Proiect Calitate si Testare Software

~ Design Patterns ~

~ Unit Testing ~

Conf. univ. dr.Catalin Boja,

Prof. univ. dr. Paul Pocatilu

Student: Zamfir Andrei

Seria B, grupa 1072

Contents

[Capitolul 1 – Descrierea scenariului proiectului 3](#_Toc452209393)

[Capitolul 2 – Design Patterns 4](#_Toc452209394)

[2.1 Singleton 4](#_Toc452209395)

[2.2 Factory 5](#_Toc452209396)

[2.3 Builder 5](#_Toc452209397)

[2.4 Flyweight 6](#_Toc452209398)

[2.5 Facade 6](#_Toc452209399)

[2.6 Observer 7](#_Toc452209400)

[2.7 Proxy 7](#_Toc452209401)

[2.8 Strategy 7](#_Toc452209402)

[2.9 Composite 8](#_Toc452209403)

[Capitolul 3 – Unit Testing 8](#_Toc452209404)

[3.1 Test Case 9](#_Toc452209405)

[3.2 Test Case – clasa ContCredit 9](#_Toc452209406)

[3.2.1 Test initializare 9](#_Toc452209407)

[3.2.2 Test plata 9](#_Toc452209408)

[3.2.3 Test tip cont 9](#_Toc452209409)

[3.3 Test Case – clasa Banca 10](#_Toc452209410)

[3.3.1 Test singleton 10](#_Toc452209411)

[3.3.2 Test primul subordonat 10](#_Toc452209412)

[3.3.3 Test existenta clienti 10](#_Toc452209413)

[3.3.4 Test decizie notificare prin telefon 10](#_Toc452209414)

[3.4 Test Case – clasa Client 10](#_Toc452209415)

[3.4.1 Test sold 10](#_Toc452209416)

[3.4.2 Test adaugare cont 11](#_Toc452209417)

[3.4.3 Test simplificare operatii 11](#_Toc452209418)

[3.5 Test Suite 11](#_Toc452209419)

[3.5.1 Test Suite - Client 11](#_Toc452209420)

[3.5.2 Test Suite – Complet 11](#_Toc452209421)

[Capitolul 4 – Github 11](#_Toc452209422)

[Bibliografie: 12](#_Toc452209423)

# Capitolul 1 – Descrierea scenariului proiectului

Pentru realizarea acestui proiect am ales construirea unei aplicatii care gestioneaza diferitele tipuri de conturi ale clientilor unei banci, de exemplu cont de credit, cont de debit sau alte tipuri din familia conturilor (FACTORY), dar si ierarhia angajatilor bancii. Banca este o resursa unica existenta in sistem si trebuie tratata ca atare. (SINGLETON)

Clientul poate detine un singur cont din fiecare tip, cont ce va fi creat daca nu exista deja, iar in caz contrar, folosit.(FLYWEIGHT) Daca clientul doreste sa faca un imprumut, aceasta optiune va fi posibila la crearea unui nou cont de credit, imprumutul fiind atasat contului intr-o etapa urmatoare. (BUILDER) Toate tipurile de conturi pot efectua operatii de depunere si de extragere monetar, iar contul de credit detine o operatie in plus, si anume una de plata imprumut din soldul curent. Pentru aceasta operatie sunt necesare mai multe verificari, precum verificarea disponibilitatii soldului sau existenta imprumutului. Aplicatia detine si un modul pentru depunerea de bani, urmata de platirea unei sume catre banca, pentru a fi sustrasa din imprumut. Acest modul faciliteaza procesul de depunere de bani si plata, permitand utilizatorului sa realizeze o singura operatie. (FACADE)

Atunci cand se efectueaza o operatie pe un anumit cont, clientul va fi informat (OBSERVER) printr-o metoda aleasa de client la run-time: SMS sau email (STRATEGY).

Aplicatia gestioneaza o referinta catre contul de debit, implementand interfata acestuia si controland accesul la el. (PROXY) Deasemenea, solutia trebuie sa ofere un mecanism unitar care sa centralizeze angajatii companiei, tinand cont de relatiile ierarhice, apartenenta angajatilor la un departament si pachetul salarial pe care il primesc. (COMPOSITE)

# Capitolul 2 – Design Patterns

Conceptul de design pattern a aparut in anii 70, evoluand treptat si ajungand in 1993 la apogeu, cand The Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) au scris o carte ce descrie 23 design pattern-uri.

Un design pattern reprezinta o solutie reutilizabila pentru o problema standard, intr-un anumit context. Deseori sunt confundate cu structurile de date, dar de fapt faciliteaza reutilizarea arhitecturilor si a design-ului software.

Printre avantajele design pattern-urilor se numara:

* permiterea reutilizarii solutiilor standard la nivel de cod sursa/arhitectura
* permiterea documentarii codului sursa/arhitecturilor
* permiterea intelegerii mai facile a codului sursa/arhitecturii
* reprezinta concepte universal cunoscute (vocabular comun)
* sunt solutii testate si foarte bine documentate

Un design pattern este format din:

* *nume* – identifica in mod unic pattern-ul
* *problema* – descrie scopul urmarit, defineste contextul si stabileste cand e aplicabil pattern-ul
* *solutie* – diagrama UML, pseudo-cod ce descrie elementele
* *consecinte* – rezultate, avantaje si dezavantaje

Design pattern-urile sunt de mai multe tipuri:

* creationale – sunt folosite la initializarea si configurarea claselor si a obiectelor
* structurale – utilizate pentru compozitia claselor si obiectelor si decuplarea interfetelor si a claselor
* comportamentale – interactiunea dintre clase si obiecte

## 2.1 Singleton

Singleton-ul este o solutie care se bazeaza pe existenta unei singure instante ce poate fi creata o singura, dar care poate fi referita de mai multe ori. Problema pe care o doreste rezolvata acest design pattern este crearea unei singure instante pentru o clasa, prin care sa fie gestionata o resursa in mod centralizat. Deasemenea, asigura un singur punct de acces, vizibil global, catre unica instanta.

In scenariul ales, este necesara o singura instanta pentru resursa Banca, deoarece se doreste centralizarea clientilor si a conturilor acestora si nu modificarea lor dintr-un alt obiect de tip Banca.

## 2.2 Factory

Pattern-ul Factory are trei cazuri particulare: Simple Factory, Factory Method si Abstract Factory. Toate cazurile sunt bazate pe pe Dependency Inversion Principle.

Simple Factory este unul dintre cele mai folosite in productie, datorita simplitatii in implementare si a avantajelor oferite. Desi este foarte folosit, nu este descris in cartea celor 4, aparand natural, din practica.

Cu ajutorul Factory Method pot fi generate obiecte noi care apartin aceleiasi familii (implementeaza o interfata comuna), iar noi tipuri, din aceeasi familie, pot fi adaugate fara modificari. Printre dezavantajele acestui pattern se numara si faptul ca nu pot fi generate obiecte noi, iar clasele nu pot fi extinse, datorita constructorilor care sunt privati.

Abstract Factory adauga un nou nivel de abstractizare la complexitatea crearii obiectelor. Acest lucru, totusi, este transparent clientului, iar solutia poate fi extinsa prin adaugarea de noi tipuri concrete de obiecte, fara a afecta codul scris.

In modelul prezentat, a fost necesara utilizarea pattern-ului Factory, pentru generarea de conturi diferite, care contin aceleasi atribute: id, sold si tip. Tipurile de conturi sunt cont de debit si cont de credit.

## 2.3 Builder

Builderul ajuta la construirea de obiecte complexe printr-un mecanism independent de procesul de creare a obiectelor. Instanta de tip Builder construieste obiectul, insa tipul acestuia este definit de subclase. Algoritmul de creare a obiectului este flexibil, deoarece clientul alege ce parti sa fie create, dar acest lucru prezinta un dezavantaj, si anume posibilitatea omiterii de atribute atunci cand se creaza obiecte.

La crearea unui cont de credit, clientul are posibilitatea de a lua un imprumut sau de a lasa acest atribut gol.

## 2.4 Flyweight

Flyweight este un design structural ce permite generarea unei multimi de obiecte cu o structura interna complexa si care ocupa un volum mare de memorie. Obiectele au o parte din atribute comune, dar o parte din starea lor variaza. Memoria ocupata de ele poate fi minimizata prin partajarea starii fixe intre ele. Utilizarea unui obiect inseamna reincarcarea starii lui variabile intr-un obiect existent. Printre avantajele acestui design pattern se numara reducerea memoriei ocupate de obiecte prin partajarea lor intre clienti sau a starii lor intre obiecte de acelasi tip. Totusi, Flyweight prezinta si dezavantaje: efectele sunt vizibile pentru solutii in care numarul de obiecte este mare.

Pentru scenariul ales, utilizarea unui design pattern de tip Flyweight a fost necesara in centralizarea conturilor unui client. Clientul poate detine un singur cont de un anumit tip, iar aceasta se realizeaza prin verificarea existentei tipului de cont pentru care se doreste executarea de operatii. Daca nu exista, contul cautat este introdus in lista cu conturi a clientului. Lista de conturi contine obiecte care implementeaza aceeasi interfata (Cont), desi au unele atribute diferite (imprumut).

## 2.5 Facade

Solutia propusa de facade contine o multime de clase, iar executia unei functii presupune apeluri multiple de metode aflate in aceste clase. Este un design pattern util in situatia in care framework-ul creste in complexitate si nu este posibila rescrierea lui pentru simplificare. Apelurile catre multiplele interfete sunt mascate de catre aceasta interfata comuna.

Facade este folosit in modelul prezentat atunci cand se doreste simplificarea operatiei de plata catre banca, dar fondurile sunt insuficiente. Acest pattern permite depunerea unei sume de bani, urmata de plata dorita, reducand astfel timpul pe care un client trebuie sa il consume pentru a efectua operatiile.

## 2.6 Observer

Acest design pattern gestioneaza evenimentele la nivel de interfata, astfel: componentele se aboneaza la acel eveniment, iar la producerea acestuia, pot fi notificate componentele abonate. Observer se foloseste atunci cand anumite componente trebuie sa fie notificate la producerea unui eveniment. Modelul implementeaza conceptul de *loose coupling* – obiectele sunt interconectate prin notificari si nu prin instantieri de clase si apeluri de metode.

Observer detine 2 modele de notificare a observatorului la modificarea starii:

* *push*: obiectul trimite toate detaliile observatorului
* *pull*: obiectul doar notifica observatorul si acesta cere datele cand are nevoie de ele

Proiectul conceput foloseste Observer, in combinatie cu Strategy pentru a notifica clientul cand s-a efectuat o operatie pe un cont detinut de acesta si printr-o modalitate exprimata de client la run-time. Astfel, clientul stie cand este depusa sau extrasa o suma de bani din contul sau.

## 2.7 Proxy

Rolul unui proxy este acela de a interconecta API-uri diferite aflate pe aceeasi masina sau in retea sau de a defini o interfata intre framework-uri diferite.

Design pattern-ul Proxy a fost folosit pentru a gestiona o referinta de tipul ContDebit. S-a verificat nulitatea obiectului prezent in Proxy si s-au executat o serie de instructiuni, in functie de raspuns: daca exista, se realizeaza operatiile; daca nu, se creaza un cont de debit, dupa care au loc operatiile necesare.

## 2.8 Strategy

Strategy este un design pattern comportamental ce permite alegerea la run-time a algoritmului care sa fie utilizat pentru procesarea unor date. Algoritmul se alege pe baza unor conditii descrise la executie, in functie de contextul datelor de intrare.

Printre avantajele prezente se numara alegerea dinamica, la run-time, a metodei de prelucrare a datelor si nelimitarea unui numar maxim de algoritmi ce pot fi folositi.

In modelul prezentat, pattern-ul Strategy s-a folosit pentru a lua decizia informarii clientului, in functie de alegerea facuta la run-time, prin SMS sau prin email.

## 2.9 Composite

Pattern-ul Composite permite gestiunea facila a unor ierarhii de clase ce contin atat primitive, cat si obiecte compuse. Astfel, codul devine mai simplu, deoarece framework-ul nu se rescrie, iar obiectele din ierarhie sunt tratate unitar. Adaugarea de noi componente care respecta interfata comuna nu ridica probleme suplimentare.

Banca din scenariu contine o ierarhie de subordonati, care sunt gestionati prin intermediul design pattern-ului Composite. Astfel, fiecare angajat are o lista de subordonati, care poate sa fie si goala (daca angajatul nu are niciun subordonat).

# Capitolul 3 – Unit Testing

Conceptul de unit testing reprezinta o metoda simpla si rapida de testare a codului sursa de catre programatori. Are loc in faza de dezvoltare si este un instrument destinat programatorilor.

Un unit test este o secventa de cod scrisa de un programator pentru a evalua o parte de mici dimensiuni, din codul sursa testat – clasa sau metoda. Un unit test evalueaza modul de functionare al unei metode intr-un context bine definit.

Unit testul este blocul de baza pentru abordarea TDD (Test-Driven Development). Acest concept reprezinta procesul care se bazeaza pe repetarea unui ciclu de dezvoltare foarte scurt: scrierea unui test automat care esueaza, modificarea cu un minim de cod, pentru a trece testul si rescrierea noului cod.

Pentru scrierea de unit teste, se pot folosi mai multe instrumente, printre care componenta standar in multiple IDE-uri de Java este JUnit. JUnit este un framework de clase ce permite scrierea si executia de teste pentru diferite metode sau clase din cod. A fost scris de Kent Beck si Erich Gamma in timpul unui zbor de la Zurich la Washington, DC..

Printre componentele framework-ului JUnit se gasesc clasele *TestRunner*, care executa teste si raporteaza *TestResults*; *TestCase*, care trebuie extinsa pentru a scrie teste; *Assert* – clasa care evalueaza anumite expresii.

## 3.1 Test Case

Un test case este o clasa ce defineste setul de obiecte pentru a rula mai multe teste. Test case-ul prezinta metode de tipul *setUp()*, care se apeleaza inaintea fiecarei metode de testare si defineste si construieste resursele sau obiectele necesare rularii testelor, dar si de tipul *tearDown(),* care elibereaza/distruge resursele alocate testului, apelandu-se dupa fiecare metoda de testare.

## 3.2 Test Case – clasa ContCredit

Pentru a testa functiile clasei *ContCredit*, a trebuit sa instantiez intr-o metoda marcata cu *@Before* un nou cont de credit.

### 3.2.1 Test initializare

A fost realizat un test pentru verificarea nulitatii contului creat in metoda care se apeleaza inainte. Pentru aceasta, apelarea metodei *assertNotNull()* pe contul creat precedent a fost de ajuns.

### 3.2.2 Test plata

Am declarat 2 variabile, *plata* si *restImprumut*, carora le-am alocat valori, dupa care am executat metoda plateste din contul de credit, cu parametru variabila *plata*. Am presupus ca noul sold, preluat din cont prin metoda *getImprumut()* va fi egal cu valoarea *restImprumut*.

### 3.2.3 Test tip cont

Pentru a testa daca tipul contului a fost asignat corespunzator, am apelat metoda *assertNotNull()* cu parametri *„credit”* si metoda *getTip()* a contului.

## 3.3 Test Case – clasa Banca

Clasa *TestBanca* contine o referinta catre un obiect de tip *Banca* si una catre un obiect de tip *Client*. In metoda marcata cu *@Before*, am preluat o instanta din clasa *Banca*, la care am adaugat un director si un client, necesare pentru testele anterioare.

### 3.3.1 Test singleton

Pentru a testa design pattern-ul Singleton, am preluat intr-un nou obiect de tip *Banca* o referinta, prin metoda statica *getInstance().* Apoi am utilizat metoda *assertSame,* pentru a verifica daca ambele instante conduc catre acelasi obiect.

### 3.3.2 Test primul subordonat

Am dorit verificarea primului subordonat al directorului. Deoarece nu am setat niciun subordonat pentru director, am presupus ca va arunca eroarea *IndexOutOfBoundsException*.

### 3.3.3 Test existenta clienti

La momentul crearii bancii, am adaugat un client, deci pornim de la premiza ca lista de clienti nu este goala. Astfel, prin metoda *assertNotNull* am ajuns la rezultatul dorit.

### 3.3.4 Test decizie notificare prin telefon

Atunci cand am creeat un client nou, am luat decizia notificarii acestuia printr-un email. Astfel, am dorit sa testez daca alegerea a fost salvata, prin metoda *assertNotSame(),* parametrizata cu un nou telefon si cu decizia clientului, preluata prin *getDecizie().*

## 3.4 Test Case – clasa Client

Testarea unei clase *Client* necesita initializarea unei instante dintr-un *ContDebit, ContCredit*, care sunt adaugate in lista unui client. Deasemenea, e nevoie de crearea unui *Observator,* care sa monitorizeze clientul, intrucat vom testa design pattern-ul Observer.

### 3.4.1 Test sold

La creearea anterioara a contului de debit, s-a mentionat si un sold prezent. Astfel, declaram o variabila *sold* pentru a testa daca cele doua coincid. Este folosita metoda *assertEquals.*

### 3.4.2 Test adaugare cont

Am dorit testarea pattern-ului Flyweight, astfel incat am creat un nou *ContDebit* „*debit2”* pe care l-am alocat clientului. In obiectul de tip *ContDebit* „*acelasi”* am preluat contul de debit pe care il detinea deja clientul. Astfel, am presupus, prin *assertSame(),* ca referintele *debit2* si *acelasi* duc catre acelasi obiect, si anume primul *ContDebit* alocat clientului.

### 3.4.3 Test simplificare operatii

Cele doua variabile declarate, *depunere* si *plata*, au fost folosite pe post de parametri pentru functia *platesteDupaDepunere()* din client, iar pentru a testa corecta functionalitate a metodei s-a folosit metoda *assertEquals().*

## 3.5 Test Suite

O suita de teste este o colectie de teste ce vor fi evaluate impreuna. Se poate spune ca o suita este un unit test definit prin combinarea totala sau partiala a altor test case-uri. In schimb, daca se utilizeaza o suita de teste, trebuie asigurat faptul ca unit testele trebuie sa fie definite fara a avea legatura intre ele.

### 3.5.1 Test Suite - Client

Am folosit o colectie de teste pentru a rula toate testele legate de client, si anume clasele *TestClient.class* si *TestContCredit.class*.

### 3.5.2 Test Suite – Complet

Pentru a executa toate testele in acelasi timp, am folosit o suita completa, care contine toate clasele in care se executa teste: *TestClient.class, TestBanca.class* si *TestContCredit.class.*

# Capitolul 4 – Github

Link catre proiectul incarcat in Github: [Repository](https://github.com/andreizamfir/proiectCTS)

# Bibliografie:

„Curs Design Patterns” – Catalin Boja, 2016

„Curs JUnit” – Catalin Boja