu interfața List, deoarece există mai multe clase în cadrul colecțiilor care le implementează. Interfața Set este modelată după principiile matematice a mulțimii . Interfața Set nu are alte metode decât cele pe care le moștenește de la Collection, dar impune regula să nu existe duplicate. Interfața Map <K, V> este modelată după căutarea definițiilor cuvintelor dintr-un dicționar.

In cazul de fata , cheile de obiecte folosite sunt de tip Person , iar valoarea asociata cheii este o multime de conturi ( adica de tip Account). Ce este specific HashMap-urilor este faptul ca un element nu este asociat cu o cheie strict dupa identitatea obiectului ci mai degraba dupa continutul cheii sale . De aceea , se va suprascrie metoda equals() .

Avantanjele folosirii colectiilor de acest tip sunt reducerea efortului de programare si cresterea vitezei si calitatii programului deoarece implementarile acestor colectii sunt de inalta performanta si folosesc algoritmi cu timp de lucru bine optimizat.

1. **Implementare**

Aplicatia contine trei pachete : model , presentation si start .

**Pachetul model**

Pachetul model contine clasele Bank , Account , SpendingAccount, SavingAccount si interfata BankProc .

In **interfata BankProc** au fost definite metodele de adaugare persoana , adaugare cont, stergere persoana , stergere cont , afisare banca , cautare persoana si cautare cont . Pentru fiecare metoda au fost definite preconditii si postconditii care sunt implementate ulterior odata cu implementarea metodelor in clasa Bank ( ce implementeaza interfata BankProc).

**Clasa Person**  implementeaza interfata Serializable pentru serializarea datelor despre o persoana si de asemenea si interfata Observer deoarece este un observatory asupra contului (Observable Desgin Pattern ) . Aici sunt stocate datele ce vor fi retinute pentru fiecare persoana din banca , precum : id-ul persoanei , numele , prenumele si cnp-ul . Metodele din aceasta clasa sunt cele de set si get pentru attribute dar si suprascrierea metodelor de hashCode , equals si update . Metoda update a fost create deoarce o persoana este un observator al unui cont , va primi ca și argumente un obiect de tip Observable (reprezentat de Account în cazul nostru) și un String. Aceasta va fi apelata doar cu string-ul dorit .

Mai exact , in clasa Saving sau Spending Account , in momentul in care se adauga sau se retrag bani din cont , se apeleaza metoda setChanged si notifyObservers ce anunta observarii ca s-a modificat ceva intr-un cont si se afiseaza un mesaj (cel dat prin parametru) in consola ( de ex : S-au adaugat bani in contul c);

**Clasa Account :** aceasta clasa a fost creata pentru a retine detaliile despre un cont din banca. Orice cont are un id de tip intreg , o suma de bani de tip double si un obiect de tip Person ( persoana ce detine contul respectiv ) . Aceasta clasa contine metodele obisnuite de set si get si afisare dar si de asemenea metodele de adaugare si retragere de bani in sau din contul existent . Aceste metode au ca parametru suma ce urmeaza sa fie depusa sau retrasa.

**Clasa SpendingAccount**  : Aceasta clasa reprezinta un cont de cheltuieli si extinde clasa Account . Constructorul acestei clase apeleaza constructorul super din clasa Account pentru a initializa datele . Aici sunt implementate cele doua metode de adaugare si retragere bani din cont . Intr-un cont de cheltuieli se poate adauga orice suma de bani si se poate retrage orice suma de bani ( in limita stocului depus ) . Acest cont nu foloseste dobanda . In momentul in care se adauga sau se retrag bani , se apeleaza metodele de setChanged si notifyObservers ce anunta utilizatorul ca au fost facute modificari in cont .

**Clasa SavingAccount**  : Aceasta clasa reprezinta un cont de economii si extinde de asemenea clasa Account . Constructorul acestei clase apeleaza constructorul super din clasa Account. De asemenea , si aici sunt implementate cele doua metode de adaugare si retragere bani din cont , dar si o metoda numita “dobanda” ce calculeaza noua suma din cont in functie de dobanda . Intr-un cont de cheltuieli se poate adauga o suma de bani >= 1000 si se poate retrage o suma >=1000 ( in limita stocului depus ) . Acest cont dispune de un camp de dobanda si unul de durata . Durata reprezinta durata in care contul a fost deschis ( implicit aceasta durata este de sase luni ) . Dobanda reprezinta dobanda de 2 % cu care va creste sum ape perioada setata de sase luni .

**Clasa Bank :** Aceasta clasa are ca atribut hashMap-ul <Person , Set<Account>> in care se retin persoanele si conturile acestora si implementeaza interfata BankProc in care sunt definite operatiile de baza ce urmeaza sa fie implementate in aceasta clasa .

* Adaugare / stergere de personae din banca
* Adaugare / stergere de conturi din banca
* Afisare banca ( afisarea persoanelor si a conturilor respective )
* Cautare persoana ( aceasta metoda returneaza un obiect de tip Person in cazul in care persoana cu id-ul dat ca parametru este inregistrat in banca )
* Cautare cont ( aceasta metoda returneaza un obiect de tip Account in cazul in care contul cu id-ul dat ca parametru este inregistrat in banca )

Pe langa aceste metode de baza , in clasa Bank mai sunt implementate si metodele :

* setMap
* getPersoane(returneaza un ArrayList<Person> in , afisarePersoane (afiseaza un arrayList de personae dat ca parametru )
* verificare persoana(verifica daca persoana data ca parametru apartine hashmap-ului ( returneaza un int 1 daca persoana nu este adaugat si 0 daca apartine deja hashmap ului ).
* verificareCont : similar cu metoda pentru verificarea existentei unei personae in hash map
* getConturi : returneaza un array list de conturi
* afisareConturi : de tip void

Pentru serializarea datelor, au fost implementate metodele de scriereBanca si citireBanca in aceasta clasa .

**scriereBanca()** : aceasta metoda foloseste instante de FileOutputStream si ObjectOutputStream pentru a scrie un obiect(hashmap-ul) in fisierul declarat la inceputul clasei . Dupa apelarea metodei de writeObject() , se inchide fisierul .

**citireBanca()** : aceasta metoda foloseste instante de FileInputStream si ObjectInputStream pentru a citi din fisierul declarant initial “bank.ser” . Se citeste din fisier folosind metoda readObject , hashMapul si este adaugat intr-un nou obiect de acest tip . Acest obiect este returnat la final .

Tot in aceasta clasa au fost implementate preconditiile si postconditiile mentionate pentru fiecare metoda in interfata BankProc .

Pentru metodele de adaugare s-au folosit asertii pentru a verifica daca persoana pe care dorim sa o adaugam nu exista deja in banca (preconditie) la inceputul metodei , iar la finalul metodei au fost folosite pentru a verifica daca persoana a fost adaugata ( daca a crescut size-ul hashmap-ului ). Similar s-au implementat preconditiile si postconditiile pentru metodele de stergere persoana sau cont – pentru a verifica initial daca persoana exista ( daca nu cumva doresc sa sterg o persoana care nu este inregistrata in banca ) .

De asemenea am mai implementat in acelasi mod asertii si pentru metodele de cautare persoana sau cont pentru a verifica daca persoana sau contul respective exista in banca .

In plus , s-a implementat o metoda numita “size0” , aceasta metoda returneaza true daca dimensiunea hash mapului este 0 si este folosita la metoda de afisare banca ( astfel nu se va putea afisa banca in cazul in care nu a fost adaugat nimic In ea ) .

Operatia principal a acestei clase este cea de adaugarea unui cont nou deoarece trebuie introduce date si despre persoana ce detine acel cont si despre contul in sine . Initial se verifica daca id-ul persoanei exista in map ( daca persoana exista ) , dupa care se verifica daca exista deja o inregistrare cu id-ul contului dat ca parametru si doar dupa aceea este adaugat in map . Pe acelasi principiu se bazeaza si operatie de stergere a unui cont.

Pachetul **presentation** : contine clasele ce realizeaza interfata cu utilizatorul si implicit fac legatura intre GUI si logica aplicatiei .

1. **Testare Junit**

JUnit este o bibliotecă Java care ne ajută să efectuăm o unitate de testare. Unitatea de testare este procesul de examinare a unei clase/metode pentru a verifica dacă aceasta îndeplinește așteptările sau specificația sa.

Pentru a efectua testarea într-un mod mai simplu, când am implementat metodele în clasa Bank, am fololosit Design by Contract, ceea ce include folosirea precondițiilor, postcondițiilor și a aserțiunilor. Prin această metodă, nu mai suntem obligați să scriem aserțiuni în unitatea de testare, deoarece aceasta se va face automat.

Precondiții

Precondițiile specifică condițiile care trebuie îndeplinite înainte ca metoda să poată fi executată. Dacă precondiția dă greș, software-ul ne va arunca o eroare. În mare parte, aserțiunile folosite în această aplicație verifică existența unui cont sau persoană în baza de date.

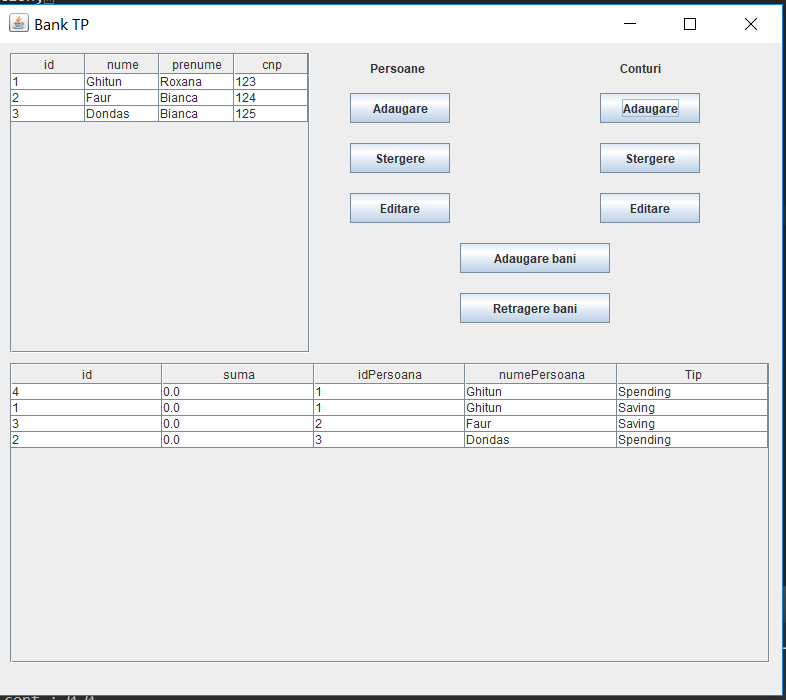
Postcondiții

Postcondițiile trebuie să specifice condiții care trebuie îndeplinite după ce o metodă se finalizează. În consecință, postcondițiile sunt executate după ce s-a finalizat metoda, și ele conțin starea vechiului sistem, starea noului sistem și argumentele metodei. Dacă postcondiția este încălcată, componenta software are un bug. În această aplicație, în postcondiții se va testa lungimea HashMap-ului .

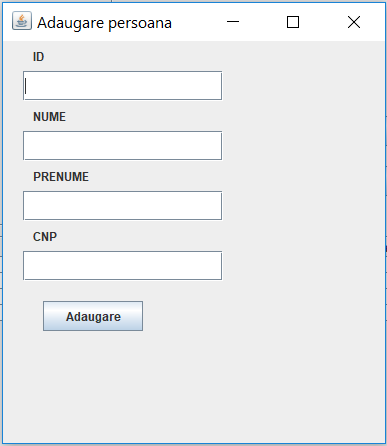
1. **Rezultate**

In momentul in care rulam aplicatie , datele sunt preluate dintr-un fisier “bank.ser ”. Aplicatia va avea mereu toate datele actualizate. Dupa ce am salvat toate datele deja putem

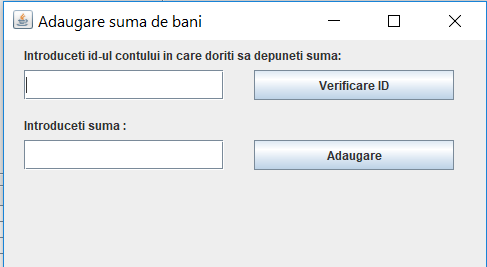
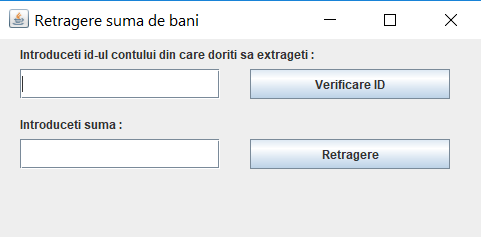
efectua diferite operatii pe conturile din banca. Dupa ce am realizat operatiile dorite) toate datele vor fi scrise inapoi in fisier, astfel ele ramn sigure.



Fiecarei operatii implementate in clasa Bank ii corespunde un buton ce duce la o noua fereastra . De exemplu , interfata pentru adaugarea unei persoanei noi in banca arata astfel :



Operatiile de retragere si adaugare de bani in cont arata astfel :



1. **Concluzii**

Ca dezvoltare ulterioara consider ca este necesara schimbarea implementarii clasei SavingAccount intr-un mod cat mai realist si cat mai contemporan deoarece nu au fost respectate notiuni de banking management .

1. **Bibliografie**

<http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW4_Tema4/HW4_Tema4_Hashing.pdf>

<http://javarevisited.blogspot.ro/2012/01/what-is-assertion-in-java-java.html>

<http://www.tutorialspoint.com/java/java_serialization.htm>

<http://javarevisited.blogspot.ro/2011/02/how-hashmap-works-in-java.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>

<http://javarevisited.blogspot.ro/2012/01/what-is-assertion-in-java-java.html>

<https://stackoverflow.com/questions/7350893/click-event-on-jtable-java>