Proiect   
  
 Calitate si Testare Software

Informatica Economica, anul III Suplimentar

Manea Daniel Gabriel

CUPRINS

[1. Introducere 3](#_Toc420596411)

[2. Definirea si detalierea pattern-urilor utilizate in proiect 5](#_Toc420596412)

[3. Definirea si detalierea metodelor testate prin Unit Testing si Test Case-uri 10](#_Toc420596413)

[4. Definirea si descrierea Test suite-ului 15](#_Toc420596414)

[5. Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate 17](#_Toc420596415)

[6. Bibliografie 18](#_Toc420596416)

# 1. Introducere

În ingineria software, un model de design (design pattern) este o soluție ce se repeta in general la o problemă ce apare in mod frecvent intr-un model software. Un model de design nu este un design terminat care poate fi transformat direct în cod. Este o descriere sau un șablon pentru modul de a rezolva o problemă care poate fi folosita în multe situații diferite.  
  
**Moduri de utilizare pentru Design Patterns**

Modelele de design pot accelera procesul de dezvoltare prin furnizarea de teste, paradigme de dezvoltare dovedite. Software-urile de proiectare eficientă necesita luarea in calcul a faptului ca ele nu ar putea deveni vizibile decat mai tarziu, in punerea in aplicare. Reutilizarea modelelor de design ajută la prevenirea problemelor subtile care pot cauza probleme majore și îmbunătățește lizibilitatea codului pentru programatori și arhitecți familiarizati cu modelele.

De multe ori, oamenii înțeleg doar cum să aplice anumite tehnici de proiectare software pentru anumite probleme. Aceste tehnici sunt dificil de aplicat la o gamă mai largă de probleme. Modelele de design oferă soluții generale, documentate într-un format care nu are nevoie de specificul legat de o anumită problemă.

În plus, modelele permit dezvoltatorilor să comunice folosind nume consacrate si interacțiuni software consacrate. Modelele de design comune pot fi îmbunătățite în timp, ceea ce le face mai robuste decat modelele ad-hoc.  
  
**Tipuri de modele de design**  
  
**Modele de design de creare**

In aceasta proiectare de modele este vorba despre instantierea claselor. Acest model poate fi împărțit în modele de clasă de creare și modele obiect de creare. În timp ce modelele de clasă de creare folosesc moștenire în mod eficient în procesul de instanțiere, modelele obiect de creare folosesc delegarea eficient pentru a face lucrurile asa cum trebuie.  
  
**Modele de design structural**

In aceasta proiectare de modele este vorba de clasa și compoziția obiect. Modelele de design structural de tip clasa de creare folosesc moștenirea pentru a compune interfețe. Modelele de design structural de tip obiect definesc moduri de a compune obiecte pentru a obține noi funcționalități.  
  
**Modele de design de comportament**

In aceasta proiectare de modele este vorba despre comunicarea intre obiectele claselor. Modelele de comportament sunt acele modele care sunt implicate mai specific cu comunicarea între obiecte.

# 2. Definirea si detalierea pattern-urilor utilizate in proiect

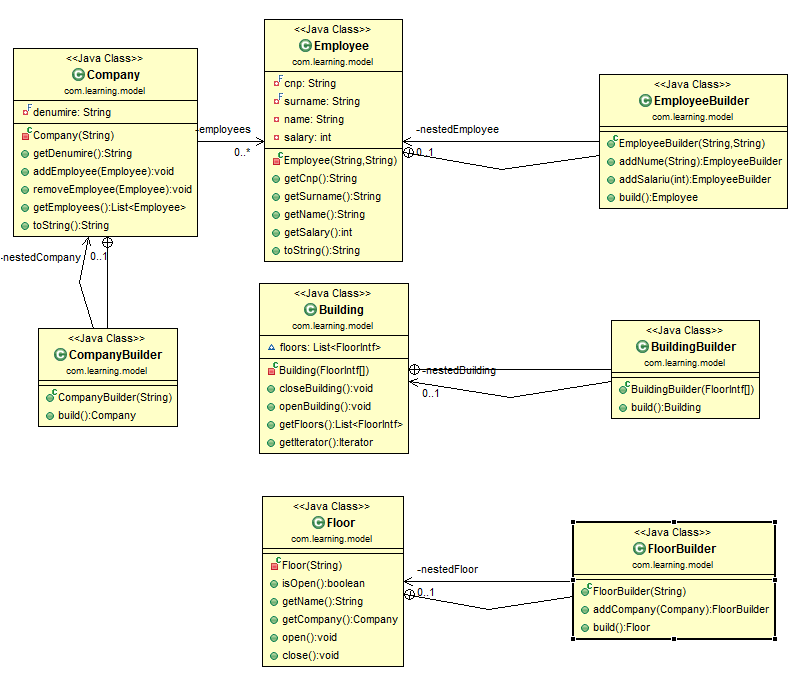
**Builder**

Modelul Builder construiește un obiect complex, folosind obiecte simple și folosind o abordare pas cu pas. Acest tip de model de proiectare se încadrează în modelul de creație, astfel incat acest model oferă unul dintre cele mai bune moduri de a crea un obiect.

O clasa Builder construiește pas cu pas obiectul final. Acest constructor este independent de alte obiecte.

In aplicatia creata in cadrul acestui proiect am utilizat acest design pattern pentru a crea obiecte de tip employee, floor, company si building. In fiecare dintre aceste clase x exista o clasa private xBuilder care va furniza un nou obiect de tip x, in timp ce constructorul clasei x este privat.

Schema de mai jos prezina relatia UML dintre clasele “model” din cadrul proiectului unde a fost folosit design patternul Builder.

  
**Facade**

Modelul Facade ascunde complexitatea sistemului și oferă o interfață pentru client, astfel incat el sa poata accesa sistemul. Acest tip de model de proiectare se încadrează în modelul structural, adaugand o interfață de sistem existent pentru a ascunde complexitatea acestuia.

Acest model presupune o singură clasă care oferă metode simplificate cerute de client și apeluri delegate catre metodele claselor sistemului existent.

In cadrul acestui proiect avem doua tipuri de etaje (Floors). Etajul – parter si etaj – normal. Fiecare etaj se inchide si se deschide diferit. De exemplu am considerat ca etajul – parter sa aiba in plus proprietatea de a fi armat. Dar aceasta functionalitate nu vrem sa ne complice functionalitatea. Astfel am creat interfata FloorIntf care contine trei metode: open(), close() si isOpen(); Clasele Floor si GroundFloor implementeaza aceasta interfata indirect prin clasa abstracta AFloor. Am creat aceasta clasa abstracta pentru a crea atributele comune celor doua tipuri de etaje. Pentru ca aceste 2 clase implementeaza interfata FloorInft fiecare dintre ele isi declara functionalitatea specifica pentru aceste actiuni. Mai departe consideram ca o cladire contine un obiect de tip GroundFloor si n obiecte de Floor. Am creat o actiune de inchidere a cladirii care parcurge cu ajutorul interfetei comune toate etajele si le apeleaza metoda de inchidere. Mai jos este prezentata aceasta metoda:

**public** **void** closeBuilding(){

**for** (FloorIntf arg : **this**.floors) {

arg.close();

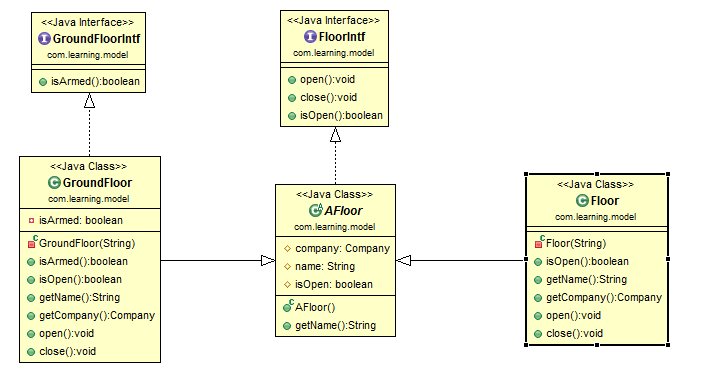
}

}

Observam ca scopul utilizarii metodei design pattern Facade a fost indeplinit cu succes pentru ca in aceasta clasa nu este specificat tipul de etaj, avem o lista de etaje si fiecarui etaj ii este apelata metoda de inchidere. Fiecare etaj se inchide in felul lui. Mai jos avem rezultatul acestei actiuni si schema design pattern-ului Facade:

Close ground floor

Close normal floor

Close normal floor  
  
  
**Iterator**

Modelul Iterator este foarte frecvent utilizat ca model de proiectare în Java și in mediul de programare .NET. Acest model este utilizat ca o cale de acces catre elementele unei colectii de obiecte în mod secvențial, fără a fi nevoie să se știe reprezentarea de bază. Design pattern-ul Iterator nu este folosit numai pentru a traversa colectii, pot fi proiectate diferite tipuri de Iteratoare in functie de necesitati.

Modelul Iterator se încadrează în categoria de model comportamental. In proiectul nostru am folosit acest design pattern pentru a parcurge lista de etaje continute intr-o cladire.

Pentru a crea acest design pattern, in primul rand este nevoie de o interfata “Iterator” in care sunt declarate doua metode hasNext() si next().

**public** **interface** Iterator {

**public** **boolean** hasNext();

**public** Object next();

}

Mai departe avem nevoie de o clasa care sa implementeze aceasta interfata. Am creat o clasa privata FloorIterator in clasa Building prin care obtinem functionalitatea dorita si o metoda prin care obtinem un nou obiect de tipul acestei clase.

**public** **class** Building **implements** FloorContainer{ //facade //builder //Iterator

List<FloorIntf> floors;

**/\* Lista de metode, attribute si alte clase private \*/**

@Override

**public** Iterator getIterator() {

**return** **new** FloorIterator();

}

**private** **class** FloorIterator **implements** Iterator {

**int** index;

@Override

**public** **boolean** hasNext() {

**if**(index < floors.size()){

**return** **true**;

}

**return** **false**;

}

@Override

**public** Object next() {

**if**(**this**.hasNext()){

**return** floors.get(index++);

}

**return** **null**;

}

}

}

Ultima unitate de care avem nevoie este o interfata pe care am denumit-o FloorContainer in care declaram metoda getIterator folosita anterior.

Mai jos avem utilizarea design pattern-ului Iterator:

**public** **void** buildingIterator(){

GroundFloor floor0 = **new** GroundFloorBuilder("Et0").build();

Floor floor1 = **new** FloorBuilder("Et1").build();

Floor floor2 = **new** FloorBuilder("Et2").build();

Floor floor3 = **new** FloorBuilder("Et3").build();

Building bld = **new** BuildingBuilder(floor0,floor1,floor2,floor3).build();

**int** i = 0;

//Iterator

**for**(Iterator iter = bld.getIterator(); iter.hasNext();){

AFloor floor = (AFloor)iter.next();

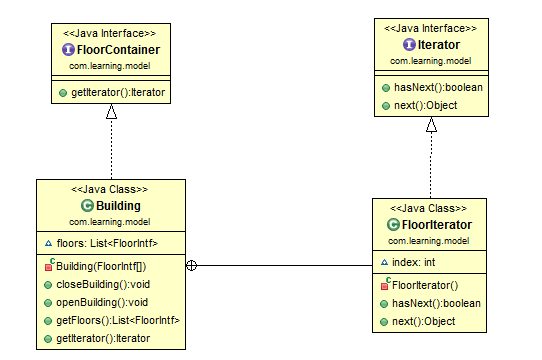
System.***out***.println("Name : " + floor.getName());

i++;

}

}

Schema UML a design pattern-ului Iterator:



# 3. Definirea si detalierea metodelor testate prin Unit Testing si Test Case-uri

**1. Metoda testEmployeeCreate**  
  
Aceasta metoda verifica daca s-a creat cu success un obiect de tip Employee prin design pattern - Builder. Obiectul care va detine informatia noului angajat este null la inceput (nu face referire catre nicio locatie de memorie). Dupa ce transmitem datele pentru a fi creat un nou angajat cu informatii despre: nume, prenume, CNP si salariu, executam metoda assertNotNull pentru a testa daca noile informatii sunt stocate sub obiectul nou creat.

@Test

**public** **void** testEmployeeCreate() {

Employee employee = **null**;

*assertNull*(employee);

employee = **new** EmployeeBuilder("1900101334444","Adrian")

.addNume("Popescu")

.addSalariu(543)

.build();

*assertNotNull*(employee);

}  
  
**2. Metoda testEmployeeAttributes**

Aceasta metoda verifica daca unui obiect de tip employee i se atribuie corect atributele. Se creaza un nou obiect de tip employee prin metoda design pattern – Builder apoi testam daca atributele sunt cele corecte.

@Test

**public** **void** testEmployeeSalary() {

Employee employee = **new** EmployeeBuilder("1900101334444","Adrian")

.addNume("Popescu").addSalariu(543).build();

*assertEquals*(("Popescu", employee.getName());

*assertEquals*("Adrian", employee.getSurname());

*assertEquals*(543, employee.getSalary());

*assertEquals*("1900101334444", employee.getCnp());

}

**3. Metoda testRiseEmployeeSalary**

Aceasta metoda verifica daca la un obiect existent de tip employee, atunci cand se doreste marirea de salariu, aceasta marire este executata corect. Prima metoda va fi executata corect iar in cea de-a doua metoda vom trimite ca parametru o valoare null in locul unui obiect de tip employee si de aceasta data verificam daca se transmite exceptia.

@Test

**public** **void** testRiseEmployeeSalary(){

Employee employee = **new** EmployeeBuilder("1900101334444","Adrian")

.addNume("Popescu").addSalariu(543).build();

EmployeeService e = **new** EmployeeService();

e.riseSalary(employee, 100);

*assertEquals*(643, employee.getSalary());

}

@Test

**public** **void** testRiseEmployeeSalaryNull(){

**boolean** thrown = **false**;

EmployeeService e = **new** EmployeeService();

**try**{

e.riseSalary(**null**, 100);

}**catch**(NullPointerException ex){

thrown = **true**;

}

*assertTrue*(thrown);

}

**4. Metoda testFloorCreate**  
  
Aceasta metoda verifica daca se creeaza cu succes un etaj. Tot prin metoda Builder se creeaza un etaj (Floor) caruia ii este specificat numele (sau denumirea etajului. Ex: Et1). Aplicam metoda assertNotNull pentru a verifica daca avem un nou obiect creat.

@Test

**public** **void** testFloorCreate() {

Floor floor = **null**;

*assertNull*(floor);

floor = **new** FloorBuilder("Et1").build();

*assertNotNull*(floor);

}

**5. Metoda testFloorName**  
  
Aceasta metoda verifica daca etajului i s-a adaugat numele corect. Se creaza un etaj nou, dupa care cu ajutorul functiei assertEquals se verifica daca numele lui este cel corect.

@Test

**public** **void** testFloorName() {

Floor floor = **new** FloorBuilder("Et1").build();

*assertEquals*("Et1", floor.getName());

}  
  
**6. Metoda testAddCompany**  
  
Aceasta metoda verifica adaugarea unei companii noi. Se creaza o companie noua, dupa care se verifica daca numele ei este cel corect si lista cu angajati este 0.

@Test

**public** **void** testAddCompany(){

Company cmp = **new** CompanyBuilder("ORACLE").build();

*assertEquals*("ORACLE", cmp.getDenumire());

*assertEquals*(0, cmp.getEmployees().size());

}

**7. Metoda testEmployeeList**  
  
Aceasta metoda verifica daca lista angajatilor este cea corecta si corespunde companiei respective. Se creaza o companie noua, se verifica daca numele ei este cel corect, apoi sunt creati angajatii, adaugati intr-un ArrayList, dupa care se compara pentru a vedea daca cele doua liste (cea a companiei si cea separata) coincid. Acest test il vom face cu metoda assertArrayEquals.

In pasul 2 vom elimina din lista un angajat si testam daca cele doua s-au modificat.

@Test

**public** **void** testEmployeeList(){

Company cmp = **new** CompanyBuilder("ORACLE").build();

*assertEquals*("ORACLE", cmp.getDenumire());

*assertEquals*(0, cmp.getEmployees().size());

Employee employee = **new** EmployeeBuilder("1900101334444","Adrian")

.addNume("Popescu")

.addSalariu(543).build();

Employee employee1 = **new** EmployeeBuilder("1911010101001","Gelu")

.addNume("Irimescu")

.addSalariu(1310).build();

Employee employee2 = **new** EmployeeBuilder("1932132132111","Andrei")

.addNume("Georgescu")

.addSalariu(2543).build();

List<Employee> employees = **new** ArrayList<Employee>();

employees.add(employee);

cmp.addEmployee(employee);

employees.add(employee1);

cmp.addEmployee(employee1);

employees.add(employee2);

cmp.addEmployee(employee2);

*assertArrayEquals*(employees.toArray(), cmp.getEmployees().toArray());

employees.remove(employee2);

*assertNotEquals*(employees.size(), cmp.getEmployees().size());

}  
  
  
**8. Metoda testBuildingOpenClose**  
  
Aceasta metoda verifica daca o cladire se deschide (se deschid toate etajele indiferent de tipul acestora), respectiv se inchide. Se creaza etajele, apoi o cladire noua care contine aceste etaje. Apelam inchiderea cladirii prin metoda .closeBuilding(). Aceasta metoda va inchide toate etajele, indiferent de tipul acestora. Vom parcurge lista de etaje si vom testa daca acestea sunt inchise. Fiecare floor(etaj) detine un atribut isOpen de tip boolean. Asemanator modului de inchidere al cladirii avem si modul de deschidere. Aceste operatiuni se realizeaza cu ajutorul modelului Facade. Fiind creata interfata comuna celor doua tipuri de etaj (GroundFloor, Floor) putem apela aceeasi metoda pentru doua tipuri de obiect diferite.

@Test

**public** **void** testBuildingOpenClose() {

GroundFloor floor0 = **null**;

floor0 = **new** GroundFloorBuilder("Et0").build();

Floor floor1 = **null**;

floor1 = **new** FloorBuilder("Et1").build();

Floor floor2 = **null**;

floor2 = **new** FloorBuilder("Et2").build();

Building bld = **new** BuildingBuilder(floor0,floor1,floor2).build();

//facade

bld.closeBuilding();

**for** (FloorIntf fl : bld.getFloors()){

*assertFalse*(fl.isOpen());

}

//facade

bld.openBuilding();

**for** (FloorIntf fl : bld.getFloors()){

*assertTrue*(fl.isOpen());

}

}

**9. Metoda testBuildingIterator**  
  
Prin aceasta metoda dorim sa testam corectitudinea modelului design pattern Iterator. Astfel vom crea 3 etaje de tip Floor si un etaj de tip GroundFloor. Acestor etaje le atribuim un obiect de tip Building la creare si apoi cu ajutorul Iteratorului testam daca au fost corect adaugate.

@Test

**public** **void** testBuildingIterator(){

GroundFloor floor0 = **null**;

floor0 = **new** GroundFloorBuilder("Et0").build();

Floor floor1 = **null**;

floor1 = **new** FloorBuilder("Et1").build();

Floor floor2 = **null**;

floor2 = **new** FloorBuilder("Et2").build();

Floor floor3 = **null**;

floor3 = **new** FloorBuilder("Et3").build();

Building bld = **new** BuildingBuilder(floor0,floor1,floor2,floor3).build();

**int** i = 0;

//Iterator

**for**(Iterator iter = bld.getIterator(); iter.hasNext();){

AFloor floor = (AFloor)iter.next();

System.***out***.println("Name : " + floor.getName());

i++;

}

*assertEquals*(i, 4);

}  
  
**10. Metoda testFLoorAddCompany**  
  
Prin aceasta metoda dorim sa testam daca la un etaj putem sa atribuim corect o companie care isi va desfasura activitatea in acest etaj. Se creaza un etaj si i se atribuie un nume. Apoi se creaza o companie care este adaugata la acel etaj. Cu ajutorul functiei assertEquals verificam corectitudinea atribuirii prin testarea denumirii companiei care se gaseste la acel etaj.

@Test

**public** **void** testFLoorAddCompany(){

Floor floor1 = **null**;

floor1 = **new** FloorBuilder("Et1").build();

*assertNull*(floor1.getCompany());

Company company = **new** CompanyBuilder("IBM").build();

floor1 = **new** FloorBuilder("Et1").addCompany(company).build();

*assertEquals*("IBM",floor1.getCompany().getDenumire());

}

# 4. Definirea si descrierea Test suite-ului

Un test suite este o grupare de cel putin 2 clase de test care vor actiona impreuna. Pentru fiecare din aceste clase care compun test suite se executa toate metodele de test. Daca o metoda din aceste clase are eroare, atunci si test suite va fi executat cu eroare. Pentru ca test suite sa fie executat fara erori trebuie ca toate metodele continute in clasele ce compun test suite sa se execute cu succes.

In JUnit sunt folosite adnotatiile:

@RunWith

@Suite

pentru a declara o suita de test (suite test).

In primul rand este nevoie de aceste clase care vor fi executate impreuna. In acest proiect am creat patru clase: EmployeeTest.java, CompanyTest, FloorTest si BuildingTest.java.

Fiecare clasa de test are rolul de a testa corectitudinea functionarii corecte a actiunilor referitoare la obiectul corespunzator.

Am creat o noua clasa JUnitTestSuite si am atasat adnotatia @RunWith(Suite.class) pentru a specifica ca aceasta este o clasa TestSuite. Am mai atasat adnotatia @Suite.SuiteClasses(…) pentru a specifica lista de clase care vor fi executate impreuna (in cazul de fata EmployeeTest.class, CompanyTest, FloorTest si BuildingTest.class).

**package test;**

**import org.junit.runner.RunWith;**

**import org.junit.runners.Suite;**

**@RunWith(Suite.class)**

**@Suite.SuiteClasses({**

**EmployeeTest.class,**

**BuildingTest.class,**

**CompanyTest.class,**

**FloorTest.class**

**})**

**public class JUnitTestSuite {**

**}**

In metoda main am creat componenta care va rula aceste clase astfel:

Obiectul de tip Result va gestiona executia acestor clase si este creat cu ajutorul metodei runClasses(TestSuite) din clasa JUnitCore.

Acestei metode ii este furnizata ca parametru clasa TestSuite.

Dupa executia metodei main(), pe ecran a fost furnizat mesajul:

Test Suite a rulat cu success!

**package** com.learning;

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**import** test.JUnitTestSuite;

**public** **class** MyMain {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.*runClasses*(JUnitTestSuite.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful() ? "Test Suite a rulat cu success!" : "Test Suite a avut erori!");

}

}

In clasa BuildingTest.class am schimbat, in metoda testBuildingIterator(), valoarea 4 (care reprezinta numarul de etaje adaugate).

De exemplu am pus valoarea 0.

*assertEquals*(i, 4); \\ cod vechi

*assertEquals*(i, 0); \\ cod nou

Acest lucru ii spune compilatorului ca ne asteptam sa nu avem nici un etaj adaugat.

La executia programului obtinem eroarea si mesajul de eroare:

testBuildingIterator(test.BuildingTest): expected:<4> but was:<0>

Test Suite a avut erori!

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

Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate a fost facuta in cadrul capitolelor anterioare.

# 6. Bibliografie

<http://www.tutorialspoint.com/design_pattern/design_pattern_tutorial.pdf>  
[Design Patterns Facade Pattern](http://www.tutorialspoint.com/design_pattern/facade_pattern.htm)  
[Design Patterns and Refactoring](https://sourcemaking.com/design_patterns)