Academia de Studii Economice București

Facultatea de Cibernetică, Statistică și Informatică Economică

Proiect

Calitate și testare software

**Profesor:** **Student:**

Asist. Alin Zamfiroiu Stama Ioana-Monica

Grupa 1055

Seria B

Cuprins

[Introducere 3](#_Toc419808630)

[Design patternuri utilizate și detalierea lor 3](#_Toc419808631)

[Definirea si detalierea metodelor testate prin Unit Testing 5](#_Toc419808632)

[Definirea si descrierea Test Case-urilor și Test Suite-urilori 7](#_Toc419808633)

[Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate 10](#_Toc419808634)

[Bibliografie 11](#_Toc419808635)

# Introducere

Proiectul este implementat în Java folsind mediul Eclipse. Acesta este o aplicație pentru gestiunea angajatilor unei firme.

În inplementarea proiectului s-au folosit 4 design patternuri respectiv: Singletone, Memento, Observer și Adapter.

# Design patternuri utilizate și detalierea lor

1.Singletone

Singletone este un design pattern creațional utilizat pentru crearea unei singure instanțe pentru o clasă prin care să fie gestionată o resursă în mod centralizat. Folosind design patternul Singletone, soluția se bazează pe existența unei singure instanțe ce poate fi creata o singură dată dar care poate fi referită de mai multe ori și asigură un singur punct de acces, vizibil global, la unica instanță creată.

Accesul la instanță este dat prin metoda unică getInstance() ce crează instanța la primul său apel, iar mai apoi, la apelurile următoare returnează instanța create anterior cu toate modificările suferite ulterior.

Avantajele utilizării acestui design pattern este că centralizează gestiunea unei resurse prin intermediul instanței unice și oferă un control strict la instanțierii unei clase, deoarece aceasta este instanțiată o singură data. De asemenea utilizarea Singletone nu permite duplicarea instanțelor.

Dar, cu toate că se lucrează cu o singură instanșp, în mult-threading pot apărea problem de sincronizare sau cooperare dacă singleton-ul este partajat și poate fi afectată performanța aplicației, acest lucrur reprezentând un dezavantaj al design pattern-ului.

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

2.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 angajați. Acest design pattern va fi utilizat atunci când se va crea o interfață în cadrul aplicației pentru a afișa constant numărul de angajați din firmă.

3. Memento

Memento este un design pattern comportamental utilizat pentru aplicații care trebuie s permit 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 angajații firmei și a permite departamentului de Resurse Umane să aibă acces la aceste schimbări. Pe baza acestei liste noi de angajți modificați, angajații pot fi readuși la starea lor inițială în cazul în care are loc o schimbare de departament, salariu etc.

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.

Componentele adapterului sunt:

* Clasa existent- ce trebuie adaptata la o noua interfață;
* ClasaContextNou ce definește interfața specifică noului domeniu;
* Adaptor –cel care adaptează interfața clasei existente la cea a clasei din noul context și care contine o referinta catre clasa sau obiectul ce trebuie adaptat
* Client -reprezintă framework-ul care apelează interfața specifică noului domeniu

Marele avantaj al acestui design pattern este că nu sunt modificare clasele existente ci doar se adaugă n layer intermediar. Adpatoarele se pot define cu ușurință pentru orice context.

Deoarece în java nu este permisă derviarea multiplă, ce stă la baza creării adaptorului, soluția alternative este formarea adaptorului prin interfeți și compunere.

În aplciația mea, a fost creat un adaptor pentru Angajat ce vor permite utilizarea aceste clase într-o nouă interfață ce va crea rapoarte pentru Casa Națională de Asigurări.

# Definirea si detalierea metodelor testate prin Unit Testing

În cadrul clasei Angajat sunt testate 3 metode: bonus(), impozit(), comision().

Metoda bonus() returneză un float, reprezentat de bonusul pe care îl primește angajatul în funcție de tipul său de program de lucru: Full Time, Part Time, Intern etc.

Metoda impozit() calculează impozitul ce trebuie plătit pentru fiecare angajat. Se ia în considerare faptul că cei din departamentul IT sunt scutiți de impozitul pe venit și plătesc doar contribuțiile la sănătate, pensie și șomaj.

Metoda comision() este utilizată pentru calcularea comisionului primit de angajați. Doar cei din departamentul vânzări primesc comision.

În cadrul clasei Firma sunt testate metodele definite și detaliate mai jos

* **Firma(ArrayList<Angajat> listaNoua)**-metodă ce crează o firmă pe baza unei liste primate ca parametru și verifică dacă lista primită este goală sau nulă
* **public void setListaAngajati(ArrayList<Angajat> list)-**metodă ce setează lista din cadrul firmei cu o listă primită ca parametru și verifică dacă lista primită este goală sau nulă
* **public Angajat getAngajat(Angajat a)** - metodă ce returnează un anumit Angajat din cadrul firmei. Este parcursă lista de angajați și e returnat angajatul dacă este găsit. Dacă nu există se returnează null, iar dacă lista e nulă sau goală se aruncă excepții.
* **public void adaugaAngajat(Angajat a)** – metodă ce adaugă un angajat în listă. Întâi se verifică pe baza cnp-ului a angajatul să nu existe deja. După adăugare se notifică observării. Dacă lista este nulă se aruncă o excepție.
* **public void eliminaAngajat(Angajat a)-**metodă ce elimină un angajat dacă acesta există. Se verifică dacă lista este nulă și dacă da, se aruncă o excepție.
* **public Angajat celMaiBinePlatit()** –metodă ce returnează cel mai bine plătit angajat parcurgând lista de angajați și comparând salariile acestora. Dacă lista e nulă sau goală se aruncă excepții.
* **public float costSalarial()** – metodă ce returnează costul slarial al firmei ce include salariul și bonusul angajaților. Se parcurge lista și se adnă salaril și bonusul fiecărui angajat. Dacă lista este goală se retunrează 0, iar dacă lista e nulă se aruncă excepții.
* **public float calculeazaImpozit()** – calculează impozitul plătit de firmă, prin însumarea impozitului specific fiecărui angajat. Dacă lista e goală se returnează 0 iar dacă lista este nulă se aruncă excepții.
* **public float calculeazaComision()** – calculează comisionul plătit de firmă către angajați adunând comisioanele tuturor angajaților. Dacă lista e goală se returnează 0 iar dacă lista este nulă se aruncă excepții.
* **public float calculeazaPlatiTotale()** – calculează totalul plăților efectuate de firmă adunând costul salarial, impozitul și comisioanele. Dacă lista e nulă aruncă excepții iar dacă este goală returnează 0.
* **public void modificaSalariu(Angajat a, int s)** – metodă ce modifică salariul unui angajat primit ca paramtru cu un slaariu primit tot ca parametru. Parcurgem lista și când găsim angajatul îniante de a-I modifica salariul îi salvăm starea și o adăugăm într-o listă, apoi îi modificăm salariul.
* **public void citireAngDinFisier()** – metodă ce adaugă în lista de angajați, angajați citiți din fișier.
* **public int numarFemei()-** metodă ce returnează numărul de femei angajate în firmă prin parcurgerea listei și adunarea tuturor femeior din companie. Acestea sun identificate pe baza cnp-ului, Dacă lista e goală se returnează 0 iar dacă lista este nulă se aruncă excepții.
* **public int numarBarbati()**- metodă ce returnează numărul de bărbați angajați în firmă similar cu metoda ce calculează numărul de femei din firmă. Dacă lista e goală se returnează 0 iar dacă lista este nulă se aruncă excepții.
* **public int numarSalariati(int s)**- metodă ce afișează angajații ce primesc un salariu mai mare decât un parametru dat. Se parcurge lista de angajați și de fiecare data când un angajat are salariul mai mare decât parametrul dat numărul crește cu o unitate. La final se returnează numărul final. Dacă lista e goală se returnează 0 iar dacă lista este nulă se aruncă excepții.

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

1. AllTestsAngajat

Acest Test Suite include cele 3 Test Case-uri utilizate pentur testarea metodelor clasei Angajat. La rularea acestuia se testează metodele clasei Angajat prin cele 10 teste implementate.

Test Case-ul testBonus testează metoda bonus() a clasei Angajat. Cele 4 teste testează inișial cazul general în se determină statusul angajatului și se calculează bonusul. S-a calculat separat bonusul și s-au comparat cele două valori.

Celelalate 3 teste testează fiecare ramură a metodei bonus(): angajatul e FullTime, angajatul e PartTime sau angajatul e Intern.

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

Test Case-ul testComision testează metoda comision() a clasei Angajat. În cadrul acestuia s-au realizat 3 teste pentru această metodă. Unul testează cazul general de utilizare la metodei și celălalt testează metoda dacă angajatl nu face parte din departamentul de vâznări sau dacă angajatul face parte din acesta.

Test Case-ul testImpozit testează metoda impozit(). Se testează mai întâi comparând rezultatul cu unul calculate anterior, apoi se testează implementând calculul în cadrul testului.

1. AllTestsFirma

Test Suite-ul AllTestsFirma include cele 15 Test Case-uri implementate. Acesta testează metodele clasei firmă folosind cele 49 de teste din Test Case-uri.

Mai jos, sunt prezentate și descries pes curt Test Case-urile pentru clasa Firma.

* TestareCalculeazaComision testează metoda de calcul a comisionului din clasa Firma. Mai întâi este testate metoda folosind o listă de angajați, iar rezultatul este comparat cu un număr calculate anterior folosind metoda assertEquals(). Se mai testează metoda și în cazul în care lista primită ca parametru e goală sau nulă.
* Prin TestareCalculeazaCostTotal se tstează metoda de calcul a costurilor toate ale firmei. Primul test tstează metoda comparând rezultatul cu un număr calculate anterior, iar apoi este testate metoda și pentru cazul ăn care lista este goală și trebuie să se returneze 0 și pentru cazul în care lista este nulă și trebuie să se arunce o excepție.
* Test Case-ul TestareCalculeazaImpozit conține teste pentru metoda calculeazaImpozit() din clasa firmă. Se testează cazurile când lista este goală și metoda trebuie să atunce excepții sau când lista este goală și metoda trebuie să returneze 0. Pentru testarea metodei când primește o listă populate se compară valoarea returnată de metodă cu una calculată anterior.
* TestareCelMaiBinePlatit este Test Case-ul utilizat pentru testarea metodei celMaiBinePlătit() din clasa Firmă. Primul test crează o listă cu 2 angajați și o Firmă 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.

* TestareCitireFișier testează metoda prin care se adaugă angajați în lista firmei cu date preluate dintr-un fișier. Pentru testare se face un fișier de test și se citesc de acolo datele. Se crează un angajat cu aceleași date și se compară rezultatele. În cazul în care fișierul nu există sau nu se paote deschide se aruncă excepții.
* Test Case-ul TestareCostSalarial conține teste pentru metoda costSalarial() din clasa Firma. Se testează cazurile când lista este goală și metoda trebuie să atunce excepții sau când lista este goală și metoda trebuie să returneze 0. Pentru testarea metodei când primește o listă populată se compară valoarea returnată de metodă cu una calculată anterior.
* TestareEliminaAngajat testează metoda eliminaAngajat(). Se fac teste pentru cazurile în care lista firmei este nulă sau goală și metoda trebuie să arunce excepții. Se mai face un test pentru eliminare unui angajat care nu există când metoda trebuie să returneze null și pentu eliminarea unui angajat deja existent. Pentru testare eliminării se comapră dimensiunile listelor folosite. Inițiaș s-a creat o listă cu angajați, se crează o firmă folosind lista respectivă apoi se elimină folosind metoda clasei angajatul din lista firmei.
* TestareFirmaCreare este Test Case-ul prin care se testează metodele de creare a listei firmei. Inițial se testează metoda setListaAngajati(ArrayList<Angajat> list). Testăm dacă lista firmei este identică cu cea primită ca parametru, apoi se testeză cazurile pentru null și not null. În cazul metodei Firma(ArrayList<Angajat> listaNoua) se testează dacă listele corespund, și se testează și null sau not null.
* TestareGetAngajat testează metoda getAngajat(). Se testează returnarea unui angajat existent și ne null, cazurile în care lista este nulă sau goală și trebuie aruncate excepții și cazul în care angajatul nu există și trebuie să returnăm null.
* Test Case-ul TestareModificaSalariu testează metoda de modificare a salariului. 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 angajat null, caz în care iar se aruncă excepții. Și se testază modificare salariului unui angajat existent . Testul compară valoarea cu care se dorește modificarea salariului cu salariul obținut în urma modificării.
* TestFirmaAdaugareAngajat testează metoda de adăugare a angajatului în firmă. În cazul în care lista este nulă se aruncă excepții.
* testNrAngajati testează metoda prin care se întoarce numărul de angajați al căror salariu depășește parametrul dat. Se testează cazul în care lista e nulă și trbeuie să se arunce excepții și cazul în care lista e goală și trebuie să se returneze 0. Se mai face un test pentru a testa metoda și se compară valoarea obținută cu una prestabilită
* Test Case-ul testNrBarbați testează metoda prin care este returant numărul de bărbați din companie. Se testează cazurile în care lista e nulă și se aruncă excepții și cauzl în care lista e goală și se returnează 0. Se testează metoda și prin compararea valorii returnate cu unnumăr dat.
* testNrFemei este Test Case-ul prin care se testează metoda prin care este returnat numărul de femei din companie. 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 angajaților din cadrul unei companii. Design pattern-urile folosite usureaza functionarea aplicatiei. Prin intermediul Singleton-ului utilizăm mai ușor lista de angajați, avem o singură instnață ce suferă modificări, putem să vizualizăm istoricul angajaților datorită implementării Memento și vom putea notifica intefața prin Oserver. De asemeanea, folosind Adapter putem să folosim ușor clasa Angajat în interfața ce folosește niște funcționalități specifice CNAS.

Metodele implementate ne ajută să obținem informații despre angajați, să gestionăm lista acestora prin adăugare sau eliminare, să obținem date despre un anumit angajat, să realziăm modificări asupra lui și să extragem informații despre angajații firmei etc.

# Bibliografie

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

Suport seminar