**Aplicație pentru domeniul comercial**

**(magazin online)**

Proiect Calitate și Testare Software

(documentație)

Realizat de:

Vișan Andreea Nicoleta

Aplicatia realizata pentru acest proiect simuleaza actiunile unui magazin online, fiind construit cu ajutorul a 3 design pattern-uri (**Builder**, **Strategy** si **Memento**).

# 1 Definirea si detalierea pattern-urilor utilizate

## 1.1 Definirea si detalierea design pattern-ului Builder

Am utilizat acest pattern pentru a crea obiecte ale clasei **Produs** specificand doar tipul si valoarea acestora, fara a fi nevoie sa se cunoasca detalii interne ale obiectului (modul in care acesta stocheaza datele sau reprezinta valorile) si trebuie sa se permita construirea de obiecte diferite din aceeasi familie. De asemenea, instanta de tip Builder, in cazul acestei aplicatii este vorba de clasa **ProdusBuilder** (din pachetul DesignPatterns) care se ocupa de crearea obiectului de tip **Produs** propriu zis, construieste obiectul insa tipul acestuia este definit de subclase ale acestuia (clasa **Produs**).

Pentru implementare a fost nevoie de urmatoarele clase:

* Clasa **Produs** – clasa concreta ce defineste obiectul complex ce este construit. Contine atributele *denumire*, *cantitate* si *pret* si metoda *toString()* pentru afisarea obiectului impreuna cu detaliile sale.
* Clasa **ProdusBuilder** – clasa concreta ce construieste partile obiectului complex si pe baza acestora, obiectul final. Contine un atribut de tip **Produs** (cel pe care urmeaza sa il construiasca), constructor in care se aloca spatiu pentru produs si metodele *setDenumire(), setCantitate(), setPret()* pentru preluarea partilor obiectului. Un aspect important al acestor metode este ca ele intorc un obiect de tip **ProdusBuilder**, tocmai pentru a putea le putea apela in lant si construi usor si eficient obiectul. Metoda cea mai importanta a acestei clase este metoda *build()* care intoarce obiectul construit.

## 1.2 Definirea si detalierea design pattern-ului Strategy

Utilizarea acestui pattern a prins contur atunci cand s-a stabilit ca utilizatorul (clientul) sa isi poata efectua plata cosului de cumparaturi in doua moduri: plata cu bani gheata si plata cu cardul. In plus, modul de plata al clientului se stabileste la run-time.

Pentru implementare a fost nevoie de urmatoarele clase:

* Clasa **ModPlataStategy** – clasa abstracta ce defineste interfata obiectelor care ofera noi functii de prelucrare. Contine metoda *plateste().*
* Clasele **PlataCash** si **PlataCuCardul** – clase care definesc obiectele ce furnizeaza solutii pentru prelucrarea datelor. Acestea mostenesc clasa abstracta **ModPlataStrategy** si suprascriu metoda *plateste(),* fiecare continand modul de plata specific.
* Clasa **CosCumparaturi** – clasa care gestioneaza cumparaturile unui client. Atributele acesteia sunt: o lista de produse si valoarea acestora.

Metodele acestei clase sunt numeroase: un constructor pentru initializarea listei de produse, o metoda pentru adaugarea produsului dat ca parametru in lista – *adaugaProdus(Produs produs),* o metoda pentru stergere produsului dat ca parametru din lista daca acesta a fost adaugat in prealabil – *stergeProdus(Produs produs),* o metoda pentru afisarea listei de cumparaturi – *afisareCos(),* o metoda pentru calcularea valorii cosului de cumparaturi care totodata seteaza atributul *valoare* al clasei – *calculeazaValoareCos(),* un getter pentru atributul *valoare*, o metoda care trimite comanda numai daca aceasta contine cel putin un produs – *trimiteComanda().*

* Clasa **Client** – cea care gestioneaza o referinta de tip **ModPlataStrategy** catre obiectul care va oferi functia, gestionand totodata si datele ce necesita prelucrare. Aceasta clasa contine atributele: un cos de cumparaturi (referinta de tip **CosCumparaturi**), un nume si un mod de plata (referinta de tip **ModPlataStrategy**).

Metodele acesteia sunt: un constructor cu parametrii si metoda *platesteComanda()* care calculeaza valoarea cosului de cumparaturi (daca acesta contine cel putin un produs) si apeleaza achitarea sumei prin metoda *plateste()* a referintei modului de plata.

## 1.3 Definirea si detalierea design pattern-ului Memento

Pentru ca utilizatorul sa isi poata salva starea unui cos de cumparaturi si sa poata restaura aceasta stare in orice moment pe baza unei imagini anterioare, s-a utilizat pattern-ul **Memento**.

Pentru implementare a fost nevoie de urmatoarele clase:

* Clasa **MementoCosCumparaturi** - este cea care gestioneaza starea interna a obiectului CosCumparaturi pentru un anumit moment. Este creat de **CosCumparaturi** si este gestionat de **CareTakerCos**. Ca si atribut contine o lista de referinte de tip Produs si un constructor care initializeaza aceasta lista.
* Clasa **CosCumparaturi** – este obiectul a carei stare este urmarita. Au mai fost adaugate metoda *creareMemento()* care permite generarea unei instante a clasei **MementoCosCumparaturi** – o imagine a cosului de cumparaturi la un moment dat si metoda *setareMemento(memento)* care realizeaza restaurarea ultimii stari salvate a cosului*.*
* Clasa **CareTakerCos** – este cea care gestioneaza obiectele de tip **MementoCosCumparaturi** fara a avea insa acces la continutul acestora. Aceasta clasa are un singur atribut: o lista cu referinte de tip **MementoCosCumparaturi** care memoreaza salvarile realizate in diferite momente de timp. Contine metoda *adaugaSalvare(MementoCosCumparaturi)* care salveaza referinta de tip memento in lista si metoda *restaurareUltimaSalvare()* care intoarce ultima referinta salvata de tip **MementoCosCumparaturi** in cazul in care aceasta exista.

Deoarece firma practica strategii de marketing prin promotii, aplicatia va utiliza si clasa **Promotie** care contine atributul *procent*, un numar intre 0 exclusiv si 100 inclusiv si doua metode. Una dintre aceste metode este *aplicareDiscount(CosCumparaturi)* care aplica procentul de discount pe valoarea cosului de cumparaturi. Cea de-a doua metoda este *scadereVoucher (CosCumparaturi,valoareVoucher)* care scade din valoarea cosului de cumparaturi voucherul pe care clientul l-a primit de la firma in urma unor cumparaturi anterioare.

# 2. Unit Testing

## 2.1 TestPromotie

Primul **TestCase** a fost implementat pentru metodele clasei **Promotie** si se numeste **TestPromotie**. In functia *setUp()* a TestCase-ului se citesc datele initiale automat din fisierul *Produse.txt*. Este prezenta si functia *tearDown()* care asigura structura completa a testului. Acesta contine urmatoarele functii de testare:

* Pentru testarea metodei *aplicareDiscount(CosCumparaturi)*
* *testValoriNormaleAplicareDiscount() –* testeaza metoda cu valori normale (cosul de cumparaturi cu care a fost testata contine doua produse si este valid). A fost facut in mod automat, valorile pentru test fiind preluate din fisierul *Produse.txt*. A fost folosita metoda de test *assertEquals (mesaj, expected, actual).*
* *testCosCumparaturiGol()* – testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un cos de cumparaturi cu niciun produs. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un cos gol ca si parametru, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).
* *testProcentDiscountNegativ() –* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un procent negativ, acesta neavand nici un sens economic. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un procent negativ ca si parametru, va arunca o exceptie definita de dezvoltator numita **ProcentNegativException** (derivata din clasa abstracta **PromotieException**).
* *testProcentNull() –* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un procent egal cu 0. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un procent negativ ca si parametru, va arunca o exceptie definita de dezvoltator numita **ProcentNullException** (derivata din clasa abstracta **PromotieException**).
* *testProcentPeste100() -* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un procent mai mare decat 100. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un procent negativ ca si parametru, va arunca o exceptie definita de dezvoltator numita **ProcentPeste100Exception** (derivata din clasa abstracta **PromotieException**).
* *testProcentLimita() -* testeaza daca metoda se comporta normal atunci cand ii este dat ca parametru un procent egal cu 100, limita superioara a intervalului de valori. A fost folosita metoda de test *assertEquals(mesaj, expected, actual).*
* Pentru metoda *scadereVoucher(CosCumparaturi, valoareVoucher)*
* *testValoriNormaleScadereVoucher() –* testeaza metoda cu valori normale (cosul de cumparaturi cu care a fost testata contine doua produse si este valid iar voucherul are o valoare acceptabila, normala). A fost folosita metoda de test *assertEquals (mesaj, expected, actual).*
* *testValoriNegativeScadereVoucher() -* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un voucher cu valoare negativa, acesta neavand nici un sens economic. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un voucher cu valoare negativa ca si parametru, va arunca o exceptie definita de dezvoltator numita **VoucherValoareNegativaException** (derivata din clasa abstracta **PromotieException**).
* *testValoareNulaScadereVoucher() -* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un voucher cu valoare egala cu 0, acesta neavand nici un sens economic. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un voucher cu valoare nula ca si parametru, va arunca o exceptie definita de dezvoltator numita **VoucherValoareNulaException** (derivata din clasa abstracta **PromotieException**).
* *testValoareVoucherPesteLimita() -* testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un voucher cu o valoare mai mare decat valoarea cosului de cumparaturi, aceasta fiind peste limita superioara a valorii voucherului. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un voucher cu o valoare mai mare decat valoarea cosului de cumparaturi ca si parametru, va arunca o exceptie definita de dezvoltator numita **VoucherValoarePesteLimitaException** (derivata din clasa abstracta **PromotieException**).
* *testCosCumparaturiGolScadereVoucher()* – testeaza daca metoda arunca exceptie atunci cand ii este dat ca parametru un cos de cumparaturi cu niciun produs. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care primeste un cos gol ca si parametru, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).

## 2.2 TestClient

Acest **TestCase** a fost implementat pentru metodele clasei **Client** si se numeste **TestClient**. In functia *setUp()* a TestCase-ului se initializeaza datele necesare testarii. Este prezenta si functia *tearDown()* care asigura structura completa a testului. Acesta contine urmatoarele functii de testare:

* Pentru metoda *platesteComanda()*
* *testValoriNormalePlatesteComanda()* - testeaza metoda cu valori normale (cosul de cumparaturi al clientului cu care a fost testata contine doua produse si este valid). A fost folosita metoda de test *assertEquals (mesaj, expected, actual).*
* *testValoriAberantePlatesteComanda()* – testeaza daca metoda arunca exceptie atunci cand cosul de cumparaturi al clientului care doreste platirea comenzii nu contine niciun produs. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care cosul de cumparaturi al clientului este gol, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).

## 2.3 TestCosCumparaturi

Acest **TestCase** a fost implementat pentru metodele clasei **CosCumparaturi** si se numeste **TestCosCumparaturi**. In functia *setUp()* a TestCase-ului se initializeaza datele necesare testarii. Este prezenta si functia *tearDown()* care asigura structura completa a testului. Acesta contine urmatoarele functii de testare:

* Pentru metoda *calculeazaValoareCos()*
* *testValoriNormaleCalculeazaValoareCos()* - testeaza metoda cu valori normale (cosul de cumparaturi cu care a fost testata contine doua produse si este valid). A fost folosita metoda de test *assertEquals (mesaj, expected, actual).*
* *testValoriAberanteCalculeazaValoareCos()* – testeaza daca metoda arunca exceptie atunci cand cosul de cumparaturi nu contine niciun produs deoarece este aberant sa se calculeze valoarea cand nu exista produse pentru care sa se calculeze. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care cosul de cumparaturi este gol, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).
* Pentru metoda *trimiteComanda()*
* *testCosGolTrimiteComanda()* – testeaza daca metoda arunca exceptie atunci cand cosul de cumparaturi nu contine niciun produs deoarece comanda nu poate fi trimisa in acest caz. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care cosul de cumparaturi este gol, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).
* Pentru metoda *creareMemento()*
* *testCosGolCreareMemento()* – testeaza daca metoda arunca exceptie atunci cand cosul de cumparaturi nu contine niciun produs si se doreste salvarea acestuia pentru o ulterioara restaurare, neavand niciun sens salvarea unui cos gol. A fost folosita metoda de test *assertFalse(mesaj, condition).* Dupa corectare metoda, in cazul in care cosul de cumparaturi este gol, va arunca o exceptie definita de dezvoltator numita **CosGolException** (derivata din clasa abstracta **PromotieException**).

## 2.4 TestSuiteOnlineShop

**TestSuite** este o colectie de teste care au fost grupate datorita similaritatii testarii. In cazul aplicatiei prezentate, TestSuite-ul implementat pentru aceasta este **TestSuiteOnlineShop** care grupeaza toate testele continute in TestCase-urile enumerate mai sus la punctele 1, 2 si 3. Acesta are ca scop rularea automata a tuturor testelor pentru aceasta aplicatie, nefiind nevoie sa se ruleze fiecare TestCase in parte.

Acesta contine metoda specifica unui TestSuite, *suite()* care realizeaza o referinta de tip TestSuite si adauga la aceasta toate TestCase-urile implementate in aceasta aplicatie cu ajutorul metodei *addTestSuite(TestPromotie.class)* si returneaza referinta.

Metodele *setUp()* si *tearDown()* ale TestSuite-ului sunt metode globale pentru TestCase-urile care intra in alcatuirea acestuia.

# 3. Concluzie

In concluzie, aplicatia a fost proiectata pentru ca utilizatorul sa poata sa isi aleaga produsele dorite, sa le poata adauga in cosul de cumparaturi si sa poata trimite comanda. Pentru asta, s-au folosit design pattern-urile Builder, Strategy si Memento care au ajutat extrem de mult la proiectarea aplicatiei si structurarea claselor si interfetelor. Builder-ul a facilitat crearea obiectelor complexe, contruind pe rand cate o particica pana s-a ajuns la obiectul intreg, pattern-ul Strategy a contribuit la buna functionare si flexibilitate a aplicatiei, permitand clientului sa aleaga la run-time metoda de plata care i se potriveste cel mai bine, iar Memento a facilitat utilizatorului posibilitatea de a reveni la o optiune de cumparaturi anterioara, prin restaurarea unui cos de cumparaturi salvat la un moment de timp anterior.

Pentru ca aceste lucruri sa se intample intr-un mod cat mai corect, dupa implementarea claselor si a metodelor cu care acestea lucreaza, s-au facut teste de determinare a cazurilor care s-au omis a se trata in cadrul metodelor. S-au descoperit astfel destul de multe metode care au avut nevoie de ajustari, de completari pe care daca nu s-ar fi folosit testele de tip TestCase ar fi fost trecute cu vederea.

De exemplu, era absolut necesara verificarea functiei *aplicareDiscount()* pentru a nu primi, din greseala, ca parametru un procent negativ,nul sau mai mare decat 100, cazuri pe care functia trebuie sa le trateze in mod special.

De asemenea, testarea continutului cosului de cumparaturi a fost un test extrem de des intalnit pentru mai multe metode cum ar fi: *aplicareDiscount()* (nu se poate aplica discount pe nimic, este aberant), *scadereVoucher()* (nu se poate scadea valoarea unui voucher din valoarea 0 a cosului de cumparaturi), *trimiteComanda()* (o comanda nu poate fi trimisa daca nu contine cel putin un produs, altfel nu mai este o comanda), *platesteComanda()* (un client nu poate plati daca nu are ce sa plateasca), *calculeazaValoareCos()* (nu se poate calcula valoarea unui cos daca acesta este gol, valoarea acestuia este 0, nu necesita calcule), etc.