**Layis Norbert, 30237**

**Procesarea comenzilor**

**-Tema 3-**

**-Documentatie-**

**Cuprins**

1. Obiectivul temei . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare . . . . 4
3. Proiectare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .6
4. Implementare si testare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
5. Concluzii, dezvoltari ulterioare . . . . . . . . . . . . . . . . . . . . . . . . 22
6. Bibliografie. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .22
7. **Obiectivul temei**

Consider an application **Order Management** for processing customer orders. The application uses (minimally) the following classes: **Order**, **OPDept** (Order Processing Department), **Customer**, **Product** and **Warehouse**. The classes **OPDept** and **Warehouse** use a **BinarySearchTree** for storing orders.

**a.** Analyze the application domain, determine the structure and behavior of its classes and draw an extended UML class diagram.

**b.** Implement the application classes. Use javadoc for documenting classes.

**c.** Implement a system of utility programs for reporting such as: under-stock, over-stock, totals, filters, etc.

d. Write the appropriate test drivers

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

**Analiza problemei**

Procesarea unor comenzilor date de clienti este un concept utilizat frecvent in domeniul comertului in special, de catre administratorii si operatorii diferitelor magazine sau depozite, dar cel mai frecvent este utilizat in cazul administrarii magazinelor virtuale, un concept care a luat o deosebita amploare in ultimii ani.

Un sistem de procesare a comenzilor trebuie sa contina in primul rand o baza de date sau orice alt mod de stocare a produselor disponibile spre vanzare si a informatiilor acestora cum ar fi cantitatea sau pretul. Se poate folosi si un cod specific fiecarui produs pentru o gestionare mai usoara, pentru o gasire mai rapida a acestora. In al doilea rand trebuie sa contina comenzile date de clienti si datele acestora pentru a se putea lua legatura cu ei.

La primirea unei noi comenzi trebuie stabilit daca aceasta poate fi efectuata (produsul si cantitatea ceruta este disponibila) sau daca nu poate fi efectuata (produsul cerut nu este disponibil in stoc sau cantitatea o depaseste pe cea existenta in depozit).

**Modelare**

Pentru modelarea unei astfel de aplicatii avem nevoie de cateva elemente esentiale:

* produsele din depozit
* comenzile cerute
* modul de stocare a acestora
* date referitoare la produse, clienti si comenzi

Modelarea unui produs presupune in primul rand stabilirea unui nume pentru acesta, iar apoi a cantitatii disponibile si a pretului unitar.

Modelarea unei comenzi presupune crearea unei legaturi intre aceasta si un client, beneficiarul comenzii, stabilirea produsului cerut, a cantitatii si calcularea pretului total. In functie de acestea se poate stabili daca acesta comanda poate fi efectuata la momentul respectiv sau va trebui sa astepte pana la reincarcarea depozitului.

Modul de stocare este deosebit de important pentru acest gen de aplicatie deoarece se doreste ca la inchiderea aplicatiei produsele si comenzile sa fie salvate pentru a se putea continua procesarea acestora din punctul in care s-au aflat la momentul inchiderii aplicatiei, oferind astfel flexibilitate acesteia.

**Scenarii, cazuri de utilizare**

La deschiderea aplicatiei se incarca automat datele existente la ultima folosire a acesteia. Daca intervine o eroare la incarcare se poate efectua acest lucru si prin actionarea butonului “Import” si cautarea fisierelor sursa.

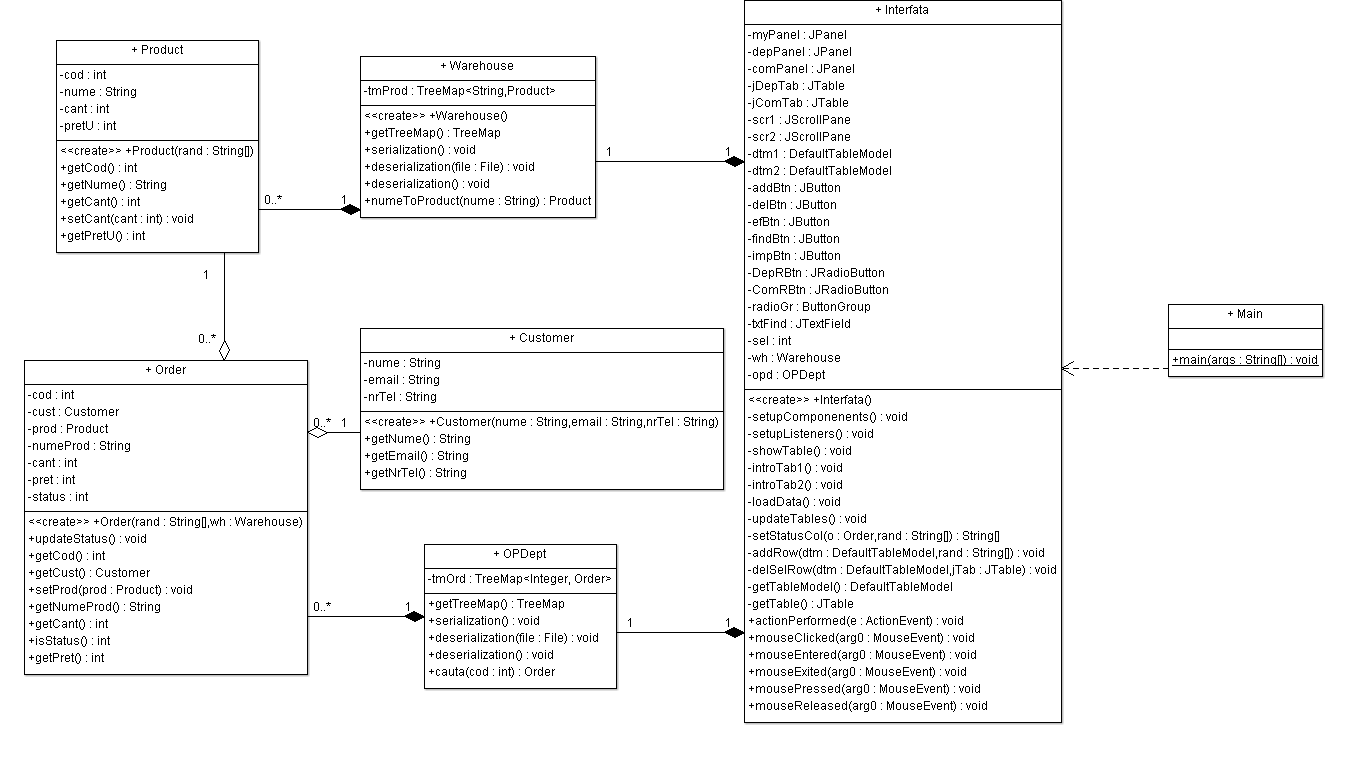
Aplicatia permite vizulaizarea a doua tabele: unul cu produsele din depozit, iar celalalt cu comenzile. In aceste tabele se pot adauga sau sterge produse sau comenzi prin actionarea butoanelor respective. In cazul adaugarii apare o fereastra pentru a completa campurile necesare produsului sau comenzii.

In tabelul de comenzi se poate determina efectuarea acestora. In cazul in care este imposibila efectuarea, utilizatorul este instiintat.

De asemenea se poate cauta in tabele prin comanda “Find” dupa o anumita secventa introdusa de utilizator.

**2. Proiectare**

**Diagrama UML**

****

**Structuri de date**

Pentru definirea unui produs am ales un cod, reprezentat prin hash code-ul obiectului, un nume de tip String, cantitatea si pretul unitar, ambele de tipul int.

Pentru definirea unei comenzi am ales deasemenea un cod, reprezentat de hash code, un client, un produs, numele acestuia (tip String), cantitatea ceruta (tip int), pretul calculat (tip int) si statusul comenzii, de tip int si care este stabilit astfel: daca este egal cu 1 comanda respectiva poate fi efectuata, daca este egal cu 0 produsul este indisponibil in stoc si daca este egal cu 2 produsul este disponibil in depozit dar cantitatea este insuficienta pentru a satisface comanda clientului. Acest status se poate schimba la introducerea de noi produse in depozit sau la marirea cantitatilor disponibile.

Clientul este la randul lui reprezentat prin nume, e-mail si numar de telefon, toate de tip String.

Pentru stocarea produselor si a comenzilor am folosit 2 arbori binari, mai precis obiecte de tipul TreeMap. Acestia fiind salvati prin serializare in fisiere pentru a se putea incarca ulterior.

**Relatii intre clase**

*Agregare:*

*-*Agregarea este relatia intre doua obiecte in care unul dintre obiecte apartine celuilalt obiect. Agregarea reda apartenta unui obiect la un alt obiect. Semantic, agregarea indica o relatie de tip "part of" ("parte din").

*-Order si OPDept*

*-Interfata si OPDept*

*-Interfata si Warehouse*

*-Warehouse si Product*

Asociere:

Asocierea este o relatie generica intre doua clase. Aceste relatii pot fi de tipurile pot defini si regulile numerice de asociere (unu la unu, unu la mai multi, mai multi la mai multi).

Intre Product si OPdept

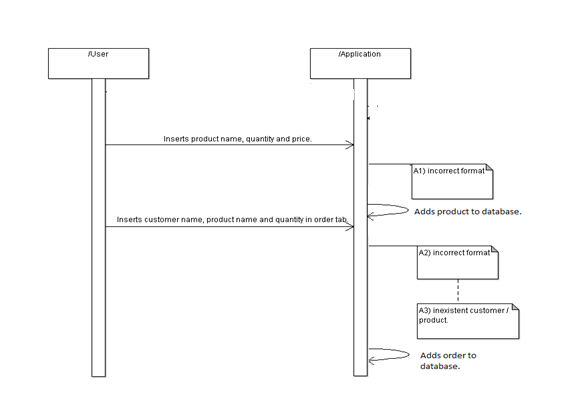
Intre Product si Customer

**Proiectare de clase**

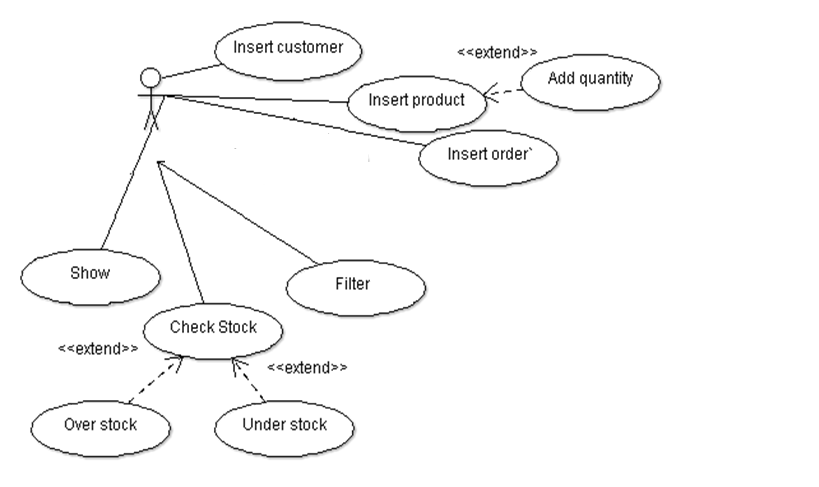
Pentru implementarea acestui proiect am folosit 7 clase: Main – clasa de unde se lanseaza in executie aplicatia, care contine doar metoda main, Customer - clasa care defineste obiectele de tip Customer, clientii, Product – clasa care defineste produsele, Order – clasa care defineste comenzile, Warehouse – clasa care simuleaza depozitul, aici stocandu-se produsele, OPDept – clasa care stocheaza comenzile si Interfata - clasa care contine si modifica daca e cazul elementele grafice (tabelele) si gestiozeaza interactiunea cu utilizatorul.

Continutul si functionalitatea claselor vor fi explicate intr-un mod mai detaliat in sectiunea *Implementare si testare.*

**DIAGRAMA DE SECVENTE**

****

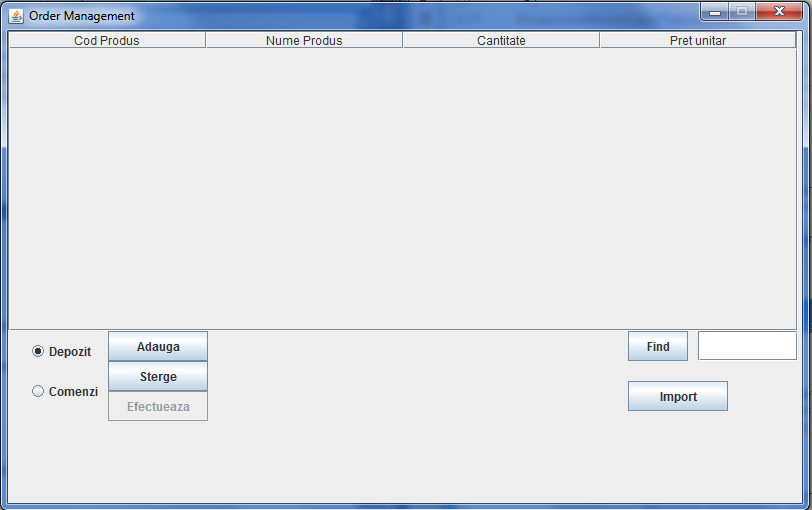
**DIAGRAMA USE CASE**

****

**Interfata utilizator**

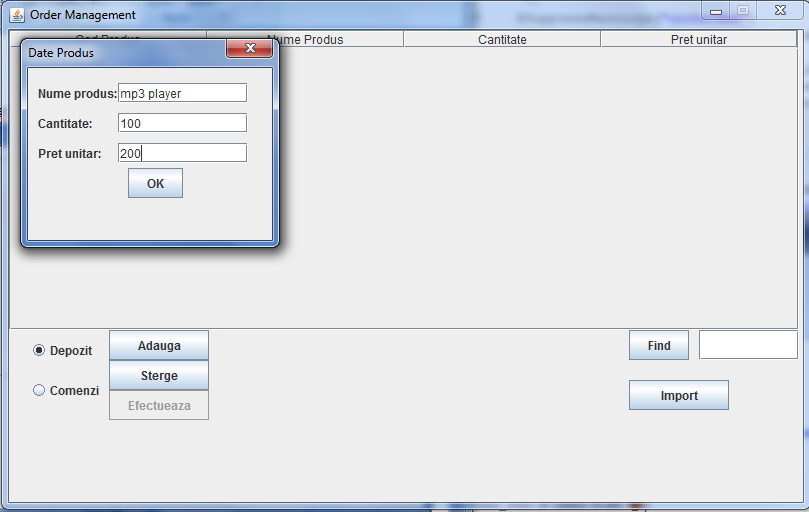
La lansarea in executie a aplicatiei interfata grafica va arata in felul urmator:

Pentru evitarea confuziilor, butonul “Efectueaza” nu este activat pentru tabelul de produse, ci doar pentru cel de comenzi, dupa cum se vede:

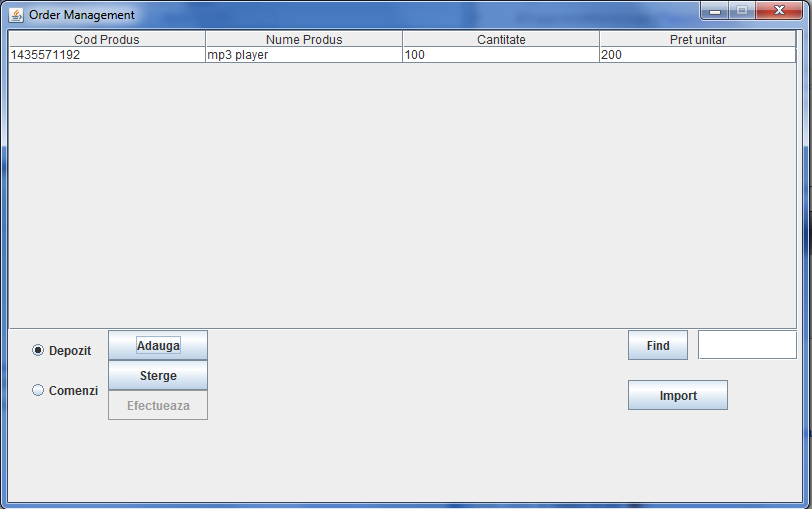
****

Pentru vizualizarea comenzilor se actioneaza butonul-radio “Comenzi”

La adaugarea unui nou produs va aparea o fereastra noua astfel:

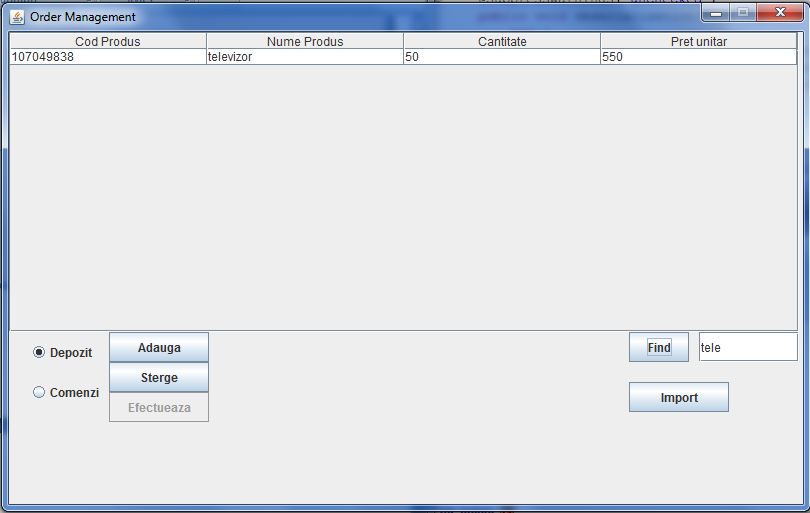


La actionarea butonului “OK” produsul va fi adaugat sau se va afisa un mesaj de eroare in cazul in care campurile nu au fost completate corect. Cantitatea si pretul unitar trebuie sa aiba valori numerice strict pozitive. Codul produsului este generat de catre program.



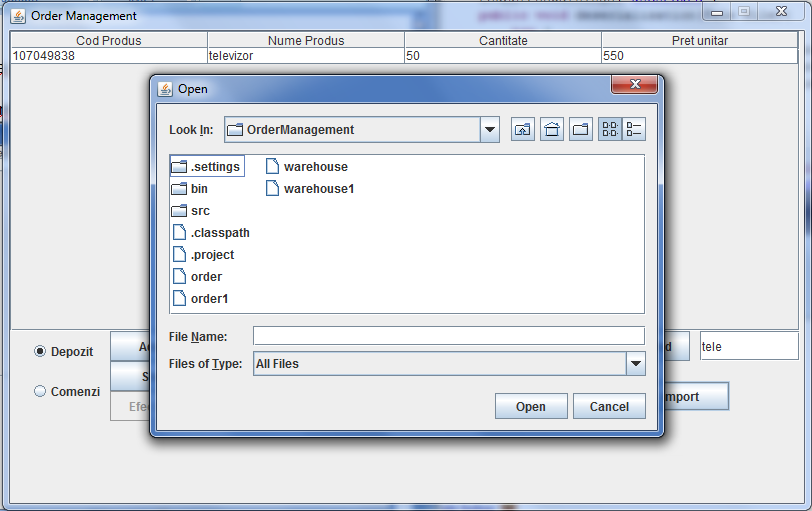
Se pot sterge unul sau mai multe produse prin selectarea randurilor respective si actionarea butonului “Sterge”.

Se poate cauta o anumita secventa in tabel cu ajutorul comenzii “Find”.

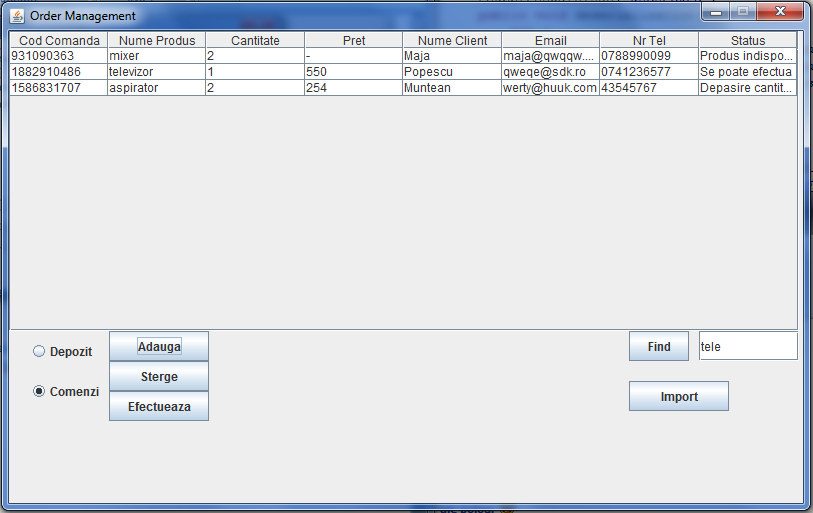


Se pot importa fisierele sursa prin actionarea butonului “Import”:

Pentru o functionare buna a programului in momentul in care este vizibil tabelul cu depozitul se va importa fisierul care contine arborele cu produse, iar apoi se va trece pe tabelul cu comenzi si se va importa fisierul corespunzator.



Tabelul cu comenzi arata in felul urmator:

 Adaugarea sau stergerea unor noi comenzi se face similar cu adaugarea sau stergerea de produse. De asemenea in cazul in care campurile nu sunt completate corect utilizatorul va fi instiintat printr-un mesaj de eroare. Cantitatea si numarul de telefon vor trebui sa aiba valori numerice strict pozitive.

Codul comenzii este generat de program, pretul este calculat in functie de pretul unitar al produsului respectiv, iar statusul este afisat in concordanta cu disponibilitatea sau indisponibilitatea produsului sau cantitatii cerute.

Comenzile al caror status este “Se poate efectua” vor putea fi efectuate prin actionarea butonului “Efectueaza”.

1. **Implementare si testare**

**Implementare**

Dupa cum s-a mentionat anterior, proiectul este structurat in 7 clase pe care le voi descrie intr-un mod detaliat in cele ce urmeaza.

* Clasa Main

Din aceasta clasa se lanseaza in executie aplicatia, creandu-se un obiect de tip Interfata. Aceasta clasa contine doar metoda **public** **static** **void** main (String[] args) .

* Clasa Customer

Aceasta clasa defineste un obiect de tip Customer, un client, avand ca atribute numele, adresa de e-mail si numarul de telefon al acestuia.

Metodele continute sunt getterele si setterele pentru atribute.

* Clasa Product

Aceasta clasa defineste un obiect de tip Product, un produs, avand ca atribute codul, numele, cantitatea disponibila si pretul unitar.

Metodele continute sunt getterele si setterele pentru atribute.

* Clasa Order

Aceasta clasa defineste un obiect de tip Order, o comanda, avand ca atribute codul, un obiect de tip Customer, un obiect de tip Product, un String care contine numele produsului, cantitatea ceruta, pretul si statusul.

Produsul cerut in comanda se cauta in TreeMap -ul care stocheaza produsele dupa numele acestuia, numele produsului fiind cheia din arbore.

Pe langa gettere si settere pentru atribute aceasta clasa contine metoda **void** updateStatus () care calculeaza pretul in functie de cantitate si pretul unitar si stabileste statusul comenzii, daca se poate efectua sau nu.

**public** **void** updateStatus(){

**if** ( prod == **null** ) {

**this**.pret = -1;

**this**.status = 0;

}

**else** {

**this**.pret = cant \* prod.getPretU();

**this**.status = 1;

}

**if** ( **this**.status == 1 && **this**.cant > prod.getCant() )

**this**.status = 2;

}

* Clasa Warehouse

Aceasta clasa reprezinta depozitul si are un singur atribut: un arbore binar, de tip TreeMap in care sunt stocate produsele. Cum am mentionat mai sus cheia pe care am ales-o in arbore este numele produsului.

Clasa contine metode pentru serializarea si deserializarea (implicita sau prin importarea fisierului) arborelui precum si o metoda pentru gasirea obiectului Product dupa numele acestuia.

**public** **void** serialization() {

**try** {

FileOutputStream fos = **new** FileOutputStream ("warehouse");

ObjectOutputStream oos = **new** ObjectOutputStream (fos);

oos.writeObject (tmProd);

oos.flush();

oos.close();

}

**catch** (Exception e) {

JOptionPane.*showMessageDialog* (**null**,

"Eroare salvare date!",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

/\*\*

\* metoda folosita pentru deserializare

\* **@param** file - fisierul sursa

\*/

@SuppressWarnings ("unchecked")

**public** **void** deserialization (File file) {

**try** {

FileInputStream fis = **new** FileInputStream(file);

ObjectInputStream ois = **new** ObjectInputStream(fis);

tmProd = (TreeMap<String, Product>) ois.readObject();

ois.close();

serialization();

}

**catch** (Exception e) {

JOptionPane.*showMessageDialog* (**null**,

"Eroare incarcare date!\nFisier sursa incorect",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

/\*\*

\* metoda folosita pt deserializare implicita

\*/

@SuppressWarnings ("unchecked")

**public** **void** deserialization(){

**try** {

FileInputStream fis = **new** FileInputStream ("warehouse");

ObjectInputStream ois = **new** ObjectInputStream(fis);

tmProd = (TreeMap<String, Product>)ois.readObject();

ois.close();

}

**catch** (Exception e) {

JOptionPane.*showMessageDialog*( **null**,

"Eroare incarcare date!\nFisierul sursa nu a fost gasit\nFolositi Import",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

/\*\*

\* metoda pt gasirea produsului dupa nume

\* **@param** nume - numele produsului

\* **@return** returneaza obiectul Product

\*/

**public** Product numeToProduct (String nume) {

Product p = tmProd.get(nume);

**return** p;

}

* Clasa OPDept

Aceasta clasa reprezinta departamentul de procesare a comenzilor si, la fel ca si clasa Warehouse, contine un singur atribut: un arbore binar, de tip TreeMap, in care sunt stocate comenzile. Aici am ales cheia ca fiind hash code -ul obiectului Order.

Clasa contine metode pentru serializarea si deserializarea (implicita sau prin importarea fisierului) arborelui precum si o metoda pentru cautarea unei comenzi dupa hash cod -ul acesteia.

**public** Order cauta (**int** cod) {

Order o;

o = tmOrd.get(cod);

**return** o;

}

Metodele pentru serializarea si deserializarea arborelui sunt similare cu cele implementate in clasa Warehouse.

* Clasa Interfata

Aceasta clasa implementeaza interfata grafica si se ocupa de interactiunea dintre utilizator si aplicatie. De asemenea aici am instantiat cate un obiect al claselor Warehouse si OPDept si ca urmare din aceasta clasa, prin apelare de metode, se efectueaza si schimbarile care intervin in cei doi arbori in urma adaugarii, stergerii sau efectuarii precum si serializarea si deserializarea acestor arbori.

Interfata contine un panel principal, denumit myPanel, si 2 paneluri: depPanel si comPanel, pentru cele 2 tabele (depozit si comenzi). Aceste 2 panel-uri nu sunt vizibile concomitent ci in functie de selectarea unuia dintre cele 2 butoane-radio, depozit si comenzi.

Contine butoane pentru adaugare, stergere, efectuare comanda, pentru importare si pentru cautare; aici se afla si un JTextField in care se va introduce secventa de cautare dorita.

Clasa contine si doua ferestre pentru introducerea datelor in cele doua tabele, dupa cum se poate vedea in continuare:

**private** **void** introTab1() {

**final** JDialog intro1 = **new** JDialog (**this**, "Date Produs");

//JLabels

JLabel lNume = **new** JLabel ("Nume produs:");

lNume.setBounds (10,10,80,30);

JLabel lCant = **new** JLabel ("Cantitate:");

lCant.setBounds (10,40,80,30);

JLabel lPret = **new** JLabel ("Pret unitar:");

lPret.setBounds (10,70,80,30);

//JTextFields

**final** JTextField tNume = **new** JTextField();

tNume.setBounds (90,15,130,20);

**final** JTextField tCant = **new** JTextField();

tCant.setBounds (90,45,130,20);

**final** JTextField tPret = **new** JTextField();

tPret.setBounds (90,75,130,20);

//Buton

JButton okBtn = **new** JButton("OK");

okBtn.setBounds (100,100,55,30);

okBtn.addActionListener (**new** ActionListener()

{

@SuppressWarnings("unchecked")

**public** **void** actionPerformed (ActionEvent e)

{

Order ord;

String[] rand = **new** String[4];

**try** {

rand[1] = tNume.getText();

rand[2] = tCant.getText();

rand[3] = tPret.getText();

**if** (Integer.*parseInt* (rand[2]) <= 0) **throw** **new** Exception();

**if** (Integer.*parseInt* (rand[3]) <= 0) **throw** **new** Exception();

Product p = **new** Product (rand);

rand[0] = Integer.*toString* (p.getCod());

addRow (getTableModel(),rand);

wh.getTreeMap().put (p.getNume(),p);

**for**(**int** i = 0; i < dtm2.getRowCount() ;i++){

**if** ((jComTab.getValueAt(i,1)).equals(rand[1])) {

ord = opd.cauta (Integer.*parseInt* ( (String) jComTab.getValueAt (i,0)));

ord.setProd(p);

ord.updateStatus();

}

}

intro1.setVisible (**false**);

wh.serialization();

opd.serialization();

updateTables();

}

**catch** (Exception e1){

JOptionPane.*showMessageDialog* (myPanel,

"Date introduse gresit!" +

"\nCantitatea si Pretul trebuie sa aiba \nvalori numerice strict pozitive",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

} );

intro1.setSize (250, 200);

intro1.setResizable (**false**);

intro1.setLayout (**null**);

intro1.setDefaultCloseOperation (*DISPOSE\_ON\_CLOSE*);

intro1.setVisible (**true**);

intro1.add (lNume);

intro1.add (lCant);

intro1.add (lPret);

intro1.add (tNume);

intro1.add (tCant);

intro1.add (tPret);

intro1.add (okBtn);

}

/\*\*

\* creaza fereastra pentru introducerea unei noi comenzi

\*/

**private** **void** introTab2(){

**final** JDialog intro2 = **new** JDialog (**this**, "Date comanda");

//JLabels

JLabel lNumeP = **new** JLabel ("Nume produs:");

lNumeP.setBounds (10,10,80,30);

JLabel lCant = **new** JLabel ("Cantitate:");

lCant.setBounds (10,40,80,30);

JLabel lNumeC = **new** JLabel ("Nume client:");

lNumeC.setBounds (10,70,80,30);

JLabel lEmail = **new** JLabel ("Email");

lEmail.setBounds (10,100,80,30);

JLabel lNrTel= **new** JLabel ("Nr. tel:");

lNrTel.setBounds (10,130,80,30);

//JTextFields

**final** JTextField tNumeP = **new** JTextField();

tNumeP.setBounds (90,15,130,20);

**final** JTextField tCant = **new** JTextField();

tCant.setBounds (90,45,130,20);

**final** JTextField tNumeC = **new** JTextField();

tNumeC.setBounds (90,75,130,20);

**final** JTextField tEmail = **new** JTextField();

tEmail.setBounds (90,105,130,20);

**final** JTextField tNrTel = **new** JTextField();

tNrTel.setBounds (90,135,130,20);

//Buton

JButton okBtn = **new** JButton("OK");

okBtn.setBounds (100,160,55,30);

okBtn.addActionListener (**new** ActionListener()

{

@SuppressWarnings("unchecked")

**public** **void** actionPerformed (ActionEvent e)

{

String[] rand = **new** String[8];

**try** {

rand[1] = tNumeP.getText();

rand[2] = tCant.getText();

rand[4] = tNumeC.getText();

rand[5] = tEmail.getText();

rand[6] = tNrTel.getText();

**if** (Integer.*parseInt* (rand[2]) <= 0) **throw** **new** Exception();

**if** (Integer.*parseInt* (rand[6]) <= 0) **throw** **new** Exception();

Order ord = **new** Order (rand, wh);

rand[0] = Integer.*toString* (ord.getCod());

rand = setStatusCol (ord, rand);

addRow (getTableModel(), rand);

opd.getTreeMap().put (ord.getCod(), ord);

intro2.setVisible (**false**);

opd.serialization();

}

**catch** (Exception e1) {

JOptionPane.*showMessageDialog* (myPanel,

"Date introduse gresit!" +

"\nCantitatea si Nr.Tel trebuie sa aiba \nvalori numerice strict pozitive",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

});

intro2.setSize (250, 300);

intro2.setResizable (**false**);

intro2.setLayout (**null**);

intro2.setDefaultCloseOperation (*DISPOSE\_ON\_CLOSE*);

intro2.setVisible (**true**);

intro2.add (lNumeC);

intro2.add (lEmail);

intro2.add (lNrTel);

intro2.add (lNumeP);

intro2.add (lCant);

intro2.add (tNumeC);

intro2.add (tEmail);

intro2.add (tNrTel);

intro2.add (tNumeP);

intro2.add (tCant);

intro2.add (okBtn);

}

Urmatoarea metoda incarca datele din fisier in cei doi arbori, iar apoi acestia sunt parcursi si datele gasite sunt transcrise in tabelul corespunzator pentru a se putea vizualiza:

**private** **void** loadData() {

String[] rand;

Product p;

Order o;

wh.deserialization();

Set set = wh.getTreeMap().entrySet();

Iterator i = set.iterator();

**while** (i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

p= (Product) me.getValue();

rand= **new** String[] {Integer.*toString*(p.getCod()), p.getNume(), Integer.*toString* (p.getCant()), Integer.*toString*(p.getPretU())};

addRow (dtm1,rand);

}

opd.deserialization();

set = opd.getTreeMap().entrySet();

i = set.iterator();

**while** (i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

o = (Order) me.getValue();

rand = **new** String[] {Integer.*toString* (o.getCod()), o.getNumeProd(), Integer.*toString*(o.getCant()), Integer.*toString* (o.getPret()), o.getCust().getNume(), o.getCust().getEmail(), o.getCust().getNrTel(), Integer.*toString* (o.isStatus() ) };

rand = setStatusCol (o,rand);

addRow (dtm2,rand);

}

}

Urmatoarea metoda reintroduce datele din arbori in tabele, stergand anterior datele existente in tabele, evitand astfel dublarea datelor:

**private** **void** updateTables() {

dtm1.getDataVector().removeAllElements();

dtm2.getDataVector().removeAllElements();

loadData();

}

Pentru afisarea in tabel a mesajelor privind statusul comenzilor am folosit metoda urmatoare:

**private** String[] setStatusCol (Order o, String[] rand) {

**if** (o.getPret() == -1)

rand[3] = "-";

**else**

rand[3] = Integer.*toString* (o.getPret());

**if** (o.isStatus() == 1)

rand[7] = "Se poate efectua";

**else**

**if** (o.isStatus() == 0)

rand[7] = "Produs indisponibil";

**else**

rand[7] = "Depasire cantitate";

**return** rand;

}

Urmatoarea metoda sterge randurile selectate din tabel si obiectele corespunzatoare din arbore:

**private** **void** delSelRow (DefaultTableModel dtm, JTable jTab) {

Object s;

**for**(**int** i = 0; i <= jTab.getRowCount() ; i++ ) {

**if** (getTable().isRowSelected(i)) {

**if** (getTable() == jDepTab) {

s = jDepTab.getValueAt (i,1);

wh.getTreeMap().remove(s);

}

**else** {

s = jComTab.getValueAt (i,0);

opd.getTreeMap().remove (Integer.*parseInt* ((String) s));

}

}

}

**for** (**int** i = 0; i < sel ; i++ ) {

dtm.removeRow (jTab.getSelectedRow());

}

}

La efectuarea cu succes a unei comenzi, aceasta se sterge din arborele de comenzi si se scade cantitatea livrata din depozit, iar daca cantitatea ramasa in depozit este egala cu 0, se sterge produsul respectiv din arborele de produse. Codul sursa se poate vedea in continuare:

**for** (**int** i = 0; i <= getTable().getRowCount() ; i++ ) {

**if** (getTable().isRowSelected (i)) {

Object s = jComTab.getValueAt (i,0);

Order ord = (Order) opd.getTreeMap().get (Integer.*parseInt* ((String) s));

**if** (ord.isStatus() == 1) {

Product p = (Product) wh.getTreeMap().get (ord.getNumeProd());

p.setCant (p.getCant() - ord.getCant()); //se scade din depozit cantitatea livrata

**for** (**int** j = 0; j < jDepTab.getRowCount(); j++ ) {

**if** (jDepTab.getValueAt (j,1) == p.getNume())

**if** (p.getCant() > 0)

jDepTab.setValueAt (p.getCant(), j, 2);

**else**

{

dtm1.removeRow (j); //se sterge produsul din depozit daca cantitatea=0

wh.getTreeMap().remove (p.getNume());

}

}

opd.getTreeMap().remove (Integer.*parseInt* ((String) s));

getTableModel().removeRow (i);

}

**else**

JOptionPane.*showMessageDialog* (myPanel,

"Comanda nu poate fi efectuata!",

"Eroare",

JOptionPane.*ERROR\_MESSAGE*);

}

}

opd.serialization();

wh.serialization();

**Testare**

Testarea aplicatiei a fost efectuata pentru o multitudine de cazuri, inclusiv pentru introducerea de catre utilizator de date de intrare eronate. In toate aceste cazuri aplicatia a furnizat rezultatele asteptate.

**5. Concluzii, dezvoltari ulterioare**

Aceasta aplicatie de procesare a comenzilor reda intr-un mod usor de inteles si utilizat un mod de gestionare a produselor dintr-un depozit sau magazin, precum si a comenzilor venite din partea clientilor.

Ca si dezvoltari ulterioare s-ar putea mentiona utilizarea unei baze de date pentru stocare in locul arborilor si a serializarii, dar aceasta in functie de preferintele fiecaruia.

De asemenea s-ar putea genera o arhiva a produselor aflate in depozit si a comenzilor efectuate in trecut. Cu ajutorul acestei arhive s-ar putea observa intervalele in care au avut loc mai multe vanzari, sau intervalele in care depozitul nu a livrat un numar de produse multumitor.

O alta idee de dezvoltare ulterioara ar putea fi retinerea profitului depozitului in anumite intervale de timp ajutand astfel la o mai buna observare a dinamicii acestuia de catre detinatorul depozitului.

**6. Bibliografie**

<http://lec-academy.ro/category/java/>

<http://download.oracle.com/javase/6/docs/api/>

<http://download.oracle.com/javase/6/docs/api/java/util/TreeMap.html>