**PROIECT CALITATE SI TESTARE SOFTWARE**

Student: Ciucă Diana-Maria Profesor Îndrumător: Alin Zamfiroiu

Grupa: 1064

Seria: A

Facultate: Cibernetică, Statistică și Informatică Economică

Specializare: Informatică Economică

**Cuprins**

**Capitolul I**

**Introducere:**

**1.1 Descriere generala proiect**

**Capitolul II**

**Design Patterns:**

**2.1 Definire si justificare alegere patternuri pentru implementare**

**2.2 Prezentarea succinta a functiilor din cadrul patternurilor**

**Capitolul III**

**Unit Testing:**

**3.1 Definire metode supuse testarii in Unit Testing**

**3.2 Prezentare detaliata a metodelor supuse testarii**

**3.3 Definire si descriere Test Case-uri**

**3.4 Definire si descriere Test Suite**

**3.4 Prezentarea succinta a functiilor implementate pe baza functiilor testate**

**Capitolul IV**

**4.1 Bibliografie**

**Capitolul I**

**1.1 Descriere generala proiect**

Proiectul realizat utilizând tehnologia Java a avut drept temă : „Baze de date și fire de execuție în Java”.

În lucrul cu thread-uri, m-am gândit la ghișeele unei bănci și la ideea de a sta la coadă ca și client pentru a efectua o plată.

Astfel că am simulat funcționarea acestor ghișee și a modului în care clienții se așează la coadă, repectiv eliberează un ghișeu.

Am folosit următoarele clase: Client, Coadă, Ghișeu\_bancă și o clasă principală în care am testat funcționalitatea claselor.

În clasa Client, am folosit drept atribute private id-ul clientului, data sosirii la bancă și data plecării.

În vederea creării instanțelor de tip Client, am declarat un constructor cu parametri. De asemenea, atributele private au fost prevăzute cu metode de acces(getteri și setteri) și am afișat aceste atribute cu ajutorul metodei toString().

Clasa Ghișeu\_bancă este un fir de execuție care extinde clasa Thread și are drept atribute un vector de clienți.

Constructorul clasei apelează constructorul superclasei.Codul firului de executie va fi executat în metoda run().

Operația executată în metoda run() este de ștergere a unui client.

Se regăsesc și metode ale clasei Thread: sleep(), wait(), notifyAll() și mecanisme try-catch de tratare a excepțiilor,.

De asemenea, am utilizat metoda synchronized pentru sincronizarea firelor de execuție.

Un alt fir de execuție este clasa Coadă care se creează tot prin extinderea clasei Thread și supradefinirea metodei run().

De asemenea, constructorul acestei clase apelează constructorul superclasei.

De această dată, operația executată în metoda run() este de adăugare a unui client.

Ca și în cazul celuilalt fir de execuție, se regăsesc mecanisme try-catch și metodele sleep(), wait(), notifyAll().

Clasa principală conține metoda main în care se testează funcționalitatea claselor.

În main am instanțiat un obiect de tip Ghișeu\_bancă și unul de tip Coadă.

Pentru lansarea în execuție a firelor de execuție am folosit metoda start().

În ceea ce privește lucrul cu baze de date, mi-am propus crearea unei baze de date ce cuprinde informații despre clienții unei bănci și despre plățile efectuate de către aceștia.

M-am conectat pentru început la o bază de date derby.

În continuare am creat o tabelă Clienți ce avea drept câmpuri: id client, nume, prenume, vârstă, adresă și o tabelă Plată ce avea drept câmpuri: id client, suma plată, dată plată.

Apoi am realizat operații de adăugare, modificare, ștergere a unui client în baza de date, iar interfața grafică utilizată a fost SWING.

Pe langa acestea, proiectul a fost imbunatatit prin implementare de diferite design patterns, dar si prin adaugare de testare de metode prin Unit Testing.

Acestea vor fi prezentate in cele ce urmeaza.

**Capitolul II**

**Design Patterns:**

**2.1 Definire si justificare alegere patternuri pentru implementare**

**Singleton**

Un prim design pattern care ar putea fi implementat este design pattern-ul creational Singleton, dat fiind faptul ca in contextul proiectului este vorba de existenta unei singure banci.

**Factory**

Deoarece proiectul vizeaza procesul de plata la banca de catre clienti, ne putem gandi la categoriile de clienti pe care le dorim. Astfel ca prin clienti ne putem referi fie la clienti persoane fizice, fie la comercianti persoane fizice, fie la persoane juridice.

Astfel ca generarea mai multor elemente din aceeasi familie(a clientilor) ne determina sa alegem pentru implementare design pattern-ul creational **Factory,** deoarce ne propunem sa cream mai multe tipuri de clienti.

**Builder**

Am utilizat design pattern-ul creational Builder in proiect pentru construirea unui obiect complex si anume un obiect de tip Client.

Clientul la care facem referire in proiect are urmatoarele attribute: id, plecare, sosire, modalitate\_plata, nume.

Insa sa presupunem ca pentru angajatii de la banca sunt relevante doar cateva dintre aceste atribute(de exemplu doar id-ul si numele clientului).

De aceea, folosim design patternul Builder, pentru a exista sansa de a alege ce parti din obiect sa fie create.

**Observer**

Observer este un design pattern comportamental, care se aplica atunci cand se doreste ca anumite componente sa fie notificate atunci cand un eveniment are loc.

Am ales sa implementez Observer intrucat firele de executie se anunta unele pe celelalte atunci cand anumite evenimente se produc(a fost adaugat un nou client la coada, a disparut un client din coada, respectiv se trimite o notificare cu lungimea curenta a cozii).

In lucrul cu fire de executie, se poate constata existenta implementarii design pattern-ului Observer, deoarece firele de executie comunica intre ele si se anunta unele pe celelalte despre monitorul fiecarui obiect, in vederea obtinerii „lacatului” acestuia(mai exact un fir de executie anunta celelalte fire de executie de faptul ca un eveniment anume s-a produs si astfel, acestea isi pot continua activitatea).

Prin monitor se intelege un concept de sincronizare, care permite thread-urilor sa se excluda reciproc si sa aiba capacitatea de a astepta ca o anumita conditie sa devina adevarata.

Un monitor contine doua aspecte: un obiect „blocat” si variabile conditionale. O variabila conditionala este un fel de container de thread-uri, care asteapta indeplinirea unei anumite conditii.

Monitoarele obliga firele de executie sa renunte la accesul total pentru a astepta producerea unei anumite conditii, inainte de recastigarea accesului complet si a posibilitatii de a-si relua activitatile.

**Strategy**

Deoarece proiectul face referire la ideea de efectuare a unor plati(platire de credite), automat ne ducem cu gandul ca efectuarea platilor se poate face in mai multe moduri si anume, de exemplu: numerar, card bancar, internet banking. Avantajul este faptul ca fiecare client al bancii poate hotari prin ce modalitate doreste sa realieze plata.

Oricare ar fi calea aleasa de acesta, aceasta va fi aleasa in momentul in care va ajunge la ghiseu, adica la run-time.

Astfel ca, avand contextul de mai sus, putem implementa design patternul comportamental Strategy. Acest design pattern se refera exact la posibilitatea alegerii la run-time de a urma o anumita solutie(algoritm) si nu alta.

**Facade**

Data fiind tema proiectului, aceea de efectuare de plati la o banca, enuntul problemei s-ar putea reformula in felul urmator:

La o banca, in vederea efectuarii platii lunare de catre clientii acesteia, se parcurg mai multi pasi si anume:

1. Mai intai are loc deschiderea bancii aferente

2. Ghiseele sunt deschise pentru a primi clienti sa efectueze platile bancare.

3. Incep sa apara clienti.

4. Clientul se aseaza la ghiseul cu coada cea mai scurta.

5. Clientul ajunge la ghiseu.

6. Clientul alege modalitatea de plata si efectueaza plata.

7. Clientul elibereaza coada si paraseste banca.

Aceste etape de mai sus nu fac altceva decat sa prezinte o succesiune logica de pasi, care reprezinta un scenariu simplificat si ele se vor transforma in niste functii care vor fi apelate si utilizate.

In cazul de fata scenariul prezentat este cel al activitatii cotidiene al unei banci, la care se efectueaza diverse plati.

Avand in vedere succesiunea de pasi existenta mai sus, se poate aplica design pattern-ul Facade, deoarece scopul acestui pattern este acela de a simplifica un anumit context sau scenariu. In cazul nostru, este prezentata o succesiune complexa de pasi, care poate fi rezolvata cu ajutorul design pattern-ului structural Facade.

**Decorator**

Un alt desig pattern implementat in proiect este design pattern-ul structural Decorator.Prin acest design pattern obiectul curent este extins prin folosirea mai multor decoratori.

In cazul de fata, o instanta de tip Banca este extinsa. In clasa Banca, pe langa metodele initiale deschide\_banca(), respectiv inchide\_banca(), dorim sa mai adaugam noi metode si anume refacere\_sistem\_incalzire() si zugravire\_incaperi().

Astfel ca, se poate spune ca „decoram” clasa Banca prin intermediul acestor metode,adaugand de fapt noi functionalitati in aceasta clasa.

**Composite**

Deoarece in contextul de fata se prezinta problema simularii efectuarii platii de catre niste clienti la o banca, de aici putem trage concluzia ca pentru a realiza plata clientii stau la o coada.

Coada este desigur alcatuita din unul sau mai multi clienti.

Pe baza acestei idei, consider ca design pattern-ul Composite poate fi aplicat cu succes, deoarece este vorba despre o componenta compusa(coada), care are noduri fii si anume clientii care stau la coada.

De asemenea, design pattern-ul Composite este cumva implementat si prin intermediul thread-urilor: Coada si Ghiseu\_Banca, prin prisma adaugarilor si stergerilor de clienti de la coada.

**Chain of Responsability**

In momentul in care banca se deschide incep sa apara si clienti la ghisee. Atunci cand apare primul client, acesta are sansa de a alege la ce ghiseu doreste sa realizeze plata, deoarece este singurul client si nu exista nicio coada la vreun ghiseu.

Dar, dat fiind faptul ca nu este foarte aglomerat dimineata, ghiseele nu vor deschide simultan.De aceea, clientul se poate indrepta catre un ghiseu care nu este disponibil deocamdata. Daca primul ghiseu ales nu este disponibil, clientul isi va indrepta atentia catre un al doilea ghiseu apropiat. Daca nici acesta din urma nu este deschis, clientul va merge la urmatorul ghiseu disponibil si tot asa.

In momentul in care clientul nu gaseste niciun ghiseu disponibil, din cauza unor probleme aparute in situatii exceptionale si este nevoit sa astepte o perioada pentru disponibilitatea ghiseelor, acesta va parasi banca, urmand sa revina la o ora mai tarzie, in speranta ca problemele aparute vor fi solutionate.

Luand in considerare scenariul prezentat mai sus, am ales sa implemntez pentru aceasta situatie design pattern-ul comportamental Chain of Responsabiliy. Acest design pattern gestioneaza o insiruire de actiuni ce se vor produce in cadrul unui anumit eveniment.

Actiunile sunt dependente unele de celelalte si efectuarea uneia poate conduce fie la efectuarea urmatoarei actiuni, fie la eliminarea celorlalte actiuni.

**2.2 Prezentarea succinta a functiilor din cadrul patternurilor**

**Singleton**

Pentru implementarea design-pattern-ului Singleton am realizat clasa „Banca”, in care am definit un atribut privat static pe care il vom returna de fiecare data.Exact acesta este scopul implementarii Singleton si anume: returnarea aceleiasi instante de fiecare data.

Tot caracteristice Singleton-ului sunt si constructorul privat, cat si functia de returnare a instantei private statice definite mai sus.Metoda de returnare a instantei private statice este metoda publica statica „getInstance”: in cadrul acestei functii, se verifica daca exista instanta si daca nu exista este creata si ulterior returnata.

In clasa „MainDiana”, in metoda main se creaza o singura instanta a clasei „Banca”, apelandu-se metoda „getInstance”, prin care se va returna de fiecare data aceasta instanta.

Astfel, am folosit Singleton pentru a demonstra faptul ca o clasa poate fi instantiata o singura data si deci instanta clasei este unica.(banca este unica)

**Factory**

Acest tip de design pattern a fost implementat in felul urmator:

S-a creat interfata IClient, care contine metoda stabilire\_tip().

Aceasta metoda este implementata de clasele urmatoare: Client\_persoana\_fizica, Comerciant\_persoana\_fizica si Persoana\_Juridica, prin afisarea unor mesaje corespunzatoare in functie de categoria clientului.

Clasa de creare efectiva a unui anume tip de client este clasa ClientFactory, care contine metoda createObiect(), ce primeste ca parametru un anumit tip de client declarat in enum-ul Clienti. Enum-ul clienti contine urmatoarele categorii de clienti: Client\_persoana\_fizica, Comerciant\_persoana\_fizica, Persoana\_Juridica.

In metoda creareObiect() a clasei ClientFactory(), se creeaza pe rand obiecte de tip Client\_ persoana\_fizica, Comerciant\_persoana\_fizica si Persoana\_juridica.

In metoda Main() se creeaza un obiect de tip ClientFactory.

Prin intermediul obiectului de tip ClientFactory() se apeleza metoda creareObiect() din clasa ClientFactory si astfel se creeaza un anumit tip de client. Pentru a vizualiza care este tipul clientului creat se apeleaza metoda stabilire\_tip().

**Builder**

Clasele utilizate in implementarea acestui design pattern sunt:

* Clasa Client, cu atributele specifice unui client, getter si setteri pentru atribute.
* Interfata IClientBuilder care contine metoda build() de constuire a obiectului.
* Clasa ClientBuilder, care implementeaza interfata IClientBuilder.

In constructorul clasei ClientBuilder, se creeaza o instanta de tip Client.

Aceasta clasa contine de asemenea setteri pentru campurile clasei Client, prin care se returneaza aceasta instanta, dupa ce ii sunt setate pe rand anumite valori ale atributelor.

Obiectul complex construit este returnat in metoda build().

Testarea se face in metoda MainDiana. Se creaza obiecte de tip Client, pentru care se apeleaza setteri pe care ii dorim din clasa ClientBuilder,impreuna cu metoda build(), pentru constuirea obiectului cu atributele care ne intereseaza.

Obiectele sunt apoi afisate.

**Observer**

In cazul de fata, avem doua fire de executie si anume: clasa Coada si clasa Ghiseu\_banca, care comunica intre ele.

Clasa care este pe post de observator si care anunta celalalt thread(subiectul, adica clasa Coada) de producerea unui eveniment este clasa Ghiseu\_Banca.

Clasa Ghiseu\_Banca contine un vector care retine toti clientii care sosesc la banca, cat si o metoda adauga\_client, prin care are loc adaugarea clientului nou in vector. In momentul in care un client soseste la banca, el doreste sa se aseze la coada cea mai scurta a unui ghiseu. De aceea, se va trimite o notificare catre clasa Coada, cu scopul asigurarii ca acest acest client va fi asezat la coada cea mai scurta.

Astfel ca, metoda adauga\_client din clasa Ghiseu\_Banca contine metoda notifyAll(), prin care anunta thread-ul Coada, ca isi poate relua activitatea si anume, aceea de adaugare in coada cea mai scurta, pentru ca a aparut un nou client ce doreste sa realizeze o plata anume.

Constructorul clasei Coada contine ca parametru un vector de ghisee, numele cozii,cat si numarul de clienti(clienti).

In clasa Coada se executa metoda run(), in care se concentreaza intreaga activitate a thread-ului respectiv.Activitatea acestei clase consta in adaugarea nou venitului la coada cea mai scurta, ca urmare a primirii notificarii.In aceasta clasa, se calculeaza care ghiseu are coada cea mai mica, prin metoda „coada\_min” si in functie de stabilirea acelui ghiseu la care coada are lungime minima, se adauga la ghiseul respectiv ultimul client sosit. Adaugarea la coada se face prin apelarea metodei „adauga\_client” a clasei Ghiseu\_Banca.

Daca activitatea thread-ului Coada consta in adaugarea unui client nou la coada cea mai scurta, rolul thread-ului Ghiseu\_Banca acela de stergere a unui client din coada, dupa ce acesta a terminat de efectuat plata.

Astfel ca, in clasa Ghiseu\_Banca exista metoda sterge\_client(), prin care este inlaturat din lista un client care a fost deja servit la ghiseu.De asemenea, este apelata metoda „notifyAll”, prin care firul de executie anunta celalalt fir de executie ca isi realizeaza activitatea. Dupa aceasta, prin functia „nr\_clienti”, se calculeaza noua lungime a cozii, dupa ce un client a disparut din coada.

In clasa principala „MainDiana”, am creat cele 2 thread-uri Ghiseu\_Banca si Coada si le-am lansat in executie prin metoda start(). Astfel, cele doua thread-uri isi pot realiza activitatile, oferindu-si acces reciproc una celeilalte, prin transmitere de notificari.

**Strategy**

Pentru a implementa design pattern-ul comportamental Stratey am definit o interfata Modalitate\_Plata, care contine semnatura metodei efectueaza().

Interfata va fi implementata de cele 3 clase concrete si anume: Numerar, Card\_Bancar si Internet\_Banking, care reprezinta cele 3 strategii ce pot fi executate de catre client.Fiecare din cele 3 clase are implementata metoda efectueaza(), cu mesajul aferent fiecarei strategii in parte.

Cel care beneficiaza insa de aceste 3 strategii este clientul, care va decide la run-time, adica in momentul in care este la ghiseu ce strategie va alege.

Clasa Client contine o referinta catre interfata Modalitate\_Plata: modalitatePlata, pentru aceasta referinta regasidu-se desigur un constructor cu parametri, getteri si setteri.

In plus, in clasa Client se gaseste metoda plateste(). Prin aceasta metoda se verifica daca a fost aleasa o cale de efectuare a platii de catre client si in caz afirmativ, se va apela metoda efectueaza() din interfata Modalitate\_Plata, pentru a se afisa mesajul corespunzator strategiei implementate.

In cazul in care clientul nu a selectat niciun mijloc de realizare a platii se va afisa un mesaj in acest sens.

Testarea functionalitatii acestui design pattern are loc in clasa “BancaFacade”, unde am creat o referinta de tip Modalitate\_Plata catre obiectul sau strategia care va fi aleasa. De asemenea, am instantiat un obiect de tip Client, caruia i-am setat ca parametru strategia pe care si-a ales-o. S-a ales strategia „Numerar”.

Pentru aceasta strategie, se apeleaza metoda plateste() a clasei Client si astfel va aparea un mesaj cu modalitatea de plata aleasa de catre client.

**Facade**

Pentru implementarea acestui design pattern, s-au creat mai multe metode relativ simple, dar dat fiind faptul ca numarul lor este mai mare, acest aspect le face sa aduca un grad ascuns de complexitate proiectului.

De aceea toate aceste functii simple au fost apelate intr-o singura functie din clasa BancaFacade, care le contine pe toate si anume functia: executa\_activitati(), in vederea simplificarii modelului.

Functiile de complexitate scazuta de care aminteam mai sus sunt:

-deschide\_banca() – prin aceasta functie se afseaza un mesaj cum ca banca a fost deschisa si este prima activitatea care are loc in cadrul activitatii unei banci

-inchide\_banca() - prin aceasta functie se afseaza un mesaj cum ca banca a fost inchisa si este ultima activitatea care are loc in cadrul activitatii unei banci

-sosire() - aceasta functie se afla in clasa Client si contine un mesaj general in care se anunta sosirea clientului la banca la o anumita ora.

-plecare() - aceasta functie se afla in clasa Client si contine un mesaj general in care se anunta plecarea clientului de la banca la o anumita ora.

Aceste functii sunt cuprinse si apelate in alte doua functii si anume:

-sosire\_clienti() –in aceasta functie se parcurge o lista de clienti si pentru fiecare client se afiseaza mesajul ca acesta a sosit la banca, prin apelarea metodei de mai sus: sosire().

-plecare\_clienti()-in aceasta functie se parcurge o lista de clienti si pentru fiecare client se afiseaza mesajul ca acesta a plecat de la banca, prin apelarea metodei de mai sus: plecare().

Toate functiile prezentate mai sus sunt incapsulate intr-o metoda statica „parinte” numita executa\_activitati(), din clasa BancaFacade.Practic prin aceasta functie, se prezinta functionalitatea patternului Facade.

In aceasta clasa se apeleaza functiile de mai sus, alaturi de alte functii precum: functia adaugaClient(), din clasa Ghiseu\_banca si functia plateste() din clasa Client().

Functia adauga\_client() are rolul de a adauga un client in lista de clienti, proiectul urmarind exact actiunile intreprinse de acestia la banca.

Functia plateste() din clasa Ghiseu\_banca() arata ce modalitate de plata prefera clientul sa aleaga in momentul in care ajunge la ghiseu. De fapt am folosit patternul Strategy in interiorul patternului Facade, pentru a oferi clientului sansa de alegere a mijlocului de plata dorit la run-time. Despre patternul Strategy am vorbit mai detaliat mai sus, in cadrul acestui design pattern.

In metoda principala „MainDiana”, am apelat metoda statica executa\_activitati(), chiar prin numele clasei: „BancaFacde”, deoarce metoda este statica.

Prin apelarea acesteia, se va executa toata secventa logica de pasi privind activitatea dintr-o banca, de care am amintit mai sus.

**Decorator**

Design pattern-ul Decorator a fost utilizat prin intermediul urmatoarelor clase si interfete, prezentate in cele ce urmeaza.

Interfata IBanca contine semanturile metodelor initiale: deschide\_banca() si inchide\_banca().

Metodele sunt implementate in clasa Banca, prin afisarea unui mesaj corespunzator.

In continuare, avem clasa abstracta Abanca, in care definim mai intai o instanta de tip IFilm, pe care o introducem ca parametru in constructorul clasei ABanca.

Aceasta instanta este folosita pentru a apela metodele deschide\_banca() si inchide\_banca() din clasa Banca, deoarece dorim sa pastram si functionalitatile initiale.

De aceea, se creeaza alte doua metode deschidere\_banca() si inchidere\_banca(), unde vor fi apelate metodele initiale.

Apoi, scriem antetele celor doua functii pe care dorim sa le folosim pentru „decorare” si anume: refacere\_sistem\_incalzire() si zugravire\_incaperi().

In clasa MainDiana am creat o instanta a noii clase(BancaRenovata), care se construieste pe baza vechii instante de tip Banca.

In cele din urma, prin aceasta instanta se apeleaza metodele clasei BancaRenovata si se afiseaza noile functionalitati.

**Composite**

Design pattern-ul Composite a fost implementat in felul urmator:

* Am folosit o interfata Persoana care contine metodele: add(), remove(), getCopil(), si detalii.Interfata este implementata de clasele Coada si Client.
* Clasa Client contine informatii despre clientul care soseste la banca sa efectueze o plata.In metoda detalii din aceasta clasa sunt afisate informatiile despre client.Aceasta clasa este un nod frunza, de aceea pentru metodele add() si remove() nu exista implementare.
* Clasa Coada – implementeaza toate cele trei metode: add(), remove() si detalii(). Prin metoda add() este adaugata o noua persoanain lista de personae, iar prin metoda remove() este stearsa o persoana din lista. Metoda detalii() parcurge lista de persoane si pentru fiecare afiseaza informatiile aferente.

Functionalitatea acetsui design pattern este testata in clasa MainDiana: ne-am creat doua instante ale clasei Client, ce urmeaza a fi adaugate intr-o coada de clienti. Apoi ne-am creat o a doua coada de client in care am introdus un singur client(nod frunza).

Cea de-a doua coada este adaugatain component primei cozi, pentru a se intari idea de compunere, pe care se bazeaza acest design pattern.

**Chain of Responsability**

Acest design pattern a fost implementat, utilizand urmatoarele clase:

* Clasa „Ghiseu” – contine atributul de tip String „alege\_Ghiseu”, care semnifica cele 3 variante de ghisee la care clientul poate fi primit. Atributul are constructor cu parametri, getteri si setteri.
* Clasa abstracta „Handler” - contine semnatura metodei servire\_ghiseu(Ghiseu ghiseu), cu un parametru de tip ”Ghiseu”, urmand sa fie implementata de clasele care extind aceasta clasa. Aceasta clasa contine si o metoda setter aferenta atributului clasei „succesor”, pentru a putea realiza acest proces de „chain”, adica de inlantuire a operattilor, in cazul in care primul criteriu nu este indeplinit.
* Clasele „Ghiseu 1”, „Ghiseu 2”, „Ghiseu 3” – sunt clasele care extinde clasa abstracta „Handler” si care implementeaza metoda „servire\_ghiseu()”.

Se verifica daca ghiseul ales de catre client este disponibil. Daca da, se afiseaza un mesaj aferent, daca nu, se apeleaza metoda „sevire\_ghiseu(ghiseu)” pentru al doilea ghiseu din sir si se verifica de asemenea daca acesta este disponibil. Daca este disponibl, se afiseaza un mesaj aferent, daca nu, se trece la al treilea ghiseu. Se repeta pasii de mai sus, cu exceptia ca daca nici acest ghiseu nu este disponibil, se afiseaza un mesaj care anunta clientul sa revina la o alta ora.

In metoda „MainDiana” este testata functionalitatea design-pattern-ului astfel: se creaza cele 3 ghisee, se stabileste inlantuirea dintre ele prin apelarea succesiva a metodei „setActiuneUrmatoare(ghiseu x)”, se creaza un obiect de tip Ghiseu si se instantiaza, dupa care se apeleaza metoda „servire\_ghiseu(ghiseu)”, care primeste ca parametru ghiseul instantiat adineauri.

Se afiseaza mesaje aferente ghiseului ales sau se informeaza clientul sa revina mai tarziu daca nu se alege niciunul din ghiseele existente.

**Capitolul III**

**Unit Testing:**

**3.1 Definire metode supuse testarii in Unit Testing**

Pentru testarea in UnitTesting s-au folosit urmatoarele metode:

Pentru a testa metodele din clasa Client s-a creat TestCase-ul TestClient.

Metodele testate au fost urmatoarele:

-getAdresa()

-verifica\_An()

- verifica\_Luna()

- verifica\_Zi()

- verifica\_Ora()

-setAdresa()

- verificaPret

-calculeazaPlataTotala()

-estePersoanaJuridica()

-varstaClient()

Pentru a testa metodele din clasa Banca s-a creat TestCase-ul TestBanca.

Metodele testate au fost urmatoarele:

-getInstance()

-stare\_banca()

Pentru a testa metodele din clasa Ghiseu s-a creat TestCase-ul TestGhiseu.

Metodele testate au fost urmatoarele:

-getAlege\_Ghiseu()

Pentru a testa metodele din clasa Ghiseu\_banca s-a creat TestCase-ul TestGhiseu\_banca.

Metodele testate au fost urmatoarele:

-adauga\_client()

-nr\_clienti()

-add()

-remove()

-getCopil()

Pentru a testa metodele din clasa Handler s-a creat TestCase-ul TestHandler.

Metodele testate au fost urmatoarele:

-setActiuneUrmatoare()

**3.2 Prezentare detaliata a metodelor supuse testarii**

-getAdresa() – getter al campului privat adresa

-verifica\_An() – se verifica daca anul de sosire la banca este acelasi cu anul de plecare de la banca.

- verifica\_Luna() – se verifica daca luna de sosire la banca este aceeasi cu luna de plecare de la banca.

- verifica\_Zi()-se verifica daca ziua de sosire la banca este aceeasi cu ziua de plecare de la banca.

- verifica\_Ora()-se verifica daca ora de sosire la banca este aceeasi cu ora de plecare de la banca.

-setAdresa()- setter pentru campul privat adresa

-verificaPret()-se verifica daca pretul platii unei facturi de exemplu este negativ sau pozitiv

-calculeazaPlataTotala()-calculeaza suma datorata de catre client, la care se aduga si un eventual comision

-estePersoanaJuridica()- se verifica tipul unui anumit client(tipul fiind generat anterior prin design pattern-ul Factory)

-varstaClient() – se verifica daca un anumit client are dreptul de a efectua o plata la banca(daca este minor, nu are dreptul)

-getAlege\_Ghiseu() – getter pentru campul privat alege\_Ghiseu

-getInstance()-aceasta metoda este caracteristica design pattern-ului Singleton si returneaza o singura instanta(instanta este unica de fiecare data)

-stare\_Banca() – aceasta metoda verifica daca banca este inchisa sau deschisa la un moment dat

-adauga\_client()-se adauga un nou client in lisat de persoane aflate la o coada

-nr\_clienti() – se calculeaza numarul de clienti din coada

- add()- se adauga de asemenea in lista

-remove()-se sterge din lista un client

-getCopil() – se returneaza un client din lista de la o anumita pozitie

-setActiuneUrmatoare() – folosit in design pattern-ul Strategy

**3.3 Definire si descriere Test Case-uri**

Pentru a testa metodele din clasa Client s-a creat **TestCase-ul TestClient**.

Pentru a testa metodele din clasa Banca s-a creat **TestCase-ul TestBanca.**

Pentru a testa metodele din clasa Ghiseu s-a creat **TestCase-ul TestGhiseu**.

Pentru a testa metodele din clasa Ghiseu\_banca s-a creat **TestCase-ul TestGhiseu\_banca.**

Pentru a testa metodele din clasa Handler s-a creat **TestCase-ul TestHandler.**

**3.4 Definire si descriere Test Suite**

Am creat **TestSuite-ul AllTests,** care cuprinde urmatoarele **TestCase-uri:**

**TestCase-ul TestClient**

**TestCase-ul TestBanca**

**TestCase-ul TestGhiseu**

**TestCase-ul TestGhiseu\_banca**

**TestCase-ul TestHandler**

**3.4 Prezentarea succinta a functiilor implementate pe baza functiilor testate**

-testGetAdresa()- testeaza daca adresa data returnata coincide cu cea introdusa

-testVerifica\_An()- testeaza ca anul sosirii sa fie acelasi cu anul plecarii

- testVerifica\_Luna() - testeaza ca luna sosirii sa fie aceeasi cu luna plecarii

- testVerifica\_Zi()- testeaza ca ziua sosirii sa fie aceeasi cu ziua plecarii

- testVerifica\_Ora() - testeaza ca ora sosirii sa fie mai mica decat ora plecarii

-testSetAdresa() – testeaza ca valoarea adresei date ac parametru sa nu fie nula

- testVerificaPret – testeaza ce intampla daca pretul este pozitiv, respectiv negativ

-testCalculeazaPlataTotala() – se doreste ca valoarea asteptata a platii sa coincida cu valoarea rezultata

-testEstePersoanaJuridica() – se testeaza tipul persoanei, cu assertTrue

-testVarstaClient() – se arunca o exceptie daca varsta clientului nu este mai mare de 18 ani

-testGetInstance() – se arata cu assertSame ca se returneaza acelasi obiect prin Singleton

-testStare\_banca() – cu assertTrue se testeaza starea curenta a bancii

-testGetAlege\_Ghiseu() – cu assertTrue se arata ca se returneaza prin getter valoarea care se trimite

-testAdauga\_client() – se asigura ca nu se tarnsmite null

-testNr\_clienti() – se asigura ca lista contine un numar pozitiv de elemente

-testAdd\_NULL()/testAdd\_NOTNULL – se observa ce se intampla daca se adauga null sau nu se adauga

-testRemove()- se observa ce se intampla daca se vrea sa se sterga un element care nu exista

-testSetCopil() - se observa ce se intampla daca se vrea sa se extraga un element care nu exista

-testSetActiuneUrmatoare() – se asigura ca nu se transmite null

**4.1 Bibliografie**

**http://acs.ase.ro/software-quality-testing**

**http://www.tutorialspoint.com/junit/**