A.S.E.

2016

Grupa 1069, seria B

Proiect C.T.S.

Ilie Dana Cristina

CUPRINS

[Introducere 2](#_Toc452638426)

[Design patternuri utilizate și detalierea lor 2](#_Toc452638427)

[Definirea si detalierea metodelor testate prin Unit Testing 6](#_Toc452638428)

[Definirea si descrierea Test Case-urilor și Test Suite-urilori 7](#_Toc452638429)

[Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate 9](#_Toc452638430)

[Bibliografie 11](#_Toc452638431)

## Introducere

Urmatorul proiect a fost realizat in Java in mediul Eclipse. Acesta presupune gestiunea studentilor dintr-o facultate privata.

În implementarea proiectului s-au folosit 8 design patternuri respectiv: Singletone, Simple Factory, Builder, Strategy, Memento, Observer, Adapter si Chain of Respnsability.

## Design patternuri utilizate și detalierea lor

1.Singletone

Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.

Aplicarea pattern-ului Singleton constă în implementarea unei metode ce permite crearea unei noi instanțe a clasei dacă aceasta nu există, și întoarcerea unei referințe către aceasta dacă există deja. În Java, pentru a asigura o singură instanțiere a clasei, constructorul trebuie să fie *private*, iar instanța să fie oferită printr-o metodă statică, publică.

În aplicația mea, clasa Facultate este implementată folosind design pattern-ul Singletone deoarece gestionează lista de studenti ai facultatii și folosind o singură instanță o putem gestiona mai ușor și putem să obținem mereu lista studentilor cu toate modificările suferite.

**private** **static** Facultate *singleton*=**null**;

**private** Facultate()

{

System.***out***.println("S-a creat instanta");

}

**public** **static** Facultate getInstance() {

**if** (*singleton* == **null**) {

*singleton* = **new** Facultate();

}

**return** *singleton*;

}

2. Simple Factory

Patternurile de tip Factory sunt folosite pentru obiecte care generează instanțe de clase înrudite (implementează aceeași interfață, moștenesc aceeași clasă abstractă). Acestea sunt utilizate atunci când dorim să izolăm obiectul care are nevoie de o instanță de un anumit tip, de creearea efectivă acesteia. În plus clasa care va folosi instanța nici nu are nevoie să specifice exact subclasa obiectului ce urmează a fi creat, deci nu trebuie să cunoască toate implementările acelui tip, ci doar ce caracteristici trebuie să aibă obiectul creat. Din acest motiv, Factory face parte din categoria *Creational Patterns*, deoarece oferă o soluție legată de creearea obiectelor.

In aplicatia mea, clasele StudentLicenta, StudentMasterat si StudentDoctorat extind aceeasi clasa si anume: Student. In main() sunt create apoi student folosind StudentFactory si tipul studentului din enumerare(LICENTA,MASTER,DOCTORAT).

**public** **class** StudentFactory {

**public** Student createInstance(TipStudent tip)

{

**switch**(tip){

**case** ***LICENTA***: **return** **new** StudentLicenta(9802,"Ilie", "Dana", "2940112090047",2000,"zi","info");

**case** ***MASTER***: **return** **new** StudentMaster(5678,"Ion", "Gabriela", "2920112090067",4000,"zi","cibe");

**case** ***DOCTORAT***: **return** **new** StudentDoctorat(4678,"Mihai", "Radu", "1900112090067",6000,"distanta","cibe");

}

**return** **null**;

}

}

3. Builder

Soluția trebuie sa construiască obiecte complexe printr-un mecanism care este independent de procesul de realizare a obiectelor. Clientul construiește obiectele complexe specificând doar tipul si valoarea sa, fără a cunoaște detaliile interne ale obiectului (cum stochează si reprezintă valorile). Procesul de construire a obiectelor trebuie sa poată fi utilizat pentru a defini obiecte diferite din aceeași familie. Obiectele sunt gestionate prin interfața comuna. Instanța de tip Builder construiește obiectul însă tipul acestuia este definit de subclase.

Avantaje:

* Obiectele complexe pot fi create independent de părțile care îl compun (un obiect poate sa le conțină pe toate sau doar o parte)
* Sistemul permite reprezentarea diferita a obiectelor create printr-o interfață comună
* Algoritmul de creare a obiectului este flexibil deoarece clientul alege ce părți sa fie create

In aplicatia dezvoltata, studentul poate fi descries doar cu NrMatricol, Nume si Prenume, celelalte campuri (cnp,taxa,formaInvatatmant,specializare) fiind completate implicit atunci cand se apeleaza metoda build() din clasa StudentBuilder

**public** StudentBuilder(**int** nrMatricol,String nume,String prenume)

{

**this**.student=**new** Student(nrMatricol,nume,prenume,"2920119070045",2500,"zi","info");

}

**public** Student build()

{

**return** **this**.student;

}

In main()

Student s3=new StudentBuilder(875, "Popa", "Andreea").build();

System.out.println( s3.toString());

Output:

Student [nrMatricol=875, nume=Popa, prenume=Andreea, cnp=2920119070045, taxa=2500, formaInvatamant=zi, specializare=info]

4. Adapter

Acesta reprezintă un design pattern structural care permite utilizare împreună a claselor ce nu au o interfață comună. Aceste clase nu se modifică însă se construiește o interfață nouă ce permit utilizarea lor în alt context și clasele sunt adaptate la acesta. Apelurile către interfața clasei sunt mascate de interfața adaptorului și datele sunt transformate dintr-un format în altul.

Marele avantaj al acestui design pattern este că nu sunt modificare clasele existente ci doar se adaugă un layer intermediar

În aplciația mea, a fost creat un adaptor pentru Student ce va permite utilizarea aceste clase într-o nouă interfață ce va crea rapoarte pentru Contabilitate (calcularea taxei la studenti).

5. Strategy

Algoritmul se poate alege pe baza unor condiții descrise la execuție in funcție de contextul datelor de intrare . Clasa existenta nu trebuie sa fie modificata. Utilizarea unei abordări tradiționale, prin includerea in clasa a tuturor metodelor posibile, duce la ierarhii complexe ce sunt greu de gestionat. Derivarea adaugă comportament nou doar la compilare

**public** String poateSaMunceasca()

{

**return** tipMunca.esteAngajat();

}

**public** **void** setAbilitateMunca(Munceste tipMuncaNou)

{

tipMunca=tipMuncaNou;

}

6. Memento

Memento este un design pattern comportamental utilizat pentru aplicații care trebuie sa permita salvarea stării unui obiect. Imaginile stării obiectului în diferite momente sunt gestionate separate și obiectul are posibilitatea să-și restaureze stare ape baza unei imagini anterioare.

Cele 3 componente ale acestui design pattern sunt:

* Memento –componentă ce gestioneaza starea internă a obiectului care este Originator pentru un anumit moment dat. El este creat de Originator și gestionat de CareTaker.
* Originator- această component reprezintă oObiectul a cărui stare este urmărita și poate genera un Memento cu starea lui la momentul respectiv. De asemanea, el își poate reface starea anterioară pe baza unui memento.
* Caretaker-este component care gestionează obiectele de tip Memento dar care nu are acces pe conținutul acestora.

În cadrul aplicației mele, acest design pattern este folosit pentru a gestiona schimbările suferite de studentii facultatii astfel incat cineva sa poata avea acces la modificari. Pe baza acestei liste noi de studenti modificați, sudentii pot fi readuși la starea lor inițială.

7. Chain of Responsibility

In cazul utilizarii design pattern-ului Chain of Responsibility, tratarea unui eveniment sau a unui obiect se face diferit in funcție de starea acestuia. Gestiunea tuturor cazurilor ar implica o structură complexă care să verifice toate cazurile particulare. Există legături de dependență între cazurile de utilizare: execuția unui caz poate implica ignorarea celorlalte sau tratarea următorului caz

In aplicatia mea am folosit Chain of Responsibility pentru a returna orarul de la secretariatul facultatii in functie de ziua saptamanii.

8. Observer

Observer este un design pattern comportamental și este utilizat deoarece în cadrul aplicației există component ce trebuie notificate la producerea unui evenimen. Acest design pattern este util și îîn gestiunea evenimentelor la nivel de interfață. Componentele abonate sau înregistrate la evenimentul respective, suferă la declanșarea sa modificări de stare sau de acțiune. În cazul producerii unui evenimnt se pot notifica mai multe componente.

Observer-ul externlizează sau deleagă funcții către compprtamente de tip observer care dau soluții la anumite evenimente independent de propietarul acestora. Obiectele sunt interconectate prin notificări și nu prin instanțieri de clase și apeluri de metode.

Există 2 modele de notificare alea observatorului la modificare stării și anume:

Pull-observatorul este doar notificat de către obiecte dar cere datele când are nevoie de ele

Push-observatorului îi sunt trimise toate datele de către obiect.

În cadrul proiectului este implementat un Observer care este notificat atunci când au loc modificări la nivelul listei de studenti. Acest design pattern va fi utilizat atunci când se va crea o interfață în cadrul aplicației pentru a afișa constant numărul de studenti din facultate.

# Definirea si detalierea metodelor testate prin Unit Testing

**În cadrul clasei Student sunt testate metodele definite si detaliate mai jos:**

* **public Student(int nrMatricol,String nume, String prenume, String cnp, int taxa,String formaInvatamant, String specializare)** – constructorul cu parametrii
* **public void setNrMatricol(int nrMatricol)** – seteaza numarul matricol care trebuie sa fie obligatoriu mai mare decat 1.
* **public float discountTaxa()** – calculeaza taxa pentru studentii care sunt inscrisi la „distanta” (formaInvatamant). Pentru acesti studenti taxa va fi cu 70% mai mica (taxa=taxa\*0.7), iar pentru restul studentilor taxa returnata va fi aceeasi.
* **public float adaosTaxa()** – calculeaza adaosul la taxa in functie de specilizarea aleasa. Astfel, pentru „info” studentii for plati cu 30 % mai mult, pentru „cibe” cu 20% mai mult, iar pentru „statistica” cu 5% mai mult.

**În cadrul clasei Facultate sunt testate metodele definite și detaliate mai jos**

* **public void addStudent(Student s) –** metoda ce adauga un student in lista. Dupa adaugare se notifica observarii. Daca lista este nula se arunca o exceptie.
* **public static Facultate getInstance()**- metoda ce creeaza o instanta unica cu ajutorul singleton-ului.
* **public Student ceaMaiMareTaxa()** – metoda ce returneaza studentul cu cea mai mare taxa parcurgand lista de studenti si comparand taxele acestora. Daca lista este nula sau goala se arunca exceptii.
* **public void citireStudDinFisier()**- metoda ce adauga in lista de studenti studentii cititi din fisier.
* **public void eliminaStudent(Student s)**- metoda ce elimina un student daca acesta exista. Daca lista este nula se arunca o exceptie.
* **public ArrayList<Student> getStudentList()**- metoda ce returneaza lista de studenti. Daca lista este nula se arunca o exceptie.
* **public Student getStudent(Student s)**- metoda ce intoarce un anumit Student. Este parcursa lista de Studenti si este returnat studentul daca este gasit. Daca studentul nu exista se va returna null, iar daca lista este goala sau nulla se arunca exceptii.
* **public void modificaTaxa(Student s, int t)**- metoda care primeste ca parametrii un student si o taxa. Taxa studentului va fi modificata cu taxa primita ca parametru. Se parcurge lista, iar cand stduentul este gasit i se salveaza starea si o adaug intr-o lista, iar apoi i se modifica taxa.
* **public int nrBaieti()**- metoda ce calculeza numarul de studenti de gen masculin. Acest numar este calculat pe baza primei cifre din cnp. Daca lista este goala se returneaza valoarea 0, iar daca lista este nulla se arunca exceptii.
* **public int nrFete()** – metoda ce calculeaza numarul e studente din facultate similar metodei nrBaieti().

# Definirea si descrierea Test Case-urilor și Test Suite-urilori

1. AllTestsStudent

Acest Test Suite include cele 4 Test Case-uri utilizate pentur testarea metodelor clasei Student. La rularea acestuia se testează metodele clasei Student prin cele 10 teste implementate.

Test Case-ul **testAdaos** testează metoda adaos() a clasei Student. Cele 4 teste testează inișial cazul general în se determină specializarea la care este inscris studentul. S-a calculat separat adaosurile și s-au comparat cele două valori.

Celelalate 3 teste testează fiecare ramură a metodei adaos(): studentul este la Info, la Cibe sau la Statistica..

Metoda utilizată este assertEquals(rezultat așteptat, rezultat efectiv, delta). Delta reprezintă valoarea cu care pot să difere cele două valori.

Test Case-ul **testDiscount** testează metoda discount() a clasei Student. În cadrul acestuia s-au realizat 3 teste pentru această metodă. Unul testează cazul general de utilizare al metodei și celălalt testează metoda dacă studentul este la “distanta” sau nu..

Test Case-ul **testConstructor** testeaza constructorul clasei Student. Primul test testeaza constructorul atunci cand sunt introduce valori normale de la tastatura, iar cel de-al doilea test testeaza cazul in care este introdus un nume null. In cazul in care cel de-al doilea test ar da fail ar trebui sa apara mesajul “Lipsa exceptie pe nume=null”.

TestCase-ul **testNrMatricol** contine un test care verifica cazul in care este introdus un nr matricol mai mic decat 1, respective 0. In cazul in care testul ar pica, s-ar afisa mesajul “Accepta nr matricol<0”.

1. AllTestsFacultate

Test Suite-ul AllTestsFacultate include cele 10 Test Case-uri implementate. Acesta testează metodele clasei facultate folosind cele 28 de teste din Test Case-uri.

**testCeaMaiMareTaxa** este Test Case-ul utilizat pentru testarea metodei ceaMaiMareTaxa() din clasa facultate. Primul test crează o listă cu 2 studenti și o facultate ce conține lista respective. Se compaă rezultatul metodei cu cel mai bine plătit angajat dintre cei doi introduși anterior. Pentru cazurile în care lista este nulă sau goală, metoda trebuie să arunce excepții și pentru asta sunt implementate 2 teste.

**testCitireFișier** testează metoda prin care se adaugă studenti în lista facultatii cu date preluate dintr-un fișier. Pentru testare se face un fișier de test și se citesc de acolo datele. Se crează un student cu aceleași date și se compară rezultatele. În cazul în care fișierul nu există sau nu se poate deschide se aruncă excepții.

**testEliminaStudent** testează metoda eliminaStudent(). Se fac teste pentru cazurile în care lista facultatii este nulă sau goală și metoda trebuie să arunce excepții. Se mai face un test pentru eliminare unui stduent care nu există când metoda trebuie să returneze null și pentu eliminarea unui student deja existent. Pentru testarea eliminării se compara dimensiunile listelor folosite. Inițial s-a creat o listă cu studenti, se crează o facultate folosind lista respectivă apoi se elimină folosind metoda clasei studentul din lista facultatii..

**testGetStudent** testează metoda getStudent(). Se testează returnarea unui student existent și ne null, cazurile în care lista este nulă sau goală și trebuie aruncate excepții și cazul în care studentul nu există și trebuie să returnăm null.

**testGetLista** testeaza metoda getStudentList(). Se testeaza returnarea unei liste existente si nenule si cazul in care lista este nulla si trebuie aruncata o exceptie.

Test Case-ul **testModificaTaxa** testează metoda de modificare a taxei. Se fac teste pentru a testa metoda în cazul în care lista este goală sau nula și trebuie aruncate excepții. Se mai testează și încercarea modificării unui student null, caz în care iar se aruncă excepții. Și se testază modificare taxei unui student existent . Testul compară valoarea cu care se dorește modificarea taxei cu taxa obținuta în urma modificării.

**testAddStudent** testează metoda de adăugare a studentului în facultate. În cazul în care lista este nulă se aruncă excepții.

Test Case-ul **testNrStudBaieti** testează metoda prin care este returant numărul de baieti din facultate. Se testează cazurile în care lista e nulă și se aruncă excepții și cazul în care lista e goală și se returnează 0. Se testează metoda și prin compararea valorii returnate cu un număr dat.

**testNrStudFete** este Test Case-ul prin care se testează metoda prin care este returnat numărul de fete din facultate. Se testează metoda prin compararea valorii returnate cu un număr dat, dar și pentru o listă nulă când se aruncă excepții și pentru o listă goală când trebuie să se returneze 0.

# Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate

Aplicația este una de gestiune a studentilor din cadrul unei facultati. Design pattern-urile folosite usureaza functionarea aplicatiei. Prin intermediul Singleton-ului utilizăm mai ușor lista de studenti, avem o singură instnață ce suferă modificări, putem să vizualizăm istoricul studentilor datorită implementării Memento și vom putea notifica intefața prin Observer. De asemeanea, folosind Adapter putem să folosim ușor clasa Stuent în interfața ce folosește niște funcționalități specifice Secretariatului. Folosind Chain of Responsibility se afiseaza programul de la secretariat. Strategy ajuta la definirea comportamentelor legate de timpul pentru a munci al unui student in functie de tipul acestuia (licenta, master sau doctorat). Cu ajutorul Simple factory-ului se creeaza trei tipuri de student care au multe attribute comuna, iar cu ajutorul builder-ului se creeaza studenti doar cu numarul matricol, nume si prenume atunci cand nu sunt necesare celelalte date (de exemplu afisarea unor liste cu studentii reprezentati pe sali alfabetic).

Metodele implementate ajută la obținerea informațiilor despre studenti, la gestionarea listei acestora prin adăugare sau eliminare, la obtinerea datelor despre un anumit student sau realizarea modificarii asupra datelor sale.

# Bibliografie

Curs Design Patterns- <http://acs.ase.ro/Media/Default/documents/designpatterns/Curs%20CTS%20%20Design%20Patterns%20v2.pdf>

Curs Junit

<http://acs.ase.ro/Media/Default/documents/cts/curs/Curs%20CTS%20%20-%20JUnit%20v2016.pdf>

Curs intro Git

<http://acs.ase.ro/Media/Default/documents/cts/curs/Curs%20CTS%20%20-%20GIT.pdf>

Java Design Patterns – Example Tutorial

http://www.journaldev.com/1827/java-design-patterns-example-tutorial