**Academia de Studii Economice, Facultatea de Cibernetica, Statistica si Informatica Economica, 2016**

**Proiect Calitate si Testare Software**

**Nume: Matei Ana-Maria**

**Grupa: 1070**

**Seria: B**

**An: 3**

1. **Definirea si justificarea pattern-urilor implementate**

In cadrul acestui proiect s-au folosit urmatoarele design pattern-uri:

* Singleton
* Simple Factory
* Builder
* Memento
* Decorator
* Observer

**Singleton:**

Am ales sa folosesc Singleton deoarece compania este unica motiv pentru care am considerat necesara restrictionarea numarului de instantieri ale clasei Companie la un singur obiect. Singleton-ul permite instantierea intarzaiata a obiectului si utilizeaza memoria doar daca este necesar deoarece instanta se creeaza prin apelul metodei getInstance().

Clasa folosita pentru implementarea Singleton-ului este: Companie.

**Simple Factory:**

Deoarece lucrez cu din familia aeronavelor(aeronava cargo si aeronava pasageri) am decis sa implementez Simple Factory. Factory reprezinta un mecanism de creare de obiecte dintr-un singur loc. Crearea obiectelor de tip pasageri si cargo se face transparenta pentru client, dar acesta nu vede cum sunt create obiectele.

In implementarea modelului am folosit:

- o enumerare: TipAeronava(CARGO, PASAGERI)

- o clasa abstracta: InterfataAbstractaAeronava, in care am urmatoarele atribute: idAeronava, nrLocuriEconomy, nrLocuriBusiness, greutateMaxima si o metoda abstracta: calculeazaCostTotal() care returneaza suma totala obtinuta din vanzarea de bilete pentru aeronava respectiva. Tot in aceasta clasa am implementat si get si set pentru atributele clasei si metoda de afisare a informatiilor despre aeronava.

Clasele utilizate:

- clasa AeronavaPasageri, cu atributele: PRET\_ECONOMY, PRET\_BUSINESS.

- clasa AeronavaCargo, cu atributele: TARIF\_UNITAR, NIVEL\_RISC.

Atat AeronavaCargo, cat si AeronavaPasageri extind clasa InterfataAbstractaAeronava

-clasa AeronavaFactory foloseste atribut de tip InterfataAbstractaAeronava si creaza obiecte de tip AeronavaPasageri sau AeronavaCargo.

**Builder:**

Design Patter-un Builder reprezinta o modalitate de creare incrementala de obiecte complexe folosindu-se de un mecanism total independent de procesul de realizare a obiectelor.

Intrucat clasa Comanda are atat de multe atribute si deoarece unele dintre ele pot fi ignorate(oraSosire, bagajCala), am decis sa implementez un Builder pentru construirea obiectelor de tip Comanda.

Clasele utilizate:

-clasa abstracta InterfataAbstractaComanda, care contine metoda build() care returneaza Comanda.

-clasa Comanda cu metode get si set pentru toate atributele.

- clasa ComandaBuider care extinde InterfataAbstractaComanda si utilizeaza instante ale Comanda.

**Memento:**

Design Pattern-ul Memento permite restaurarea starilor anterioare ale unui obiect.

La un moment dat, un angajat al companiei, care este insarcinat cu verificarea comenzilor, ar putea, din greseala sa stearga o anumita comanda. Pentru a evita pierderea completa a comenzii respective am implementat un Memento care sa poata returna datele, in starea in care se aflau inainte de ultima salvare. Astfel, angajatul poate recupera comanda.

Clasele utilizate:

-ComandaMemento salveaza informatiile despre comanda si le returneaza prin apelul metodei getInformatiiSalvate().

-clasa InformatiiComanda tine o lista de obiecte de ComandaMemento. Clasa permite:

* adaugarea de noi obiecte in lista: addSalvare: ComandaMemento;
* recuperearea informatiilor aflate in lista pana in momentul ultimei salvari: recupereazaUltimaSalvare: ComandaMemento;

**Decorator:**

Decorator-ul este un design pattern care permite adaugarea de comportament pe un obiect, in mod static sau dinamic, fara sa afecteze comportamentul altor obiecte din aceeasi clasa.

Acest design pattern a fost folosit pentru a face o diferentiere a preturilor biletelor, in functie de clasa la care s-a rezervat locul din aeronavele de pasageri.

Compania are stabilite niste preturi fixe: BEBELUS: 100 lei, COPIL: 140 lei, ADULT: 200 lei, dar pretul biletelor variaza in functie de distanta cursei. Astfel, pentru clasa economy pretul variaza intre 0 si 30 lei, iar pentru clasa business, intre 0 si 70 de lei.

Din aceste motive, am creat o clasa PretBilet care returneaza preturile standard pentru fiecare tip de loc, clasa pe care am decorat-o cu ajustarile descrise mai sus.

Clasele utilizate:

-clasa abstracta AbstractPretBilet care contine metoda abstracta getPret(TipBilet tipBilet)

-clasa PretBilet, cu metoda getPret(TipBilet tipBilet)

-clasa DecoratorPretBilet care foloseste obiecte de tipul AbstractPretBilet si suprascrie metoda getPret(TipBilet tipBilet).

-clasele PretBiletBusiness si PretBiletEconomy sunt cele “decorate”. Ele extind clasa DecoratorPretBilet si prin urmare suprascriu si metoda getPret(TipBilet tipBilet).

**Observer:**

Observer este un design pattern in care un obiect numit subiect, tine o lista a obiectelor dependente de el(observatorii) si ii notifica automat in momentul aparitiei unei schimbari de stare a sa, prin apelul uneia dintre metodele observatorilor. Acest pattern este in general folosit pentru implementarea sistemelor distribuite de gestionare a evenimentelor.

In momentul in care un pasager urca in avion, insotitorii de zbor aflati in aeronava il observa. Analog si cand acesta coboara.

Clasele implementate:

-interfata HandlerCheckin cu metodele checkin() si checkout().

-clasa InsotitorDeZbor care implementeaza interfata HandlerCheckin(acestia sunt observatorii).

-clasa Pasager(obiectele de observat).

-clasa AeronavaPasageri care contine lista de observatori si lista de pasageri. In clasa AeronavaPasageri s-au implementat si metodele: abonareHandler(HandlerCheckin handler), dezabonareHandler(HandlerCheckin handler), notificareObservatoriLaCheckin() si notificareObservatoriLaCheckout(). Aceste metode sunt folosite pentru “lucrul” cu observatorii. Pe langa aceste metode, s-au implementat si metode de adaugare sau scoatere din lista a pasagerilor. Prin aceste metode au fost notificati si observatorii(am aplelat notificareObservatoriLaCheckin() si notificareObservatoriLaCheckout() in metodele adaugaPasager() si stergePasager()).

1. **Definirea si detalierea metodelor testate prin Unit Testing**

Prin Junit s-au testat urmatoarele metode:

- setarea id-ului aeronavei cu null sau un empty string

- setarea numelui companiei cu null sau empty string

- testarea listei de aeronave din clasa Companie

- testarea tabelei de dispersie din clasa Companie

- testarea numelui clientului din clasa Comanda

- testarea varstei clientului din clasa Comanda

- testarea orasului de plecare din clasa Comanda

- testarea orasului destinatie din clasa Comanda

- testarea orei de plecare din clasa Comanda

- testarea zilei de plecare din clasa Comanda

- testarea tipului de bilet ales(clasa Comanda)

-testarea clasei la care clientul va calatori(clasa Comanda)

-testarea citirii din fisier a datelor

-testare de obiecte mock

Pentru clasa abstracta InterfataAbstractaAeronava am testat metoda de tip set: setIdAeronava(). Metoda este de tip void, iar rolul ei este de a asigma id-ului aeronavei o valoare introdusa de client.

S-a testat daca pentru String-ul idAeronava pot fi setate valori aberante cum ar fi: null, empty string(“”), dar si valori corecte.

Testele Junit scrise pot fi gasite in TestCase-ul numit TestIdAeronava.

Pentru clasa Companie am teste metoda set corespunzatoare numelui companiei. Metoda este de tip void si verfica daca numele companiei poate sau nu fi setat cu valori aberante(null, “”).

Testele Junit scrise pentru numele companiei se gasesc in TestCase-ul TestNumeCompanie.

Tot pentru clasa Companie s-au mai testat si lista de aeronave si tabela de dispersie pentru comenzi. Testele s-au concentrat pe validarea dimensiunilor listei si hash-ului la adaugare si stergere de obiecte.

Testele Junit corespunzatoare acestora sunt: TestListaAeronave si TestHashMapComenzi.

Pentru clasa Comanda se testeaza mai multe metode set: setNume(), setVarsta(), setOrasPlecare(), setOrasDestinatie(), setZiPlecare(), setTipBilet(), setClasa(). Scopul este validarea atributelor astfel incat acestea sa nu ia valori aberante.

Pentru clasa ComandaBuilder s-au implementat inca doua metode: faScumpire() sau faIeftinire() folosite mai apoi pentru testarea cresterii si scaderii preturilor din PretBilet. Aceste metode sunt apelate in cadrul Test Case-ului *TestObiecteMock*.

1. **Definirea si descrierea Test Case-urilor**

*Testul TestIdAeronava:*

Rol: testarea posibilitatii setarii id-ului aeronavei pe valori aberante: null, empty string, dar si pe valori asteptate.

TestIdAeronava contine 3 teste:

* *testSetIdAeronavaNull()* care testeaza daca id-ul aeronavei poate avea valoarea null. Testul foloseste metodele *assertNotNull(aeronavaPasageri.getIdAeronava())* si *assertNotNull(aeronavaCargo.getIdAeronava())*. Testul da fail daca id-ul aeronavelor poate lua valoarea null si success in cazul in care codul a fost scris corect si id-ul a fost validat inainte de a fi setat.
* *testSetIdAeronavaEmptyString()* care testeaza daca id-ul aeronavei poate fi setat cu un string gol: “”. Testul foloseste metodele: *assertFalse(aeronavaPasageri.getIdAeronava().isEmpty())* si *assertFalse(aeronavaCargo.getIdAeronava().isEmpty()).* Testele vor da fail daca id-ul poate fi setat cu empty string si success daca id-ul a fost validat corect si respinge valorile empty string(“”).
* *testSetIdAeronavaCorect()* – testeaza daca id-ul poate fi setat cu valori de gama celor asteptate. Testul foloseste metodele: *assertEquals(myId, aeronavaPasageri.getIdAeronava())* si *assertEquals(myId, aeronavaPasageri.getIdAeronava())*, unde myId este un string dat de client. Prin folosirea acestui String myId, in locul metodei clasice(a scris in test direct valoarea la care ne asteptam) are rolul de a evita eventualele erori de scriere in moementul scrierii testului.

Testul *TestNumeCompanie*:

Rol: testarea posibilitatii setarii numelui companiei pe valori aberante: null, empty string, dar si pe valori asteptate.

TestIdAeronava contine 3 teste:

* *testSetNumeCompanieNull()* care testeaza daca numele companiei poate avea valoarea null. Testul foloseste metodele *assertNotNull(companie.getNume())* si *assertNotNull(companie.getNume())*. Testul da fail daca numele companiei poate lua valoarea null si success in cazul in care numele a fost validat inainte de a fi setat.
* *testSetNumeCompanieEmptyString()* care testeaza daca numele companiei poate fi setat cu un string gol: “”. Testul foloseste metodele: *assertFalse(companie.getNume().isEmpty())* si *assertFalse(companie.getNume().isEmpty()).* Testele vor da fail daca numele companiei poate fi setat cu empty string si success daca numele a fost validat corect si respinge valorile empty string(“”).
* *testSetNumeCompanieNenull()* – testeaza daca numele companiei poate fi setat cu valori de gama celor asteptate. Testul foloseste metodele: *assertEquals(nume, companie.getNume())* si *assertEquals(nume, companie.getNume())*, unde myId este un string dat de client. Prin folosirea acestui String “nume”, in locul metodei clasice(a scris in test direct valoarea la care ne asteptam) are rolul de a evita eventualele erori de scriere in moementul scrierii testului.

Testul *TestListaAeronave*:

Rol: testarea listei de aeronave aflata in clasa Companie(testarea numarului de elemente din lista inainte, dupa adaugarea unei aeronave si dupa stergerea aeronavei).

*TestListaAeronave* contine 2 teste:

* *testAdaugaAeronavaInLista()* testeaza lungimea listei in momentul adaugarii de aeronave. Testul foloseste metoda *assertEquals(lungimeLista, companie.getNumarAeronave2())*. Metoda getNumarAeronave2() aflata in clasa Companie, returneaza un int: numarul de elemente ale listei(lista este statica).
* *testStergeAeronavaDinLista()* testeaza lungimea listei dupa ce au fost sterge aeronave din aceasta. La fel ca si *testAdaugaAeronaveInLista()*, testul foloseste metoda *assertEquals(lungimeLista, companie.getNumarAeronave2())*. Testul va “trece” daca lungimea listei este cea la care ne asteptam si va “pica”, in sens contrar.

Testul *TestHashMapComenzi*:

Rol: testarea tabelei de dispersie pentru comanzile din clasa Companie(testarea dimensiunii hash-ului, testarea operatiilor de stergere si de adaugare in hashmap).

*TestHashMapComenzi* contine 4 teste:

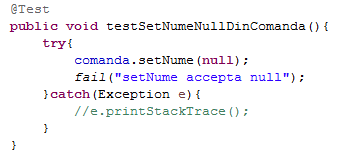
* *testHashMapComenziGoala()* care testeaza daca tabela de dispersie nu tine nicio comanda. S-a folosit metoda *assertTrue(map.isEmpty())* care se termina cu success cand tabela este goala si cu fail cand aceasta contine comenzi.
* *testAdaugaComandaInHashMap()* – testeaza daca, la adaugarea unei comenzi in tabela, dimensiunea acesteia creste. Testul foloseste metoda *assertEquals(dimensiune, map.size()).* Testul trece daca dimensiune hash-ului creste in sensul in care ne asteptam si da fail daca dimensiunea este gresita.
* *testStergeComandaDinHashMap()* – testeaza daca, la stergerea comenzilor din tabela, dimensiunea acesteia scade. Metoda folosita este, si in acest caz, *assertEquals(dimensiune, map.size()).*
* *testDimensiunHashMap()* – testeaza dimensiunea tabelei de dispersie prin metoda *assertEquals(dimensiune, map.size()).*

Testul *TestNumeClient*

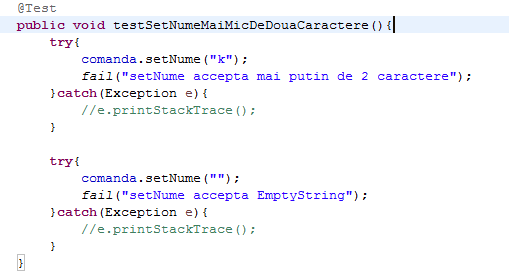
Rol: testarea numelui clientului in vederea stabilirii corectitudinii validarii atributului nume

*TestNumeClient* are 3 teste:

* *testSetNumeDinComanda()* – testeaza daca numele numele este setat corect. Este utilizata metoda: *assertEquals(numeClient, comanda.getNume())*
* *testSetNumeNullDinComanda()* – testeaza daca numele poate fi setat cu valoarea null. Metoda utilizata este: *fail()* si testul trece pe atributul nume avem validari(!nume.isEmpty()), deci testul nu trece de prima instructiune si nu ajunge la *fail():*



* *testSetNumeMaiMicDeDouaCaractere()* – testeaza daca numele unui client este mai mic de 2 caractere(acest caz include si setarea numelui cu empty string). Se foloseste metoda *fail()* la care testul nu ajuge pentru ca numele nu poate avea un singur caracter(vezi setNume() din clasa Comanda).



Test *TestValoriVarstaDinComanda*

Rol: testarea valorilor pe care le poate lua varsta unui client

*TestValoriVarstaDinComanda* are e teste:

* *setVarstaValoriNormale()* – testeaza valorile normale ale varstei(mai mare ca 0 si mai mica decat 91 de ani), Este folosita metoda *assertEquals(varsta, comanda.getVarsta())*.
* *setVarstaEgalZero()* – verifica daca varsta poate fi egala cu 0. Se foloseste metoda fail().
* *setVarstaMaiMicaDecatZero() –* verfica daca varsta poate lua valori mai mici decat 0. Metoda utilizata: fail()
* *setVarstaMaiMareCa90() –* verifica daca varsta ia valori mai mari decat 90 de ani folosind metoda fail().

Test *TestOrasPlecare*

Rol: testeaza daca orasul de plecare a fost completat sau nu

*TestOrasPlecare* foloseste 3 teste:

* *setOrasPlecareNull()* – verifica daca atributul orasPlecare poate lua valoarea null. Testul foloseste metoda *assertNotNull(comanda.getOrasPlecare())* care trece atunci cand orasul de plecare nu este null si da fail cand acesta fost setat pe null.
* *setOrasPlecareEmptyString()* – testeaza daca atributul orasPlecare poate fi setat cu “”. Se utilizeaza metoda *assertFalse(comanda.getOrasPlecare().isEmpty())*. Testul da fail daca orasul poate avea valoarea “”.
* *setOrasPlecareCorect()* – testeaza daca numele orasului este setat corect. Se foloseste metoda *assertEquals(orasPlecare, comanda.getOrasPlecare())* care se termina cu succes daca numele este corect.

Test *TestOrasDestinatie*

Rol: testare setarii orasului destinatie cu valori aberante

*TestOrasDestinatie* are 3 teste:

* *setOrasDestinatieNull()* – verifica daca atributul orasDestinatie poate lua valoarea null. Testul foloseste metoda *assertNotNull(comanda.getOrasDestinatie())* care trece atunci cand orasul destinatie nu este null si da fail cand acesta fost setat pe null.
* *setOrasDestinatieEmptyString()* – testeaza daca atributul orasDestinatie poate fi setat cu “”. Se utilizeaza metoda *assertFalse(comanda.getOrasDestinatie().isEmpty()).* Testul da fail daca orasul poate avea valoarea “”.
* *setOrasDestinatieCorect()* – testeaza daca numele orasului este setat corect. Se foloseste metoda *assertEquals(orasDestinatie, comanda.getOrasDestinatie())* care se termina cu succes daca numele este corect.

Test *TestZiPlecare*

Rol: testeaza daca ziPlecare este completata

TestZiPlecare foloseste 3 metode:

* *setZiPlecareNull()* – verifica daca atributul ziPlecare poate lua valoarea null. Testul foloseste metoda *assertNotNull(comanda.getZiPlecare())* care trece atunci cand ziua plecarii nu este null si da fail cand aceasta fost setata pe null.
* *setZiPlecareEmptyString()* – testeaza daca atributul ziPlecare poate fi setat cu “”. Se utilizeaza metoda *assertFalse(comanda.getZiPlecare().isEmpty()).* Testul da fail daca ziua poate avea valoarea “”.
* *setZiPlecareCorect()* – testeaza daca ziua plecarii este setata corect. Se foloseste metoda *assertEquals(orasDestinatie, comanda.getZiPlecare())* care se termina cu succes daca ziua este corecta.

Test *TestOraPlecare*

Rol: testeaza daca ora de plecare ia valori corecte(intre 0 si 24)

TestOraPlecare foloseste 3 teste:

* setOraPlecareMaiMicaDecatZero() – testeaza daca ora de plecare are valori negative. Se foloseste metoda fail()
* setOraPlecareMaiMareDecat24() – testeaza daca ora de plecare este mai mare decat 24. Metoda folosita este fail()
* setOraPlecareCorecta() – testeaza daca ora de plecare este corecta. Se foloseste metoda assertEquals(ora, comanda.getOraPlecare()). Testul trece daca ora este cea “asteptata” si da fail daca setarea orei nu se realizeaza in mod corect.

Test *TestTipBilet*

Rol: testeaza daca tipul de bilet poate fi seta pe null sau alte valori decat cele ale enumeratiei TipBilet(BEBELUS, COPIL, ADULT).

TestTipBilet are 2 teste:

* setTipBiletNull() – testeaza daca tipul de bilet poate fi null. Se foloseste metoda assertNotNull(comanda.getTipBilet()).
* setTipBiletCorect – foloseste metoda assertEquals(tipBilet, comanda.getTipBilet()).

Test *TestClasa*

Rol: testeaza daca clasa la care se rezerva bilet ia valorile “economy” sau “business”.

*TipClasa* are 2 teste:

* *setClasaCorect()* – testeaza daca clasa este setata pe “economy” sau “business”. Foloseste metoda *assertEquals(clasa, comanda.getClasa()).* Testul “trece” daca atributul clasa a fost setat corect si “pica”, altfel.
* *setClasaGresita()* – testeaza clasa poate lua alte valori decat cele mai sus mentionate. Foloseste metoda *fail().*

Pe langa metodele specificate, au mai fost implementate si metode clasice Junit 4: *setUpBeforeClass(), tearDownAfterClass(), setUp(), tearDown().*

Test *TestCaseFisierText*

Rol:testeaza corectitudinea preluarii unor atribute dintr-un fisier text si construirea de obiecte de tip Comanda(prin intermediul ComandaBuilder) cu ajutorul acestora.

*TestCaseFisierText* contine o singura metoda de test, testFisierText(), in care se verifica daca atributele comenzilor create sunt identice/egale cu datele preluate din fisier.

Exemplu de test pentru prima comanda:

*assertEquals*("nume: ", *lista*.get(0).getNume(), "ana");

*assertEquals*("varsta: ", *lista*.get(0).getVarsta(),22);

*assertEquals*("nationalitate: ", *lista*.get(0).getNationalitate(),"roman");

Test *TestObiecteMock*

Rol: testeaza functionalitatea unor metode externe clasei PretBilet.

*TestObiecteMock* utilizeaza framework-ul EasyMock, folosit sa imite interfete folosite apoi in Unit Testing(aici, am folosit interfata ModificaPret). Foloseste Java Reflection pentru a crea obiecte care imita interfete. Obiectele mock reprezinta niste proxy-uri ale unor implementari existente.

Prin obiectele mock se testeaza functionalitatea unei clase in izolare, din interiorul unor teste Junit. Ele joca rolul unor obiecte sau servicii reale.

Metodele folosite: testFaScumpire() si testFaIeftinire() care au rolul de a creste sau scadea valoarea preturilor biletelor.

1. **Definirea si descrierea Test Suite-ului**

Test Suite-ul, cunoscut si sub denumirea de *validation suite*, contine o colectie de test case-uri folosite pentru a testa o serie de metode din cadrul unei program. Are rolul de a verifica si demonstra anumite comportamente ale acelui program.

Test Suite-ul contine toate Test Case-urile mai sus descrise:

1. *TestIdAeronava*
2. *TestNumeCompanie*
3. *TestListaAeronave*
4. *TestHashMapComenzi*
5. *TestNumeClient*
6. *TestValoriVarstaDinComanda*
7. *TestOrasPlecare*
8. *TestOrasDestinatie*
9. *TestZiPlecare*
10. *TestOraPlecare*
11. *TestTipBilet*
12. *TestClasa*
13. *TestCaseFisierText*
14. *TestObiecteMock*
15. **Descrierea sumara a functiilor aplicatiei cu referire la pattern-uri si la metodele testate**

O companie aeriana gestioneaza avioane de doua tipuri: avioane pentru transportul pasagerilor si avioane de tip cargo, pentru marfuri.

La bordul avioanelor de pasageri se afla mai multi insotitori de zbor care au grija sa verifice calatorii la urmare si coborare.

O comanda trebuie sa contina informatii detaliate despre client si preferintele acestuia legate de facilitatile oferite de companie(nume, varsta, nationalitate, orasPlecare, ziPlecare, oraPlecare, orasDestinatie, ziSosire, oraSosire, bagajCala, clasa la care rezerva biletul). De asemenea, calatorul poate alege ce tip de bilet comanda(bebelus, copil, adult) si modalitatea de plata(card, transfer bancar, cash).

Pretul biletelor variaza in functie de clasa la care se calatoreste.

1. **Bibliografie**
2. acs.ase.ro
3. online.ase.ro
4. <http://www.tutorialspoint.com>
5. [www.youtube.com](http://www.youtube.com)