Tema 4

Student:Oprean Andrei,Grupa:30228

1.Obiectivul temei

Implementarea unei diagrame UML date .

2.Analiza problemei, modelare, scenarii de utilizare

Pentru o a implementa diagram UML data este nevoie de urmatoarele compoente:

-o interfata ‘BankProc’ in care declaram antetul metodelor specifice unei banci

-o clasa ‘Bank’ care va implementa metodele din interfata descrisa mai sus

-o clasa abstracta/interfata ‘Account’ care va defini metodele specifice unui cont. Un cont poate fi de 2 feluri ‘Saving Account’ si ‘Spending Account’

-o clasa ‘Spending Account’ care va implementa atat metodele clasei pe care o mosteneste, ‘Account’ , cat si metode specifice acestei clase.

-o clasa ‘Saving Account’ asemanatoare celei de mai sus, singura diferenta fiind faptul ca aceasta va tine cont de o anumita dobanda introdusa de utilizator atunci cand va face depuneri si retrageri de bani din acest tip de cont

-o clasa ‘Person’ care va modela tipul clientului bancii

Pentru a realiza aceasta aplicatie va trebui folosit ‘Design by contract’ , adica in interfata vor trebui specificate conditii pentru datele de intrare si pentru valoarea returnata, conditii care vor fi mai apoi implementate in clasa ‘Bank’.

In detaliu, Design by contract are urmtoarele semnificatii:

- reprezinta un "contract" care specifica restrictiile la care trebuie sa se supuna datele de intrare ale unei metode, valorile posibile de iesire si starile in care se poate afla programul
- aceste restrictii sunt date sub forma unor:
    a) preconditii: reprezinta obligatiile 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

Pentru a nu pierde lista de conturi din banca si avand in vedere ca nu putem folosi o baza de date, trebuie sa folosim serializarea. In forma cea mai simpla serializarea obiectelor inseamna salvarea si restaurarea starii obiectelor. Obiectele oricarei clase care implementeaza interfata *Serializable*, pot fi salvate intr-un stream(fluxde date) si restaurate din

acesta. Pachetul java.io contine doua clase speciale *ObjectInputStream* *r*espectiv *ObjectOutputStream* pentru serializarea tipurilor primitive. Subclasele claselor serializabile vor fi si ele serializabile. Mecanismul de serializare salveaza variabilele membri nestatice si netransiente. Daca un obiect este serializat, atunci orice alt obiect care contine o referinta la obiectul serializat va sisi el insusi serializat. Se poate serializa orice multime de obiecte interconectateintr-o structura de graf.

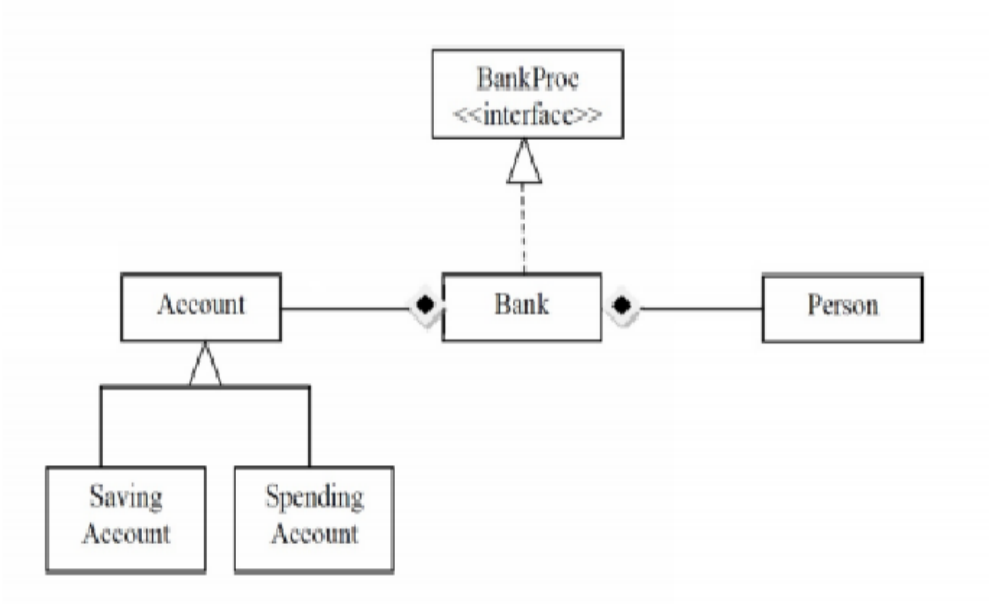
Pentru a vedea tranzactiile care se fac in fiecare cont va trebui implementata o clasa de tip Observer.

Putem observa urmatoarele scenarii de utilizare:

-Utilizatorul tine in aceasta aplicatie o lista de conturi pe care le poate manipula in functie de dorintele clientului .

-Clientul foloseste aplicatia el insusi, cu anumite restrictii, el putand observa soldul contului sau cu ajutorul clasei Observer.

3.Proiectare



Dupa cum se specifica in documentatie, cerinta acestei teme este implementarea acestei diagrame UML. Dupa modelul diagramei UML am definit corect urmatoarele clase.

-Interfata BankProc

Aceasta interfata defineste metode pentru operatiile specifice unei banci: addAccount(Account a), deleteAccount(Account a), getAccount(int n) care lucreaza doar cu Hashtable-ul din clasa Bank. Pe langa aceste definitii de metode, avem definite si preconditiile si postconditiile. De exemplu pentru metoda addAccount o preconditie este ca ‘a’ sa nu fie ‘null’.

-Clasa Bank

Aceasta clasa implementeaza interfata BankProc si evident metodele acesteia. Pentru a stoca lista de conturi specifice unei banci am folosit structura de date ‘Hastable<Key, Object>’ care mapeaza un obiect intr-o lista in functie de o cheie de tip intreg. In cazul nostru aceasta cheie va fi un id unic introdus de catre client. Pe langa aceste metode, clasa Bank implementeaza si conditiile definite in interfata cu ajutorul instructiunii ‘assert’. De exemplu , pentru conditia explicitata mai sus, instructiunea de tip assert va fi “assert a != null : “un mesaj”; “.

-Clasa Person

Este o clasa simpla, scopul ei fiind sa modeleze un posibil client al bancii. Aceasta clasa este descrisa de un nume, o adresa de email, si de varsta.Singurele metode definite de aceasta clasa sunt cele de get si set.

-Clasa abstracta Account

Aceasta clasa modeleaza un cont generic, ea fiind descrisa de un id, o persoana, un tip de cont si o suma.

Nici aceasta clasa nu are metode specifice, in afara de cele de get si set.

-Clasa SavingAccount

Este una din cele 2 clase care mostenesc clasa Account. Pe langa variabilele si metodele mostenite, aceasta clasa este descrisa si de o dobanda. Aceasta dobanda este folosita pentru a calcula suma pe care clientul o retrage dupa un anumit timp. Ca o alta particularitate a acestei clase, putem observa ca permite depunerea si retragerea de bani dintr-un cont o singura data.

-Clasa SpendingAccount

Este a 2 a clasa care mosteneste clasa Account. Aceasta clasa nu are nimic special, retragerile si depunerile de bani fii facute fara restrictii, dar la suma depusa nu se mai aduna in timp dobanda.

-Clasa Serializare

In aceasta clasa sunt definite cele 2 metode pentru a putea memora datele generate de aceasta aplicatie :

-serializare

In aceasta metoda se declara un fisier de tip FileOutputStream si un obiect de tip ObjectOutputStream. Constructorul acestui tip de fisier primeste ca argument un cale din sistem la care se afla fisierul cu extensia .ser. Daca fisierul nu exista, il va crea. In variabila obiect voi “scrie” obiectul de tip Bank transmis ca si parametru, iar apoi serializarea este incheiata.

-deserializare

Aceasta metoda primeste ca argument doar calea fisierului de tip .ser si declara aceleasi variabile ca in metoda de mai sus. Din acest fisier este citit un obiect iar apoi cu ajutorul unui 'cast' il vom scrie intr-o variabila de tip Bank pe care o vom si returna.

-Clasa Observer

Aceasta clasa nu face nimic altceva decat sa afiseze un mesaj in consola atunci cand un utilizator face operatii pe un cont. Prin operatii se intelege adaugarea unei sume de bani sau scoaterea unei sume de bani.

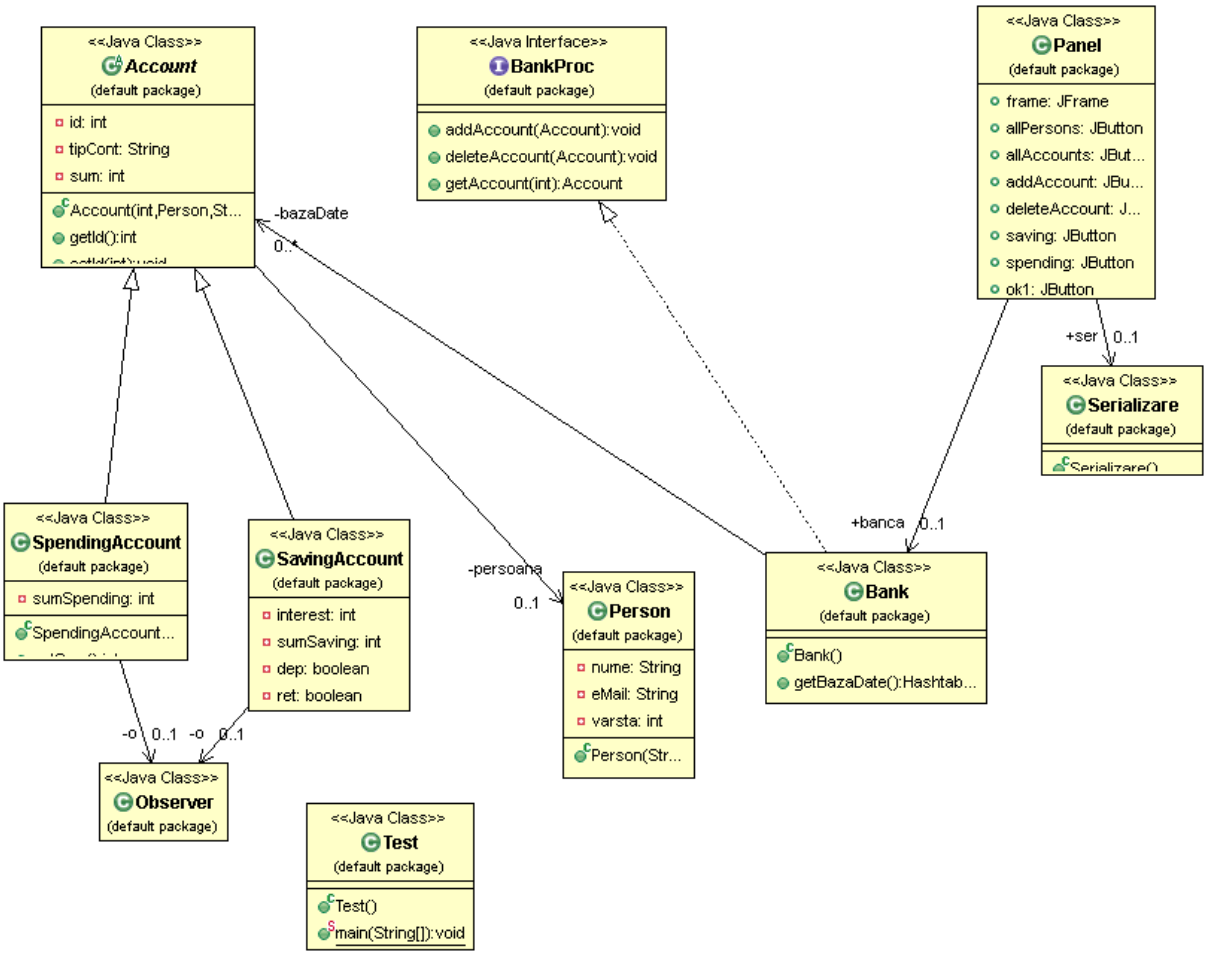
Pentru interfata cu utilizatorul m-am decis sa folosesc un design simplu si usor de inteles astfel ca oricine stie sa citeasca poate folosi aceasta aplicatie. Diversele operatii pe care pot fi efectuate in cadrul unei banci si asupra unui cont sunt implementate cu ajutorul unor butoane etichetate sugestiv, de exemplu petru a afisa un tabel cu lista tuturor conturilor din banca utilizatorul nu trebuie decat sa apese pe buton pe care scrie "Toate conturile". Pentru operatii care necesita mai multe decizi, interfata este actualizata in functie de fiecare caz. De exemplu, pentru a introduce un cont nou, utilizatorul va trebui sa apese pe butonul "Adaugare cont" , dupa care va trebui sa introduca un id pentru acest cont si datele clientului. Apoi va trebui sa aleaga intre un "Saving Account" sau "Spending Account" tot cu ajutorul unor butoane. Dupa ce a selectat una dintre optiuni, utilizatorul va trebui sa mai introduca suma de bani cu care se deschide contul, iar daca acesta a ales "Saving Account " si dobanda aferenta.

4. Implementare si testare

Pentru testare am folosit JUnit pentru a face testarea unitara. Unit testing-ul s-a impus în ultima perioadă în dezvoltarea proiectelor scrise în limbajul Java şi nu numai, pe măsura apariţiei unor utilitare gratuite de testare a claselor, care au contribuit la creşterea vitezei de programare şi la micşorarea semnificativă a numărului de bug-uri.

Printre avantajele folosirii framework-ului JUnit se numără:

- îmbunătăţirea vitezei de scriere a codului şi creşterea calităţii acestuia
- clasele de test sunt uşor de scris şi modificat pe măsură ce codul sursă se măreşte, putând fi compilate împreună cu codul sursă al proiectului
- clasele de test JUnit pot fi rulate automat (în suită), rezultatele fiind vizibile imediat
- clasele de test măresc încrederea programatorului în codul sursă scris şi îi permit să urmărească mai uşor cerinţele de implementare ale proiectului
- testele pot fi scrise înaintea implementării, fapt ce garantează înţelegerea funcţionalităţii de către dezvoltator



5.Rezultate

Ca rezultat am obtinut o aplicatie care gestioneaza conturile dintr-o banca si ofera pentru efectuarea acestor operatii o interfata grafica simpla si usor de folosit. Corectitudinea operatiilor efectuate in aplicatie este asigurata de catre preconditiile si postconditiile declarate in interfata BankProc si implementate in clasa Bank. Totodata mai este asigurata si de catre testarea aplicatiei cu ajutorul framework-urlui JUnit. Este o aplicatie buna pentru un utilizator “autorizat” cat si pentru un client deoarece, cu ajutorul clasei Observer, clientul poate urmari orice tranzactie este efectuata asupra datelor din contul acestuia.

6. Concluzii

In concluzie, pot spune ca facand aceasta tema am invatat despre importanta testarii unei aplicatii, mai ales in cazul unei aplicatii de genul acesta in care se simuleaza functionarea unei banci. Cred ca testarea metodelor si a corectitudinii transmiterii datelor intre acestea este cu atat mai importanta cu cat numarul de date stocate in aplicatie creste, deoarece daca apare o

eroare sau un bug, programatorului ii este tot mai greu sa urmareasca un flux de date intr-o crestere rapida. Astfel ca, de exemplu, testarea unitara este o unealta foarte puternica pentru depanarea aplicatiilor. Totodata am invatat si un mod eficient de stocare a datelor, respectiv structura de date Hashtable. “An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.”. Dupa cum putem afla si din definita data de documentatia Oracle, aceasta strucura de date “mapeaza” un obiect sau o valoare dupa o cheie unica. Functia dupa care este mapat obiectul poate fi liniara sau cuadratica, asta depinzand de programator. In concluzie, aceasta tema a fost o modalitate foarte buna de a imi dezvolta in continuare tehnicile deprogramare in limbajul Java.