2015

Bank Management

Tehnici de programare – Tema 4

Bolba Raluca Maria, grupa 30225

Universitatea Tehnica din Cluj-Napoca

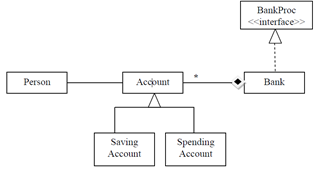
Cuprins

1. Obiectivul temei ................................................................................................................2
2. Analiza problemei, modelare, scenarii, cazuri de utilizare ................................................3
3. Proiectare (diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator) ...........................................................................4
4. Implementare și testare.....................................................................................................9
5. Rezultate ..........................................................................................................................13
6. Concluzii, dezvoltări ulterioare ........................................................................................14
7. Bibliografie .......................................................................................................................14
8. Obiectivul temei

*Enunț :*

Utilizand schema din figura, sa se implementeze un sistem de gestiune a unei banci.

1. Definiti interfata BankProc(add/remove account, read/write account, report). Specifica preconditiile si postconditiile metodelor interfetei.
2. Implementati clasele Person, Account, SavingAccount, SpendingAccount. Se pot adauga si alte clase de care aveti nevoie (argumentati motivele pentru care le-ati adaugat).
3. Implementati clasa Bank utilizand tabel de dispersie. Pentru coliziuni se va folosi inlantuierea. De asemenea, pentru clasa Bank se va defini o metoda toString.
   1. Definiti o metoda de tip “well formed” pentru clasa Bank.
   2. Implementati clasa utilizand metoda “Design by Contract”(implicand pre, post conditii, invarianti, asertii).
4. Implementati drivere de test pentru sistem. Datele initiale pentru popularea obiectului Bank cu obiecte Account se vor lua dintr-un fisier.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Sistemul ar trebui să modeleze un sistem de gestiune a unei bănci, pentru care o persoană poate să-și creeze atât un cont de economii, cât și unul de cheltuieli care să permit tranzacționarea cât mai eficientă și sigură a banilor. Persoana ar trebui să poată efectua operații precum interogarea soldurilor, adăugarea de bani în cont, retragere din cont dar și închidere a conturilor. De asemenea sistemul ar trebui să permit vizualizarea tuturor conturilor și starea acestora, precum si posibilitatea de a căuta o persoană sau un cont din baza de date.

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

Pentru o bună implementare a sistemului de gestiune a băncii, trebuie mai întâi să definim conceptele folosite, să stabilim datele de intrare și cum sunt ele furnizate și procesate pentru a fi folosite în cadrul sistemului pentru a rezulta date de ieșire corecte.

Conturile bancare utilizate în cadrul acestui sistem sunt de două tipuri : cont de economii și cont de cheltuieli. Contul de economii este definit ca unul pe care o persoană îl folosește pentru a-și depozita bani pe o perioadă mai lungă de timp. Conturile de economii beneficiază de o dobândă, care se adaugă la soldul curent de fiecare data când persoana realizează o depunere de bani în cont. Dobânda este definită în cazul nostru ca fiind 10% din suma pe care o depune persoana în cont. De asemenea, persoana nu poate retrage din cont o sumă mai mare decât cea existentă în cont. În cazul contului de cheltuieli, deținătorul contului nu mai beneficiază de dobândă, datorită faptului ca frecvența retragerilor și depunerilor în cont este mai mare decât în cazul celuilalt cont. În schimb, cu un cont de cheltuieli, persoana poate cheltui o sumă mai mare decât cea din cont, până la o anumită limită impusă de bancă.

În cazul aplicației noastre, o persoană poate să își creeze cel mult două conturi la bancă, respective unul de economii și unul de cheltuieli. Pe baza codului numeric personal, el poate să își vizualizeze conturile, și să facă anumite operații pee le, respectiv : depunere în cont, retragere din cont sau închidere cont. Așadar o parte din datele de intrare reprezintă informații personale ale clientului.

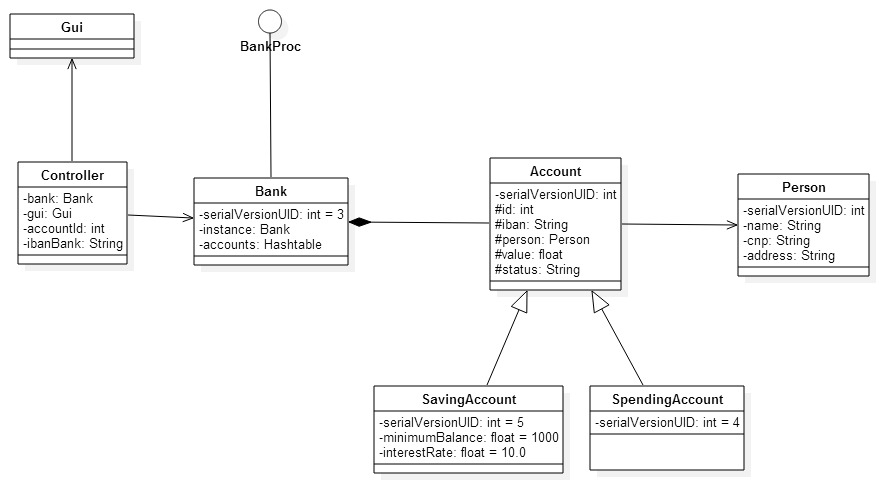
Cum am mai spus, sistemul trebuie să ofere posibilitatea management-ului băncii, respective vizualizarea conturilor, căutarea unui cont sau a unei persoane sau închiderea unui cont.

Interfața grafică cu utilizatorul trebuie să fie una simplă, intuitivă și care să permită utilizatorului să observe operațiile descrise mai sus.

1. Proiectare (diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator)
   1. Proiectare clase

Pentru implementarea acestei probleme am considerat necesare următoarele clase : *BankProc , Bank, Account, Person, SavingAccount, SpendingAccount, Gui, Controller* și *Main.*

* Interfața *BankProc* reprezintă șablonul sau contractul pe care o bancă trebuie să îl respecte. Aceasta are definite metodele pe care clasele ce o implementează sunt constrânse să le definească.
* Clasa *Bank* modelează, după cum spune și numele, conceptul de bancă și implementează interfața *BankProc* și *Serializable*. Clasa este implementată pe baza design pattern-ului *Singleton*, conform căruia clasa este responsabilă de creerea unui obiect în timp ce se asigură ca doar acel obiect este creat. Datorită acestui lucru constructorul clasei este unul privat, și care nu face nimic. Clasă oferă de asemenea o cale de acces la obiectul creat printr-o metodă statică, fără necesitatea instanțierii unui obiect al clasei. Clase este așadar definită printr-o instanță a propriei clase, printr-un tabel de dispersie care mapează liste de conturi la un anumit cod de dispersie și un id util pentru serializarea clasei.
* Clasa *Account* modelează conceptul de cont bancar, este o clasă abstractă și implementează de asemenea interfața *Serializable.* Aceasta este definită de asemenea de un id folosit pentru serializare, de un id util pentru determinarea codului de dispersie, de un cod unic iban, de un status care poate fi *active/inactive*, de un sold curent al contului și de asemenea de un obiect de tip *Person*, reprezentând deținătorul contului.
* Clasa *Person* , după cum sugerează și numele, modelează conceptul de persoană care deține unul sau două conturi bancare. Clasa implementează și ea interfața *Serializable,* contribuind la serializarea obiectelor bancare, așadar unul dintre atributele clasei îl reprezintă id-ul de serializare. Restul atributelor sunt : numele persoanei, codul numeric personal al persoanei și de asemenea adresa acesteia. Obiectele acestei clase sunt instanțiate în constructorul cu același nume, care primește ca parametrii numele, cnp-ul și adresa unei persoane.
* Clasa *SavingAccount* modelează conceptul de cont de economii. Clasa implementează și ea interfața *Serializable,* contribuind la serializarea obiectelor bancare, așadar unul dintre atributele clasei îl reprezintă id-ul de serializare. Clasa extinde clasa abstractă *Account,* suprascriind o parte din metodele acesteia. Pe lângă atributele moștenite de la clasa părinte, aceasta mai are definite încă două atribute : respectiv dobânda și suma minimă care trebuie să se afle în cont pentru ca acesta să fie activ. Obiectele acestei clase sunt instanțiate în constructorul cu același nume, care primește ca parametrii id-ul contului, iban-ul contului, persoana care îl deține și valoarea inițială.
* Clasa *SpendingAccount* modelează conceptul de cont de cont de cheltuieli. Clasa implementează și ea interfața *Serializable,* contribuind la serializarea obiectelor bancare, așadar unul dintre atributele clasei îl reprezintă id-ul de serializare. Clasa extinde clasa abstractă *Account,* suprascriind o parte din metodele acesteia. Obiectele acestei clase sunt instanțiate în constructorul cu același nume, care primește ca parametrii id-ul contului, iban-ul contului, persoana care îl deține și valoarea inițială.
* Clasa *Gui* modelează conceptul de interfață grafică cu utilizatorul prin care un utilizator poate interacționa cu aplicația propriu-zisă. Ca bibliotecă de dezvoltare a interfeței grafice am folosit *Swing*.
* Clasa *Controller* modelează conceptual de controller care face legătură între partea grafică a sistemului și modelul acestuia. Printre atributele acestei clase se numără un obiect de tip *Gui* și instanța clasei *Bank*, între care se fac principalele legături. Pe lângă acestea, clasa mai prezintă două atribute statice, respectiv un *id* și un *iban* utile la crearea unor conturi noi, pentru a asigura unicitatea iban-ului pentru conturi, și încă un atribut reprezentând un obiect de tip *Person*, care reține persoana care la un moment dat efectuează operații asupra conturilor bancare personale.
* Clasa *Main* este clasa în care se pornește aplicația, prin instanțierea unor obiecte de tip *Gui* și *Controller.*
  1. Diagrama UML



* 1. Relațiile între clase

Pe baza diagramei UML se observă relațiile între clase.

Între clasele *SavingAccount*, *SpendingAccount* și *Account*  se observă relația de de generalizare, sau de moștenire simplă, conform căreia primele două clase reprezintă clase derivate sau subclase ale clasei *Account*, definită în acest context clasă părinte sau superclasă.

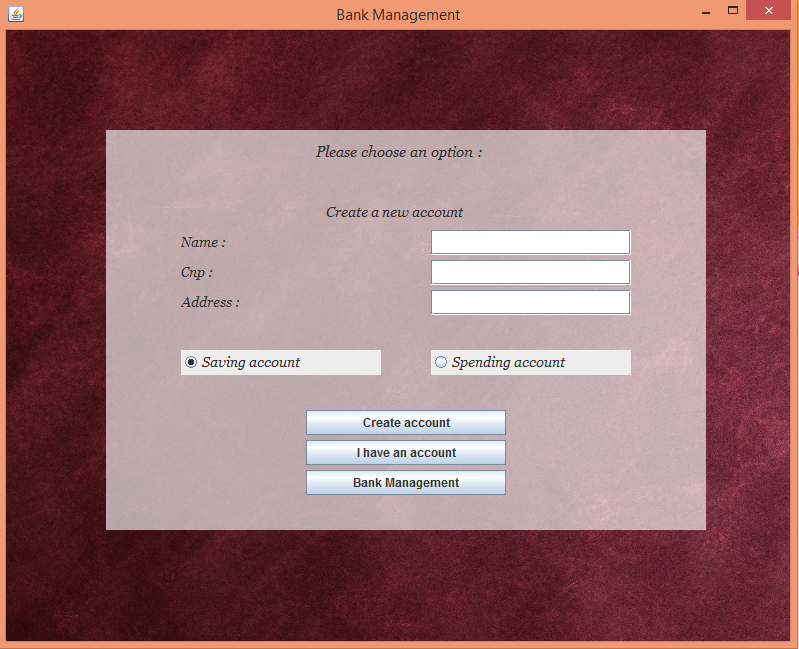
Între clasa *Account* și *Person*, *Controller și Bank* și de asemenea între *Controller și Bank* se observă relație de asociere care reprezintă cazul în care un obiect dintr-o clasăfolosește serviciile unui obiect al celeilalte clase.

Între clasa *Account* și *Bank* se observă relația de agregare, care reprezintă o relație specială de asociere ce modelează relația întreg – parte între un agregat(întreg) și părțile sale . Agregatul reprezintă în cazul nostru clasa *Bank,* iar partea clasa *Account .*

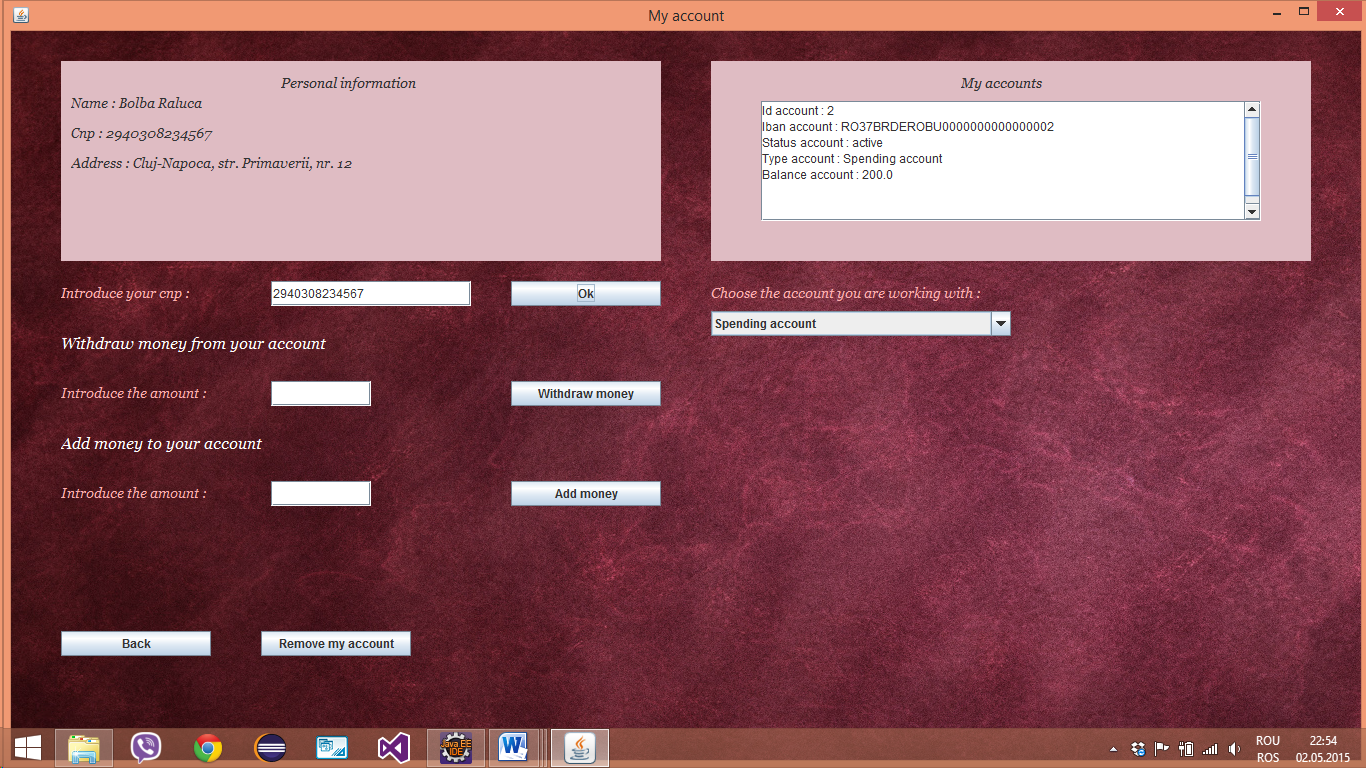
* 1. Interfața grafică

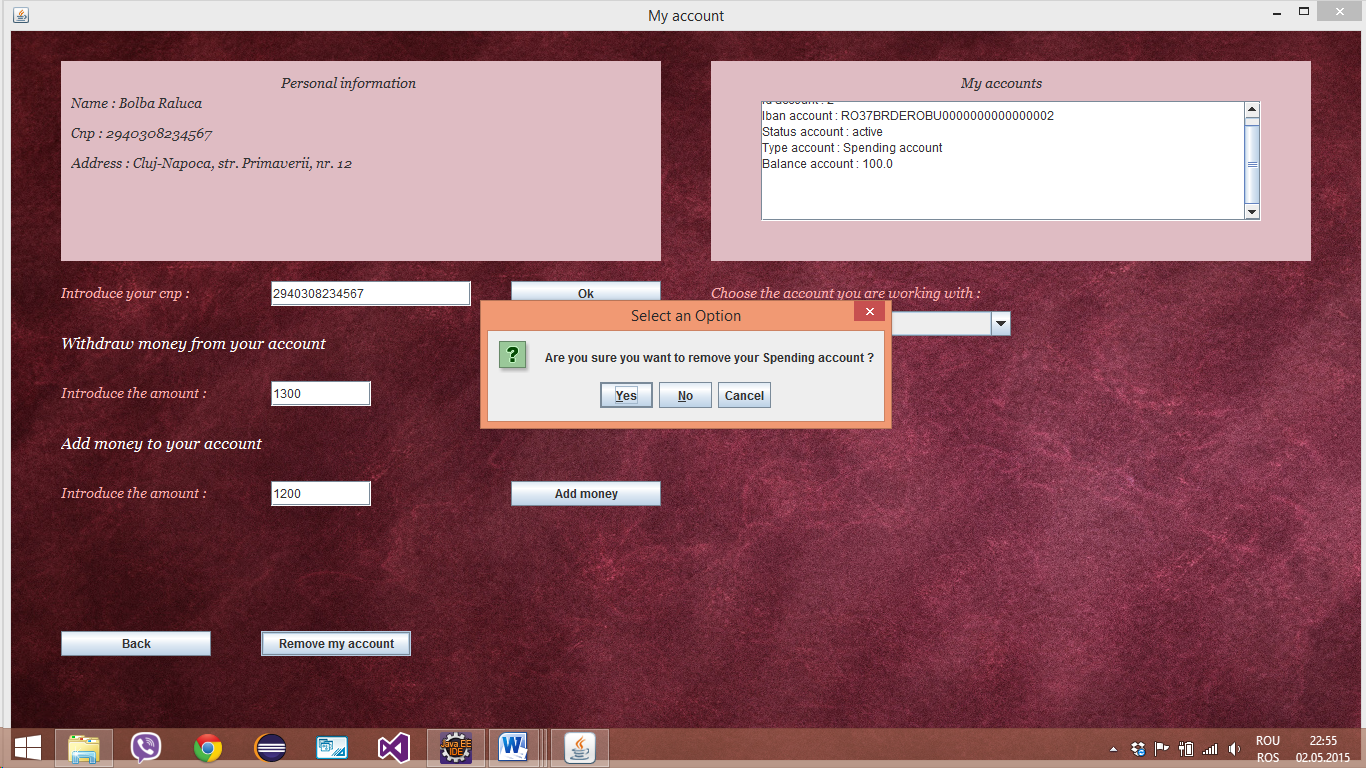
Așa cum am mai menționat, interfața grafică cu utilizatorul am implementat-o cu ajutorul bibliotecii Swing. Principalele clase folosite în cadrul realizării acestei interfețe sunt : J*Button, JComboBox, JLabel, JTextField, JFrame , JdesktopPane, JPasswordField etc.*

Fereastra de început a aplicației permite utilizatorului să aleagă ce dorește să facă mai departe, respectiv, să își creeze un cont bancar prin completarea câmpurilor cu datele personale și specificarea tipului contului și apăsarea butonului *Create account*, poate de asemenea să meargă la o altă fereastră unde va putea efectua operații asupra conturilor personale prin apăsarea butonului *I have an account*, sau poate să vizualizeze conturile din baza de date prin apăsarea butonului *Bank Management.*

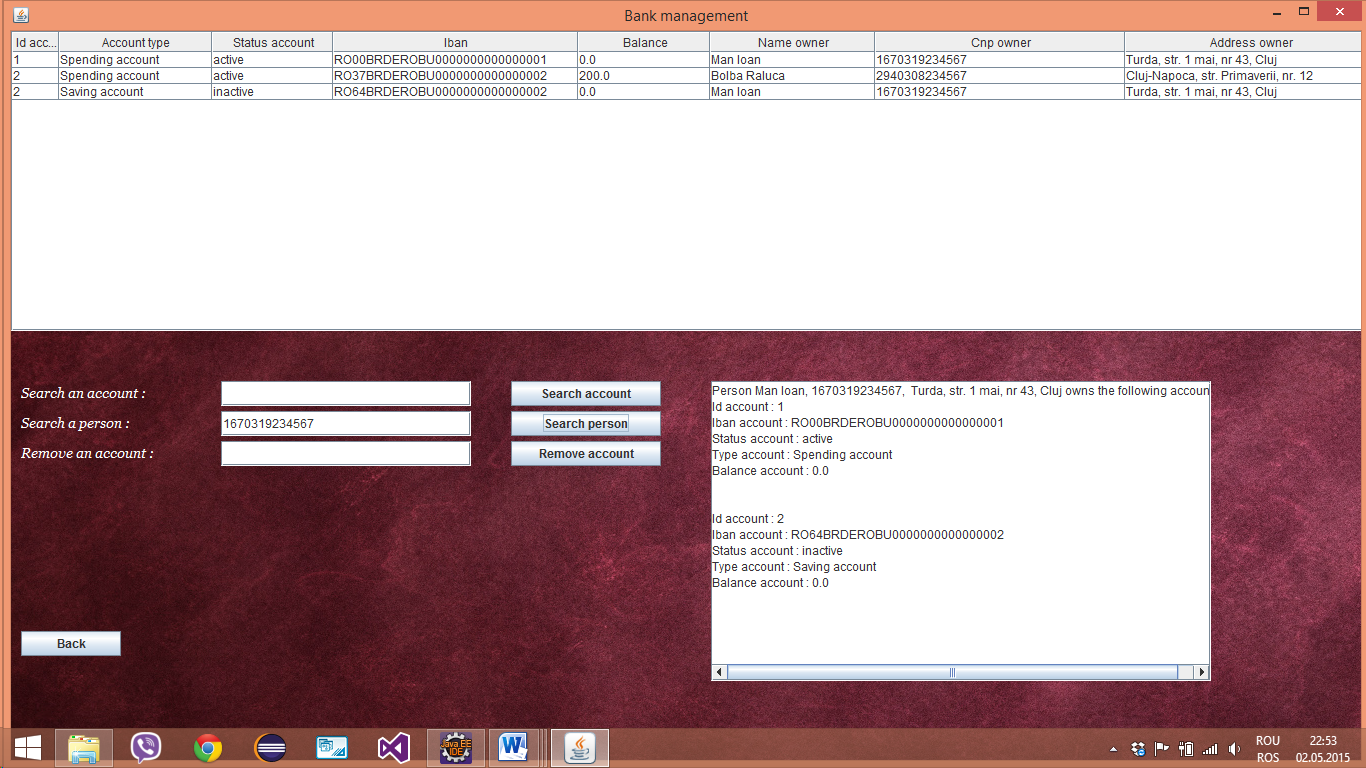
**

În cazul în care acesta și-a creat un nou cont bancar sau are unul existent și apasă butonul al doilea, el va fi redirecționat către altă fereastră. În cadrul acesteia, el va putea să își vizualizeze conturile și datele personale prin introducerea codului numeric personal și apăsarea butonului *Ok*. În cazul în care acesta deține două conturi, în partea dreaptă se observă un câmp de tip *JcomboBox*, prin care poate alege asupra cărui cont poate efectua depuneri sau retrageri.

Dacă persoana dorește să retragă o anumită sumă de bani, ea trebuie să o introducă în câmpul special alocat pentru acest lucru și să apese butonul *Withdraw money*, iar în cazul în care suma introdusă a fost corectă, va fi confirmată retragerea. În mod similar acesta trebuie să procedeze și dacă dorește să adauge o sumă de bani la unul din conturi. În cazul în care acesta dorește să închidă un cont, trebuie să apese butonul *Remove account*, și înainte de a închide contul, utilizatorului i-i se cere să confirme închiderea.



În cazul în care utilizatorul a apăsat al treilea buton din pagina de start a aplicației, el va fi redirecționat către următoarea fereastră :



În cadrul acesteia, se poate observa în partea de sus a ferestrei un tabel ce conține conturile existente ale băncii, iar în partea de jos a ferestrei se observă câmpuri și butoane care permit efectuarea unor operații asupra conturilor și persoanelor deținătoare de conturi precum : căutarea unei persoane, căutarea unui cont sau închiderea unui cont.

1. Implementare și testare
   1. Implementare

* Interfața *BankProc* modeleză conceptul de contract pe care clasele care o implementează trebuie să îl respecte. Metodele abstracte ale acesteia sunt :
  + **public void** addAccount(Account account)
  + **public** **boolean** removeAccount(Account account)
  + **public** Hashtable<Integer, List<Account>> readAccounts() **throws** IOException
  + **public** **void** writeAccounts(Hashtable<Integer, List<Account>> accounts)
* Clasa *Bank* modelează conceptul de bancă. Metodele acestei clase sunt :
  + **public void** addAccount(Account account) – metodă care adaugă un cont la tabela de dispersie a băncii. Dacă contul care se dorește să se adauge nu este null, atunci se determină codul de dispersie a contului, se verifică dacă pentru codul obținut există o listă creată în tabela de dispersie. În caz afirmativ, se adaugă contul la lista existentă, iar în caz contrar se crează o nouă listă conținând contul de adăugat, și se mapează această listă în cadrul tabelei la codul de dispersie obținut.
  + **public** **boolean** removeAccount(Account account) – metodă care șterge un cont din cadruk tabelei de dispersie. Pentru a afectua ștergerea se calculează mai întâi codul de dispersie al contului, și se verifică tabelul la poziția indicate de codul de dispersie. Dacă poziția respectivă nu este goală, atunci se parcurge lista de conturi asociată și dacă aceasta din urmă conține contul pe care vrem să îl ștergem, îl eliminăm din listă.
  + **public** Hashtable<Integer, List<Account>> readAccounts() **throws** IOException – metodă care citește și returnează tabelul de dispersie conținând conturile băncii dintr-un fișier, prin intermediul deserializării.
  + **public** **void** writeAccounts(Hashtable<Integer, List<Account>> accounts) – metodă care scrie tabelul de dispersie conținând conturile într-un fișier, prin intermediul serializării.
  + **public** **static** Bank getInstance()– metodă care returnează instant clasei *Bank*
  + **public** **void** setHash(Hashtable<Integer, List<Account>> accounts) – metodă care setează tabelul de dispersie cu cel primit ca parametru
  + **public** Hashtable<Integer, List<Account>> getHash()– metodă care returnează tabelul de dispersie
  + **public** List<Account> getAccounts(**int** hashcode) – metodă care returnează lista de conturi ce sunt mapate la codul de dispersie primit ca parametru
  + **public** List<Account> getAccounts(String cnp) – metodă care returnează lista de conturi ale persoanei cu cnp-ul primit ca parametru
  + **public** Account getAccount(String iban) – metodă care returnează contul din cadrul tabelei de dispersie ce are iban-ul egal cu cel primit ca parametru
  + **public** String toString()– metodă care generează și returnează un șir de caractere reprezentând eticheta clasei
  + **public** **boolean** isWellFormed()– metodă care reprezintă invariantul clasei *Bank*. Această metodă verifică dacă clasa este bine formată, verificând tabelul de dispersie.
* Clasa *Account* modelează conceptul de cont bancar. Metodele acestei clase sunt :
  + **public** **int** getId()– metodă care returnează id-ul contului
  + **public** String getIban()– metodă care returnează iban-ul contului
  + **public** String getStatus()– metodă care returnează statusul contului
  + **public** Person getOwner()– metodă care returnează un obiect de tip Person reprezentând deținătorul contului
  + **public** **float** getValue()– metodă care returnează soldul curent al contului
  + **public** **void** setValue(**float** newValue) – metodă care setează soldul curent cu o valoare nouă primită ca parametru
  + **public** **void** activateAccount()– metodă care setează statusul contului la *active*
  + **public** **abstract** **void** withdrawMoney(**float** amountMoney) **throws** Exception – metodă abstractă care retrage o sumă de bani din cont
  + **public** **void** addMoney(**float** amountToAdd) – metodă care adaugă o sumă de bani la soldul curent al contului
  + **public** **int** hashCode()– metodă care generează și returnează un cod de dispersie pentru cont
  + **public** **boolean** equals(Object a) – metodă care verifică dacă instanța curentă a clasei este egală cu obiectul primit ca parametru
  + **public** String getType()– metodă care returnează tipul contului
* Clasa *SavingAccount* modelează conceptul de cont de economii. Metodele acestei clase sunt :
  + **public** **void** activateAccount()– metodă care activează contul de economii dacă acesta este inactiv și dacă soldul curent depășește suma minima de 1000 lei
  + **public** **void** deactivateAccount()– metodă care dezactivează contul de economii dacă acesta este active și dacă soldul curent este mai mic decât minumul de 1000 lei
  + **public** **void** addMoney(**float** amountToAdd) – metodă care depune o sumă de bani la soldul curent. La suma adăugată se adaugă și o dobândă, reprezentând 10% din suma adăugată.
  + **public** **void** withdrawMoney(**float** amountMoney) **throws** Exception – metodă care retrage o sumă de bani din contul de economii. Suma de bani retrasă nu poate fi mai mare decât soldul curent al contului.
  + **public** String getType()– metodă care returnează tipul contului, respectiv *Saving account*
* Clasa *SpendingAccount* modelează conceptul de cont de cheltuieli. Metodele acestei clase sunt :
  + **public** **void** withdrawMoney(**float** amountMoney) **throws** Exception– metodă care retrage o sumă de bani din contul de cheltuieli. Suma de bani retrasă poate fi mai mare decât soldul curent al contului cu cel mult 2000 lei.
  + **public** String getType()– metodă care returnează tipul contului, respectiv *Spending account*
* Clasa *Person* modelează conceptul de persoană care deține unul sau două conturi bancare. Metodele acestei clase sunt :
  + **public** String getCnp()– metodă care returnează codul numeric personal al personei
  + **public** String getName()– metodă care returnează numele persoanei
  + **public** String getAddress()– metodă care returnează adresa persoanei
  + **public** String toString()– metodă care generează și returnează o reprezentare sub formă de șir de caractere a persoanei
  + **public** **boolean** equals(Object obj) – metodă care verifică dacă instanța curentă a clasei este aceași cu obiectul primit ca parametru. Comparația se face pe baza codului numeric personal
* Clasa *Gui* modelează conceptul de interfață grafică cu utilizatorul. Printre metodele acestei clase se numără :
  + **public** **void** bankManagement()– metodă care redirecționează utilizatorul la fereastra corespunzătoare butonului *Bank Management*
  + **public** **void** MyAccount()– metodă care redirecționează utilizatorul la fereastra în cadrul căreia poate efectua operații asupra propriilor conturi bancare
* Clasa *Controller* modelează conceptul dintre controller. Printre metodele acestei clase se numără :
  + **public** String getAccountInformation(List<Account> list) – metodă care generează și returnează un șir de caractere conținând informații despre conturile transmise sub forma de listă ca parametru
  + **public** **void** addAccountsToModel(DefaultTableModel model) – metodă care adaugă toate conturile existente în clasa *Bank*, mai exact din tabela de dispersie a acestei clase, la componenta de tip *DefaultTableModel*, utilizată în cadrul interfeței grafice pentru vizualizarea conturilor sub forma unui tabel
  1. Testare

În cadrul acestui proiect, testarea a constat prin specificarea și verificarea pentru fiecare metodă a unor precondiții, postcondiții și de asemenea a unui invariant pentru clasa *Bank*.

Specificarea precondițiilor, postcondițiilor și invariantului s-a făcut la începutul fiecărei metode, într-un comentariu bloc. Verificarea acestora s-a făcut în cadrul metodei, prin intermediul aserțiunilor. Precondițiile au fost verificate la începutul metodei, iar postcondițiile și invariantul la sfârșitul acesteia, înainte de *return* .

1. Rezultate

În urma rulării aplicației se observă că aceasta rulează fără probleme, crearea conturilor de către utilizator se face fără problem, la fel și operațiile de retragere sau depunere numerar, precum și restul operațiilor, rezultatul acestora observându-se imediat. Se mai observă că atunci când utilizatorul introduce date de intrare invalide acesta este atenționat cu mesaje de eroare corespunzătoare.

1. Concluzii

În concluzie, se poate spune că am reușit implementarea unui sistem de gestiune a unei bănci. În cadrul acestei teme am reușit să acumulez cunoștințe în ceea ce privește serializarea, cât și lucrul cu tabele de dispersie .

Dezvoltari ulterioare:

* Tipărirea pe chitanță a operațiilor efectuate pe cont
* La ștergerea unui cont, să apară un mesaj cu suma care trebuie eliberată respectivei persoane
* Dobânda sa fie calculată în funcție de data la care a fost creat contul și a soldului curent
* Modificarea interfeței grafice astfel încât partea de modificare a conturilor să simuleze un ATM

1. Bibliografie

<https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html>

<http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>