

Calitate si testare software

Proiect



Academia de Studii Economice

Iordache Robert

Grupa 1069

Seria B

Cuprins

# 1.Descrierea aplicatiei

# 2.Argumentarea pattern-urilor folosite

## 2.1.Builder

## 2.2Chain

## 2.3.Facade

## 2.4.Simple Factory

## 2.5.Singleton

## 2.6. Proxy

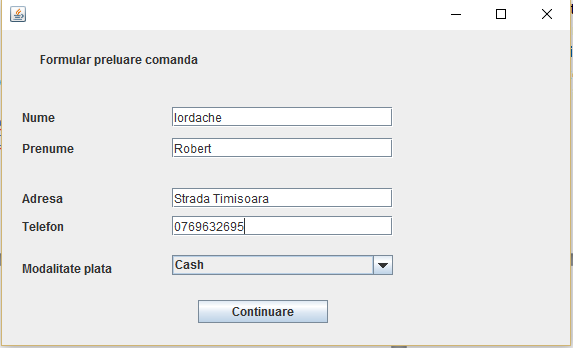
## 2.7.State

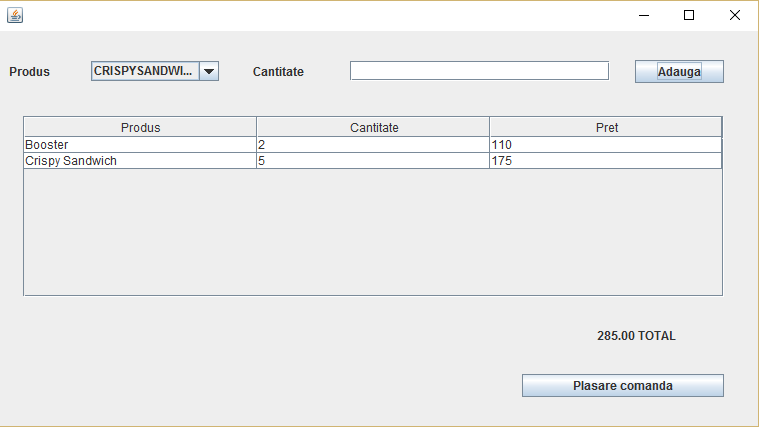
## 2.8.Strategy

## 2.9.Observer

# 1.Descrierea aplicatiei

Aplicatia ce face obiectul acestei lucrari reprezinta o unealta prin intermediul careia clientii pot comanda diferite tipuri de produse alimentare cum ar fi sandwich-urile. Ca faza initiala, clientul trebuie sa isi completeze datele de contat, urmand sa iti umple „cosul de cumparaturi” dupa preferinte. Restaurantul trebuie sa proceseze comanda pentru ca produsele sa fie pregatite din momentul plasarii comenzi si livrate cat mai optim din punct de vedere al resurselor utilizate.





# 2. Argumentarea pattern-urilor folosite

## 2.1 Builder

Builder este unu pattern de tip structural care usureaza creare obiectelor cu multe proprietati. Pentru a lucra cat mai usor cu obiectul de tip Client am pastrat in constructor doar proprietatile (nume si prenume) care sunt critice la intializare.

**public** **class** ClientBuilder {

**private** Client client;

**public** ClientBuilder(String nume, String prenume) {

**super**();

**this**.client=**new** Client(nume, prenume, "Bucharest", "0762000000", TipPlata.***CASH***);

}

**public** Client build(){

**return** **this**.client;

}

**public** ClientBuilder setAdresa(String adresa){

**this**.client.setAdresa(adresa);

**return** **this**;

}

**public** ClientBuilder setTelefon(String telefon){

**this**.client.setTelefon(telefon);

**return** **this**;

}

**public** ClientBuilder setTipPlata(TipPlata tip){

**this**.client.setTipPlata(tip);

**return** **this**;

}

}

## 2.2 Chain

Livrarea comenzilor se realizeaza cu ajutorul unei masini si a unui scuter. Alegerea tipului de transport se face in funtie de cantitatea comenzi si anume: daca in componenta comenzi sunt mai putin de 20 de produse livrarea se realizeaza cu scuterul, altfe livrarea se realizeaza cu masina.

**public** **abstract** **class** Curier {

**protected** Curier succesor=**null**;

**public** **void** setSuccesor(Curier curier){

**this**.succesor=curier;

}

**public** **abstract** **void** livrareComanda(Comanda comanda);

}

**public** **class** Masina **extends** Curier {

@Override

**public** **void** livrareComanda(Comanda comanda) {

**if**(succesor!=**null** && comanda.getProduseCos().size()<20)

succesor.livrareComanda(comanda);

**else**{

System.***out***.println("Comanda a fost livrata cu masina.");

}

}

}

**public** **class** Scuter **extends** Curier{

@Override

**public** **void** livrareComanda(Comanda comanda) {

System.***out***.println("Comanda a fost livrata cu scuter-ul.");

}

}

## 2.3.Facade

Am folosit acest pattern pentru a implementa o interfata ce permite controlul tuturor entitatilor contribuitoare al realizarea unei comenzi. Magazinul este compus dintr-un depozit, mijloacele de curierat si bucatarii.

**public** **class** Magazin {

**private** ProxyDepozit depozit;

**private** Masina masina;

**private** Scuter scuter;

**private** Bucatar bucatar1;

**private** Bucatar bucatar2;

**private** Bucatar bucatar3;

**public** **void** procesareComanda(Client client, Comanda comanda, DefaultTableModel model) **throws** Exception {

**if**(client==**null**)**throw** **new** ExceptieNull("Un client null nu poate face o comanda.");

**if**(comanda==**null**)**throw** **new** ExceptieNull("Nu se poate procesa o comanda null");

**if**(model==**null**)**throw** **new** ExceptieNull("Exceptie model null");

**if**(comanda.getProduseCos().size()==0)**throw** **new** Exception("Exceptie procesare comanda fara produse");

preparareProduse(comanda);

masina.livrareComanda(comanda);

newCos(model);

}

**private** **void** preparareProduse(Comanda comanda) {

**for**(Produs produs:comanda.getProduseCos()){

**for**(String ingredient:produs.ingrediente)

depozit.obtineIngrediente(ingredient);

}

bucatar1.procesareComanda(comanda,bucatar2);

bucatar1.setStareCurenta(StareBucatar.***OCUPAT***);

}

}

## 2.4.Simple Factory

Simple Factory permite crearea obiectelor din familia produselor cum ar fi

***BOOSTER*** si ***CRISPYSANDWICH.***

**public** **class** ProdusFactory {

**public** **static** Produs creareProdus(TipProdus tipProdus){

Produs produs=**null**;

**switch** (tipProdus) {

**case** ***BOOSTER***:

produs=**new** Booster();

**break**;

**case** ***CRISPYSANDWICH***:

produs=**new** CrispySandwich();

**break**;

}

**return** produs;

}

}

## 2.5.Singleton

La nivelul magazinului aprovizionarea cu ingrediente pentru prepararea produselor se face dintr-un singur depozit. Prin urmare accesarea acestuia reprezinta accesarea aceleasi instante indiferent de locul

**public** **class** Depozit **implements** IDepozit {

**private** **static** Depozit *instance*=**null**;

**private** ArrayList<String> ingrediente=**new** ArrayList<>();

**private** Depozit(String file) **throws** IOException {

**super**();

ingrediente=loadFromFile(file);

}

**public** ArrayList<String> loadFromFile(String string) **throws** IOException {

ArrayList<String> ingredienteFisier=**new** ArrayList<>();

BufferedReader br = **new** BufferedReader(**new** FileReader(string));

**try** {

String line = br.readLine();

**while** (line != **null**) {

System.***out***.println(line);

ingredienteFisier.add(line);

line = br.readLine();

}

} **finally** {

br.close();

}

**return** ingredienteFisier;

}

**public** **static** Depozit getInstance(String file) **throws** IOException{

**if**(*instance*==**null**){

*instance*=**new** Depozit(file);

}

**return** *instance*;

}

}

## 2.6. Proxy

Depozitul are un numar finit de ingrediente, iar acest pattern asigura aprovizionarea ori de cate ori este nevoie.

**public** **class** ProxyDepozit **implements** IDepozit{

**private** Depozit depozit;

**public** ProxyDepozit(String file) {

**super**();

**try** {

**this**.depozit=Depozit.*getInstance*(file);

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

@Override

**public** **boolean** obtineIngrediente(String ingredient) {

**if**(ingredient==**null**)**throw** **new** ExceptieNull(" Obtinere ingredient null");

**if**(ingredient.length()==0) **throw** **new** InvalidParameterException("Exceptie obtinere ingredient de lungime 0");

// **TODO** Auto-generated method stub

**if**(!(depozit.getIngrediente().contains(ingredient)))

depozit.aprovizionare();

**return** depozit.obtineIngrediente(ingredient);

}

}

## 2.7.State

Magazinul dispune de mai multi bucatari care sunt reprezentati de 2 stari, OCUPAT sau LIBER. Comanda odata plasata ajunge (chain of responsability) la primul bucatar LIBERT pentru a fi pregatita.

**public** **class** Bucatar **extends** BucatarGeneric {

String nume;

IProcesabil stareCurenta;

**public** Bucatar(String nume) {

**super**();

**this**.nume=nume;

**this**.stareCurenta = **new** BucatarLiber();

}

**public** **void** setStareCurenta(StareBucatar stareBucatar) {

**if**(stareBucatar==StareBucatar.***LIBER***)

**this**.stareCurenta=**new** BucatarLiber();

**else** **this**.stareCurenta=**new** BucatarOcupat();

}

@Override

**public** **void** procesareComanda(Comanda comanda, BucatarGeneric succesor) {

// **TODO** Auto-generated method stub

stareCurenta.procesareComanda(**this**.nume,comanda,succesor);

}

}

**public** **class** BucatarLiber **implements** IProcesabil{

@Override

**public** **void** procesareComanda(String nume,Comanda comanda,BucatarGeneric succesor) {

// **TODO** Auto-generated method stub

System.***out***.println("Produsele sunt gatite de "+nume);

}

}

**public** **class** BucatarOcupat **implements** IProcesabil {

@Override

**public** **void** procesareComanda(String nume,Comanda comanda,BucatarGeneric succesor) {

// **TODO** Auto-generated method stub

**if**(succesor!=**null**)

succesor.procesareComanda(comanda, succesor);

**else** System.***out***.println("Comanda nu poate fi preluare. Toti bucatarii sunt ocupati.");

}

}

## 2.8.Strategy

Fiecare Comanda este compusa dintr-o lista de produse cu ajutorul carora calcula valoarea totala a comenzi. In functie de numarul de produse (>10) se poate adauga un disocount de 30%.

**public** **class** Comanda **extends** Observabil{

**private** ArrayList<Produs> produseCos=**new** ArrayList<>();

**private** ICalculabil metodaCalculDiscount;

**private** **int** capacitateMaxima=100;

**public** Comanda() {

**super**();

**this**.metodaCalculDiscount = **new** Discount0();

}

**public** **void** setMetodaCalculDiscount(ICalculabil metodaCalculDiscount) {

**this**.metodaCalculDiscount = metodaCalculDiscount;

}

**public** **long** calculeazaValoareaTotala(){

**if**(**this**.produseCos.size()>10)

**this**.metodaCalculDiscount=**new** Discount30();

**long** suma=0;

**for**(Produs produs:**this**.produseCos)suma+=produs.pret;

**return** **this**.metodaCalculDiscount.calculValoareTotala(suma);

}

}

## 2.9.Observer

**public** **class** HandlerDiscountLabel **extends** JLabel **implements** HandlerDiscount{

@Override

**public** **void** afisarePretTotal() {

// **TODO** Auto-generated method stub

setText(getText().toString()+" DISCOUNT30%");

}

}

**public** **abstract** **class** Observabil {

ArrayList<HandlerDiscount> observatori = **new** ArrayList<HandlerDiscount>();

**public** **void** abonareHandler(HandlerDiscount handlerDiscount){

**if**(observatori!=**null**)

observatori.add(handlerDiscount);

}

**public** **void** dezabnareHandler(HandlerDiscount handlerDiscount){

**if**(observatori!=**null**)

observatori.remove(handlerDiscount);

}

**public** **void** notificareObservatori(){

**if**(observatori!=**null**)

**for**(**int** i=0;i<observatori.size();i++)

observatori.get(i).afisarePretTotal();

}

}

**public** **interface** HandlerDiscount {

**public** **void** afisarePretTotal();

}