

ŞABLOANE DE PROIECTARE

30 octombrie 2023

Cuprins

1	Prezentare generală	1
1.1	Ce este un șablon de proiectare?	1
1.2	De ce sunt utile?	3
1.3	Clasificarea șabloanelor	4
I	Șabloane creaționale	5
2	Singleton	7
3	Factory Method	11
4	Abstract Factory	17
II	Șabloane structurale	23
III	Șabloane comportamentale	25
5	Strategy	27
5.1	Scop și motivare	27
5.2	Aplicare	28
5.3	Structură	29
5.4	Exemplul 1	30
5.5	Exemplul 2	31
5.6	Exemplul al 3-lea	32

6	Observer	35
6.1	Scop și motivare	35
6.2	Aplicare	36
6.3	Structură	37
6.4	Exemple	38
6.4.1	Exemplul 1	38
6.4.2	Exemplul 2	39
6.4.3	Exemplul 3	41
7	Template	45
7.1	Scop și motivare	45
7.2	Aplicare	46
7.3	Structură	46
7.4	Exemplu	48

Capitolul 1

Prezentare generală

Scopul șabloanelor de design este de a înregistra experiență referitoare la programarea obiect-orientată. Fiecare astfel de șablon numește, explică și evaluează un design în sisteme obiect orientate.

1.1 Ce este un șablon de proiectare?

Prima oară, în literatura de specialitate conceptul de șablon (pattern) a apărut în anul 1977 când Christopher Alexander a publicat cartea “A Pattern Language”. În cartea sa el nu a scris despre șabloane de proiectare în domeniul informaticii, ci despre moduri de a automatiza procesul de realizare a unei arhitecturi de succes.

El dă și prima definiție a ceea ce ar putea însemna un șablon de proiectare: *“Fiecare șablon descrie o problemă care tot apare în mediul nostru, după care descrie esența soluției acelei probleme, în așa fel încât ea să poată fi folosită de mii de ori, fără a mai face același lucru de mai multe ori.”*

Această carte a lui care a promovat ideea de șablon în arhitectură i-a făcut pe cei care se ocupau de industria software să se întrebe dacă același lucru nu ar putea fi valabil și pentru domeniul informaticii.

Întrebarea care se punea era următoarea: dacă se poate afirma că un design este bun, cum se poate face ca prin anumiți pași să putem automatiza obținerea unui astfel de design?

Tot Christopher Alexander este cel care a adus în prim plan ideea că, în general, în viața de zi cu zi, aceste șabloane se compun unele cu altele

conducând la soluții mai complexe care agregă mai multe șabloane.

Momentul care a condus la popularizarea pe scară largă a noțiunii de șablon de proiectare în domeniul informaticii a fost apariția în 1995 a cărții *“Design patterns - Elements of Reusable Object Oriented Software”*, carte scrisă de E. Gamma, R. Helm, Johnsson și Vlissides.

În această carte cei patru autori nu au meritul de a fi inventat șabloanele de proiectare pe care le-au prezentat ci mai degrabă ei au documentat ceea ce deja exista în sistemele soft ale vremii respective. Principalele lor merite sunt următoarele:

- au introdus ideea de șablon în industria software
- au catalogat și descris fiecare șablon în parte
- au prezentat într-un mod logic, strategiile care stau la baza acestor șabloane de proiectare.

Această carte este una din cele mai de succes cărți de informatică din toate timpurile. Chiar și la aproximativ 15 ani de la apariția ei, ea este citată și utilizată la fel de mult ca la început, dacă nu chiar și mai mult.

În semn de apreciere, autorii acesteia sunt cunoscuți sub numele de “Gang of Four” (GoF).

Nu numai cei patru care au scris această carte au un merit deosebit la popularizarea șabloanelor de proiectare. Există și alte persoane, care sunt poate chiar mai importante datorită faptului că ele au introdus aceste concepte. Nume precum Kent Beck, Ward Cunningham și James Coplien au contat foarte mult în dezvoltarea acestui domeniu.

O altă definiție a unui șablon de design, care este dată chiar de către cei patru sună astfel: *“Șabloanele de proiectare sunt descrieri ale unor obiecte și clase care comunică și care sunt particularizate pentru a rezolva o problemă generală de design într-un context particular”*.

În continuare prezentăm o ultimă definiție care îi aparține tot lui Christopher Alexander. Chiar dacă ea a fost gândită referitor la arhitectură ea este foarte potrivită în domeniul informaticii: *“Fiecare șablon de proiectare este o regulă în trei părți, care exprimă o relație între un context, o problemă și o soluție”*.

După cum vom vedea în continuare, pentru a descrie un șablon de proiectare, trebuie neapărat să specificăm următoarele lucruri:

- *numele șablonului* - este important deoarece el ne ajută să comunicăm mai ușor cu cei cu care lucrăm
- *problema* - descrierea situației în care se aplică
- *soluția* - modalitatea de a rezolva acea problemă
- *consecințele* - avantajele și dezavantajele acelei abordări

1.2 De ce sunt utile?

Până acum am văzut ce sunt șabloanele de proiectare. O altă întrebare legitimă în acest moment ar fi, de ce sunt ele utile. Oare este neapărat necesar să înțelegem conceptul de șablon de proiectare? Este neapărat necesar să cunoaștem exemple de șabloane de proiectare?

Răspunsul autorului este că da, din mai multe motive:

- Utilizarea șabloanelor de proiectare conduce la reutilizarea unor soluții care și-au arătat de-a lungul timpului eficiența. După cum se știe una din cele mai importante probleme în domeniul informaticii ține de reutilizabilitate. Dacă până acum am tot vorbit despre reutilizarea codului, oare nu se poate ca reutilizarea ideilor să fie la fel de importantă?
- Permite stabilirea unei terminologii comune. Practic, printr-un singur cuvânt se poate ca să comunicăm ceea ce altfel ar fi destul de greu de prezentat.
- Oferă o perspectivă mai înaltă asupra analizei și designului sistemelor obiect orientate.
- Au ca principal obiectiv crearea de cod flexibil, ușor de modificat. Scopul lor este ca, în măsura în care este posibil, să permită adăugarea de noi funcționalități fără a modifica cod existent.
- Odată înțelese bine, ele sunt niște exemple foarte bune relativ la principiile de bază ale programării obiect orientate.
- Permite însușirea unor strategii îmbunătățite care ne pot ajuta și atunci când nu lucrăm cu șabloanele de proiectare:

- Lucrul pe interfețe nu pe implementări
- Favorizarea compoziției în dauna moștenirii
- Găsirea elementelor care variază și încapsularea lor

1.3 Clasificarea șabloanelor

În funcție de nivelul la care apar, șabloanele sunt de mai multe feluri:

- *idioms* - sunt primele care au apărut și sunt dependente de anumite tehnologii (de exemplu, lucrul cu smart pointers în limbajul C++).
- *design patterns* - reprezintă soluții independente de un anumit limbaj, putem spune că sunt un fel de microarhitecturi (ele sunt cele care vor fi prezentate în continuare).
- *framework patterns* - sunt șabloane la nivel de sistem, adică sunt șabloane care sunt folosite pentru a descrie la nivel înalt arhitectura unui întreg sistem.

Există șabloane care de-a lungul timpului au evoluat, de la prima categorie către ultima: MVC (Model View Controller).

Din punctul de vedere al scopului lor, șabloanele de proiectare se împart în 3 categorii:

- creaționale - abstractizează procesul de creare a obiectelor pentru a crește flexibilitatea designului
- structurale - permit gruparea obiectelor în structuri complexe
- comportamentale - permit definirea unui cadru optim pentru realizarea comunicării între obiecte.

În cele ce urmează vom prezenta, pe scurt, un idiom, pentru a putea face ulterior o comparație între el și șabloanele de proiectare.

Partea I

Șabloane creaționale

Capitolul 2

Singleton

Scop

Asigură faptul că o clasă are o singură instanță și oferă un singur punct de acces global la ea.

Motivare

În anumite situații vrem ca o clasă să aibă o singură instanță care să poată fi accesată de oriunde dintr-o aplicație.

Am putea folosi în acest caz o variabilă globală, dar atunci am putea crea oricâte instanțe ale acelei clase.

Am putea de exemplu să avem în obiect un câmp static care să numere câte obiecte au fost create și să genereze eroare când vrem să creem unul nou, dar acest lucru nu este atât de elegant, precum soluția pe care o vom prezenta în continuare și nici nu rezolvă problema accesului global la acea instanță.

Aplicabilitate

Aplicăm acest șablon de proiectare atunci când:

- Vrem să existe o singură instanță a unei clase, care să poată fi accesibilă clienților printr-un punct de acces binecunoscut.

- Când singura instanță trebuie să poată fi extensibilă prin subclasare și clienții ar trebui să fie capabili să folosească o instanță extinsă fără a-și modifica codul.

Structură

Structura șablonului de proiectare Singleton este prezentată în figura 2.1.

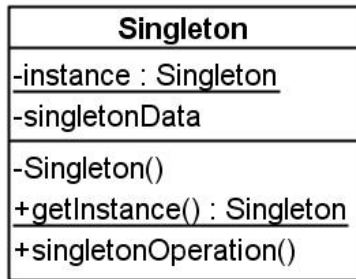


Figura 2.1: Structura șablonului Singleton

După cum se poate observa, pentru a rezolva această problemă trebuie să facem întâi și întâi astfel încât să nu putem crea obiecte de tipul Singleton. Acest lucru se face prin realizarea unui constructor privat.

Un obiect de tip Singleton este reținut într-o instanță statică de tip Singleton. Foarte important este rolul metodei statice `getInstance()` care arată astfel:

```
public static Singleton getInstance()
{
    if(instance==null)
        instance=new Singleton();
    return instance;
}
```

Astfel, dacă atunci când este cerut obiectul acesta există, atunci el este doar returnat fără a se crea altul. Dacă el nu există, atunci este creat și apoi returnat.

Singura modalitate prin care se poate accesa obiectul unic de tip Singleton este prin intermediul metodei `getInstance()`.

Folosirea acestui șablon are mai multe consecințe:

- Prin faptul că nu se folosește o variabilă globală, se asigură nepoluarea spațiului de nume cu o nouă denumire.
- Permite subclasarea unei clase de tip Singleton, astfel încât noua clasă să poată avea ea însăși o singură existență.
- Cu anumite modificări se poate obține o nouă clasă care poate crea un număr limitat de obiecte, dar mai mult decât unul.

Exemplu

În cele ce urmează este prezentat un exemplu care ilustrează modul în care se poate folosi acest șablon de proiectare. Am construit în continuare o clasă `Printer` care simulează o imprimantă. Pentru această clasă trebuie să avem o singură instanță la un moment dat, lucru care este asigurat de folosirea șablonului de proiectare *Singleton*.

```
public class Printer {
    private static Printer instance;

    private Printer()
    {
    }

    public synchronized static Printer getPrinter()
    {
        if(instance==null)
            instance=new Printer();
        return instance;
    }

    public synchronized void print(String str)
    {
        System.out.println("Text de tiparit: "+str);
        try {
            Thread.sleep(2000);
        }
    }
}
```

```
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    System.out.println("Terminat de tiparit...");  
}  
}
```

```
class TiparireThread extends Thread  
{  
    private String textDeTiparit;  
    public TiparireThread(String str)  
    {  
        textDeTiparit=str;  
    }  
  
    public void run()  
    {  
        Printer p=Printer.getPrinter();  
        p.print(textDeTiparit);  
    }  
}
```

```
public class TestSingleton {  
    public static void main(String[] args) {  
        TiparireThread t1=new TiparireThread("primul text...");  
        TiparireThread t2=new TiparireThread("al doilea text...");  
        t1.start();  
        t2.start();  
    }  
}
```

Capitolul 3

Factory Method

Încărcarea dinamică a claselor

Încărcarea dinamică a claselor este mecanismul prin care în limbajul Java se poate încărca și folosi în timpul rulării programului, o clasă care nu există neapărat la momentul compilării.

Acest lucru se întâmplă, de exemplu, în momentul în care lucrăm cu baze de date atunci când scriem cod precum:

```
Class.forName("com.mysql.jdbc.Driver");
```

În plus, în momentul în care se încarcă dinamic o clasă se execută și inițializatorul ei static. Acesta este o construcție de tipul următor, care poate inițializa attribute statice și eventual efectuează anumite operații în plus:

```
class Test
{
    static
    {
        ... codul pentru initializator
    }
    ...
}
```

În cele ce urmează vom prezenta codul pentru o aplicație care folosește încărcarea dinamică a claselor și inițializatorul static.

```
public interface Interfata {
    void metoda();
}

public class TestDinamic {
    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException, ClassNotFoundException {
        Interfata i;
        System.out.println("incepe programul...");
        i=(Interfata)Class.forName("Clasa").newInstance();
        i.metoda();
    }
}

public class Clasa implements Interfata{
    static {
        System.out.println("in initializatorul static...");
    }

    public void metoda(){
        System.out.println("s-a apelat metoda...");
    }
}
```

Scop

Definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă care clasă trebuie instanțiată. Permite unei clase să transfere responsabilitatea instanțierii către subclase.

Motivare

În această secțiune vom prezenta ceea ce se întâmplă într-un framework în care dorim să creem mai multe tipuri de documente diferite. Totuși, pentru aceste tipuri de documente, chiar dacă ele sunt diferite, există anumite lucruri care se petrec la fel.

În acest caz trebuie să separăm lucrurile care sunt comune de cele care variază, lucru ilustrat în figura 3.1

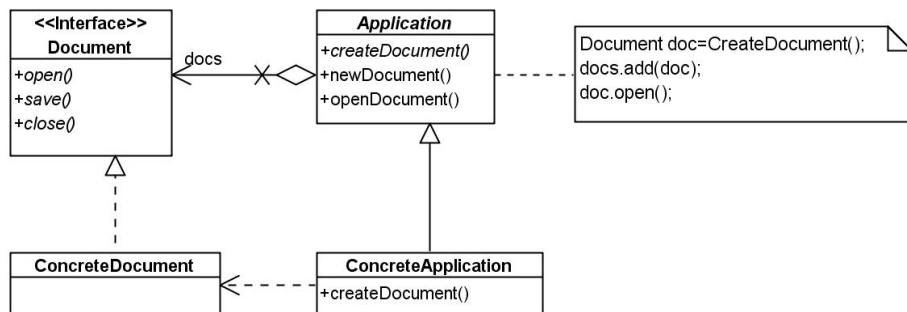


Figura 3.1: Motivare Factory Method

În această diagramă partea comună este reprezentată de interfața **Document** (partea comună tuturor documentelor) și de clasa abstractă **Application** (partea comună tuturor aplicațiilor).

Aplicabilitate

Se folosește *Factory Method* atunci când:

- O clasă nu poate anticipa clasele obiectelor pe care va trebui să le creeze.
- O clasă vrea ca subclassele sale să specifice obiectele pe care le creează.

Structură

Structura acestui șablon este prezentată în diagrama 3.2.

Se poate observa în figura 3.2 că avem o clasă abstractă **Creator** în care avem definită o metodă abstractă care are rolul de a crea și returna un anumit tip de obiect de tipul **Product**.

Această metodă este rescrisă într-o subclassă pentru a realiza o anumită operație.

Trebuie precizat faptul că acest șablon are mai multe consecințe:

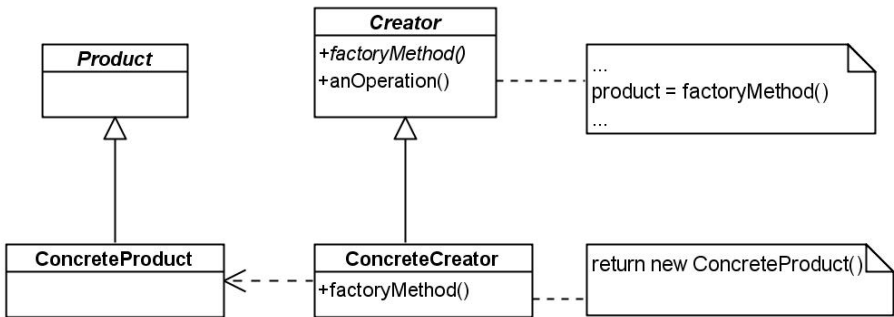


Figura 3.2: Structura șablonului Factory

- Permite crearea unui framework generic în care ulterior se poate adăuga funcționalitate, prin suprascrierea unor anumite metode și folosirea polimorfismului. Este o ilustrare a principiului: "Când ceva variază acel lucru trebuie încapsulat". În cazul nostru s-a încapsulat funcționalitatea comună în clasa **Creator**.
- Crearea unui obiect printr-o metodă factory este mai bună pe termen lung deoarece astfel se poate foarte ușor aplica aceeași logică prin crearea unei noi clase și suprascrierea unei singure metode.

Există, în practică și metode factory care primesc un parametru care specifică tipul obiectului care va fi creat.

Exemplu

Am ales ca exemplu pentru acest capitol o aplicație care implementează un pic diferit șablonul Factory Method, dar care ne va ajuta să înțelegem mai bine șablonul de proiectare din capitolul următor.

În primul rând definim o interfață **Shape**:

```
public interface Shape {
    void draw();
}
```

După aceasta definim o clasă **ShapeFactory** care știe să creeze obiecte de tip **Shape** pe baza unui șir de caractere transmis ca parametru.

```

import java.util.*;

public abstract class ShapeFactory {
    protected abstract Shape create();

    private static Map<String, ShapeFactory> factories=
        new HashMap<String, ShapeFactory>();

    public static void addFactory(String id, ShapeFactory f) {
        factories.put(id, f);
    }

    public static final Shape createShape(String id) {
        if(!factories.containsKey(id)) {
            try {
                Class.forName(id);
            }
            catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            if(!factories.containsKey(id))
                throw new RuntimeException("This id does not exist...");
        }
        return ((ShapeFactory)factories.get(id)).create();
    }
}

```

Avem două clase **Circle** și **Square** care se înregistrează singure în fabrica de obiecte prezentată anterior:

```

public class Circle implements Shape{
    private Circle(){};
    public void draw() {
        System.out.println("Circle.draw()...");
    }
    private static class Factory extends ShapeFactory {
        protected Shape create() {
            return new Circle();
        }
    }
}

```

```

    }
}

static {
    ShapeFactory.addFactory("Circle", new Factory());
}
}

public class Square implements Shape{
    private Square(){};
    public void draw() {
        System.out.println("Square.draw()...");
    }
    private static class Factory extends ShapeFactory {
        protected Shape create() {
            return new Square();
        }
    }

    static {
        ShapeFactory.addFactory("Square", new Factory());
    }
}

```

În cele din urmă prezentăm o mică aplicație care ilustrează modul în care se pot folosi clasele prezentate anterior:

```

import java.util.*;
public class TestPolymorphicFactories {
    public static void main(String[] args) {
        String[] list={"Circle","Square","Circle"};
        List<Shape> shapes=new ArrayList<Shape>();
        for(String shapeName:list)
            shapes.add(ShapeFactory.createShape(shapeName));
        for(Shape shape:shapes)
            shape.draw();
    }
}

```

Capitolul 4

Abstract Factory

Scop

Oferă o interfață pentru crearea familiilor de obiecte înrudite sau dependente fără a specifica clasele lor concrete.

Motivare

Un astfel de șablon de proiectare se folosește în JAVA pentru a construi cadrul pentru lucrul cu baze de date în JDBC. Iată, în figura 4.1 modul în care sunt definite principalele clase și interfețe folosite pentru lucrul cu baze de date în Java.

După cum se poate ușor observa, pentru a lucra cu baze de date se folosesc în codul pe care-l scriem mai mult interfețe decât clase propriu-zise. Avem astfel un exemplu care ilustrează lucrul pe interfețe nu pe implementări.

Acest lucru este necesar deoarece, cei care au făcut limbajul Java nu au putut scrie codul pentru fiecare tip de sistem de gestiune al bazelor de date care ar putea fi folosit împreună cu limbajul Java, ci mai degrabă au lăsat acest lucru pe seama implementatorilor SGBD-urilor.

Astfel, noi când lucrăm cu baze de date în Java, de fapt lucrăm pe niște interfețe, codul propriu zis care este apelat, pentru a realiza conexiuni cu serverul de baze de date și pentru a efectua diverse operații fiind cuprins în driver-ul pe care îl folosim.

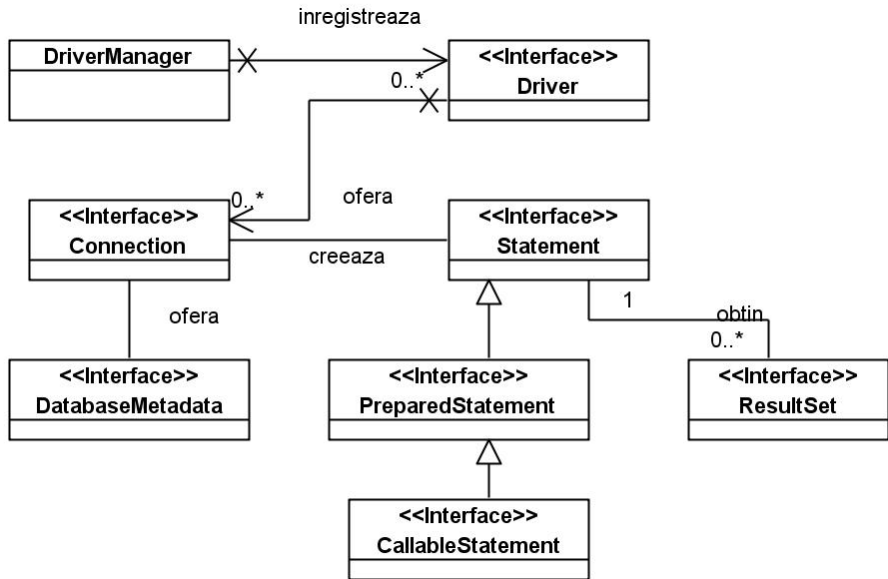


Figura 4.1: Arhitectura JDBC

Reamintim că un driver este un fișier JAR care trebuie inclus la momentul *rulării* aplicației. El este responsabil cu tot ceea ce ține de comunicarea cu serverul de baze de date.

Această arhitectură prezintă mai multe avantaje, precum faptul că în acest fel numărul de SGBD-uri suportate de Java este practic nelimitat. Oricine vrea să-și construiască un SGBD, poate face acest lucru și, mai mult, după ce-și construiește propriul driver, poate accesa acest SGBD din Java.

În continuare este prezentată o mostră de cod care accesează o bază de date în limbajul Java:

```

Class.forName("com.mysql.jdbc.Driver");
conn=DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/bazamea","root","");
Statement s=conn.createStatement();

ResultSet rs=s.executeQuery("select * from persoane");
  
```

```

while(rs.next())
{
    long id=rs.getLong("persoana_id");
    String nume=rs.getString("nume");
    String prenume=rs.getString("prenume");
    String oras=rs.getString("oras");
    Date d=rs.getDate("data_nasterii");
    SimpleDateFormat sdf=new SimpleDateFormat("dd/MM/yyyy");
    System.out.println(String.format("%-20s %10s %12s", nume+
                                     " "+prenume, oras, sdf.format(d)));
}
conn.close();

```

Vom prezenta în secțiunea dedicată exemplurilor ce se întâmplă de fapt în acest cod.

Aplicabilitate

Se folosește *Abstract Factory* atunci când:

- Un sistem trebuie să fie independent de modul în care produsele sale sunt create, compuse și reprezentate.
- Un sistem trebuie să poată fi configurat cu una din mai multe familii de produse.
- Mai multe produse de același tip trebuie folosite împreună și vrem să forțăm această constrângere.
- Vrem să oferim o librărie de clase, și vrem să lucrăm doar pe interfețele lor nu și pe implementări.

Structură

Structura acestui șablon este prezentată în figura 4.2

După cum se poate vedea avem o interfață **AbstractFactory** care descrie modul în care se pot crea obiectele. Această interfață este implementată de câte o clasă pentru fiecare tip de familie de obiecte în parte.

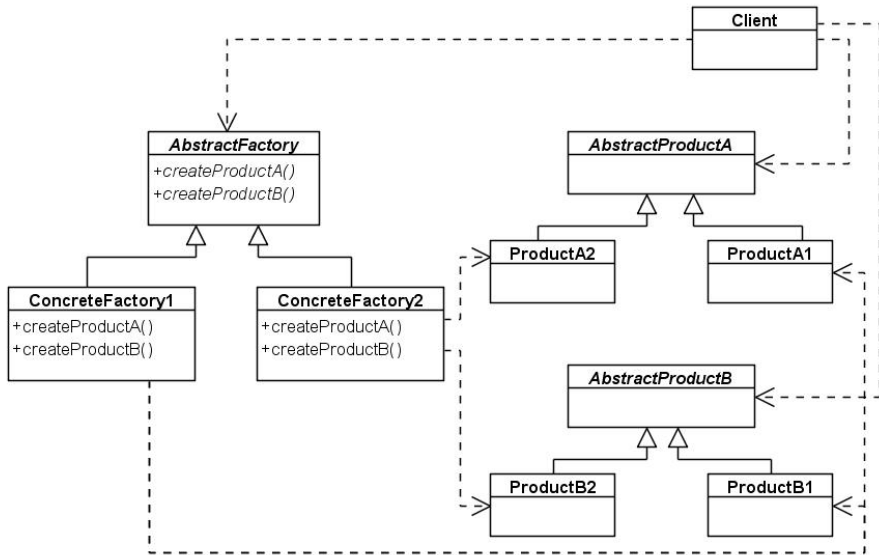


Figura 4.2: Structura șablonului Abstract Factory

De asemenea, pentru fiecare tip de obiect în parte există câte o interfață care descrie funcționalitatea lui.

Folosirea acestui șablon are mai multe consecințe:

- Prin folosirea fabricii de obiecte se izolează clienții de produsele efective pe care le folosesc. Acest lucru reprezintă o modalitate de asigurare a încapsulării.
- Folosirea Abstract Factory ajută la schimbarea ușoară a familiilor de obiecte. Pentru a realiza acest lucru trebuie schimbată o singură linie de cod.
- Se promovează consistența între diversele familii de produse.
- Un posibil dezavantaj al acestei abordări este faptul că este mai greu de adăugat un nou tip de produs. Acest lucru ar presupune modificarea mai multor clase (toate cele din lanțul de derivare al lui **AbstractFactory**) precum și introducerea mai multor clase corespunzătoare noului produs.

Exemplu

În secțiunea referitoare la acest șablon de proiectare vom prezenta modul în care ar trebui folosită o familie de produse precum și modul în care se poate modifica familia de produse folosită.

```
AbstractFactory factory=new ConcreteFactory1();
//AbstractFactory factory=new ConcreteFactory2();
```

```
AbstractProductA pa=factory.createProductA();
AbstractProductB pb=factory.createProductB();
```

A doua parte a acestei secțiuni va prezenta ce se întâmplă în spatele cortinei atunci când vrem să ne conectăm la o bază de date în Java:

1. Se încarcă dinamic clasa din pachetul JAR corespunzătoare driverului cu care vrem să lucrăm.
2. Această clasă are un inițializator static care este executat. În acest inițializator este asociat un obiect de tip `Driver` corespunzător driverului utilizat (`com.mysql.jdbc.Driver` dacă lucrăm cu baze de date MySQL) unui șir de caractere care descrie acel driver (de ex `"jdbc:mysql://"`).
3. Când se apelează metoda `getConnection()` a clasei `DriverManager`, de fapt se parcurge lista de drivere, se selectează cel corespunzător șirului de caractere transmis și cu ajutorul acestuia se creează o nouă conexiune, care este returnată (un fel de `com.mysql.jdbc.Connection`). Această conexiune reprezintă fabrica concretă de obiecte.
4. Cu ajutorul acestei fabrici de obiecte se creează mai multe obiecte de tipul `Statement`, `PreparedStatement` etc.

Pentru a putea înțelege foarte bine corespondențele între clasele existente în cazul lucrului cu baze de date și a șablonului de proiectare `Abstract Factory`, este bine să vedem ce rol are fiecare clasă:

- *AbstractFactory* - `java.util.Connection`
- *ConcreteFactory* - `com.mysql.jdbc.Connection`

- *AbstractProduct* - `Statement`, `PreparedStatement`, `CallableStatement`
- *ConcreteProduct* - `com.mysql.jdbc.Statement`, etc.

Cel mai bine se vede acest lucru dacă urmărim figura 4.3.

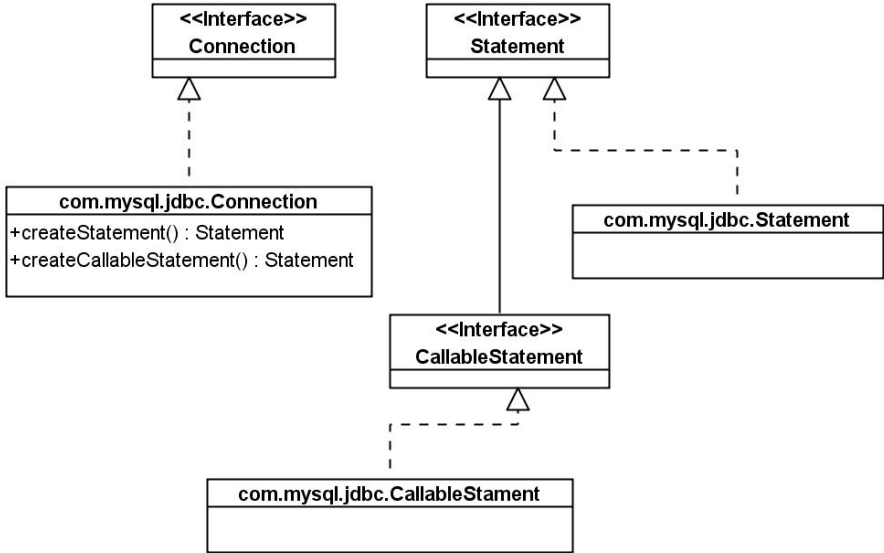


Figura 4.3: Ilustrare Abstract Factory în JDBC

Partea II

Șabloane structurale

Partea III

Șabloane comportamentale

Capitolul 5

Strategy

5.1 Scop și motivare

Scop: Definește o familie de algoritmi, îl încapsulează pe fiecare și îi face interschimbabili. Permite varierea independentă a algoritmului de clasa care îl utilizează.

Motivare: Există numeroși algoritmi pentru spargerea unui stream în linii. Introducerea algoritmului în clasa care-l folosește nu este cea mai bună soluție din mai multe motive:

- Clienții vor deveni cu atât mai complecși cu cât numărul algoritmilor suportați va fi mai mare.
- Nu toți algoritmi vor fi folosiți de fiecare dată și nu vrem să suportăm algoritmi care s-ar putea să nu fie folosiți.
- Este dificil de adăugat noi algoritmi și de a-i varia pe cei existenți fără a fi nevoie să modificăm codul curent.

Principiul pe care vrem să-l respectăm este principiul Open-Closed care spune că:

- Un sistem trebuie să fie *deschis la extensie* (să fie construit astfel încât să permită adăugarea de noi funcționalități).
- Un sistem trebuie să fie *închis la modificări* (adăugarea noilor funcționalități nu trebuie să necesite schimbarea codului existent).

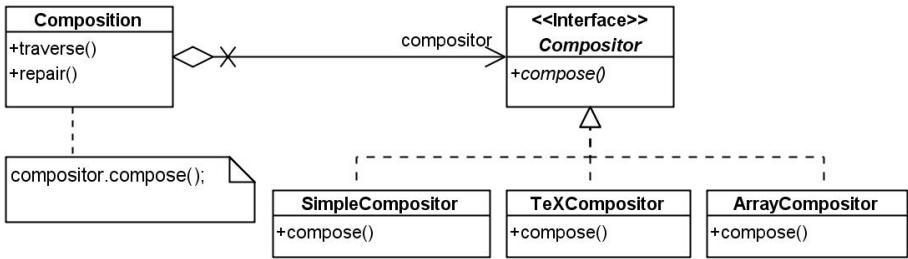


Figura 5.1: Motivare utilitate Strategy

Se poate rezolva această situație prin crearea unei interfețe care să reprezinte un algoritm, iar pentru fiecare implementare propriu zisă a algoritmului să avem câte o clasă separată.

Clasa care folosește algoritmul va reține o referință la obiectul care va conține algoritmul propriu-zis. Avantajul acestei abordări este că algoritmul poate varia la momentul rulării.

5.2 Aplicare

Se folosește șablonul Strategy atunci când:

- Mai multe clase înrudite diferă numai prin comportament. Strategiile oferă o modalitate de configurare a unei clase cu unul din mai multe comportamente.
- Avem variante diferite ale unui algoritm. De exemplu, s-ar putea să definim algoritmi diferiți din punctul de vedere al raportului spațiu/timp. Strategiile pot fi folosite când acești algoritmi sunt implementați în ierarhii de clase.
- Un algoritm folosește date de care clienții n-ar trebui să știe. Se evită expunerea de structuri complexe specifice algoritmului.
- O clasă definește mai multe comportamente și acestea apar apelate în urma mai multor instrucțiuni condiționale complexe în interiorul operațiilor clasei. Fiecare ramură a condițiilor este mutată într-o clasă separată.

5.3 Structură

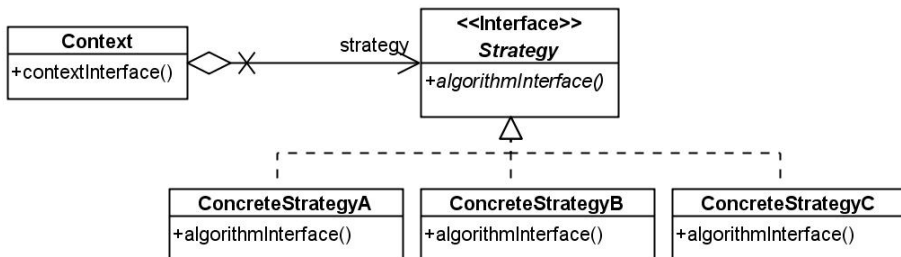


Figura 5.2: Structură șablon Strategy

Există două metode de transmitere a parametrilor către strategie:

- Se poate ca toate datele necesare algoritmului să fie transmise ca parametri atunci când se apelează acesta.
- Se poate ca atunci când este apelat algoritmul să i se transmită o referință la obiectul **Context** și obiectul strategie să apeleze metode ale clasei **Context** pentru a obține datele necesare.

Ca și modalitate de lucru, în general, se parcurg următorii pași:

1. Se creează obiectul **Context**
2. La crearea acestuia sau ulterior i se transmite de către client o strategie care va fi folosită. Ulterior, clientul are posibilitatea, dacă dorește să schimbe strategia.
3. Clientul interacționează numai cu obiectul **Context**, acesta lucrând indirect cu obiectul strategie curent.

Folosirea acestui șablon are mai multe consecințe (avantaje și dezavantaje):

- Este necesară construirea unei ierarhii de clase cu algoritmi. Acest lucru poate fi bun deoarece oferă un suport pentru factorizarea în clasa de bază a comportamentului comun.
- Oferă o alternativă foarte bună la subclasare. S-ar fi putut să derivăm direct clasa **Context** dar acest lucru nu ar fi fost așa de bun din mai multe motive:


```

public class TestCompare {
    public static void main(String[] args) {
        String[] s={"Angela","Adriana","Viorel","Mariana","Cristian"};
        Arrays.sort(s, new Comparator<String>()
        {
            public int compare(String s1, String s2) {
                return s1.compareToIgnoreCase(s2);
            }
        });
        for(int i=0;i<s.length;i++)
            System.out.print(s[i]+" ");
    }
}

```

5.5 Exemplul 2

Problemă: Să se implementeze același lucru folosind facilitățile oferite de mediul Java și C#.

```

import java.util.Arrays;

public class TestCompare {
    public static void main(String[] args) {
        String[] s={"Angela","Adriana","Viorel","Mariana","Cristian"};
        Arrays.sort(s, new java.util.Comparator<String>()
        {
            public int compare(String s1, String s2) {
                return s1.compareToIgnoreCase(s2);
            }
        });
        for(int i=0;i<s.length;i++)
            System.out.print(s[i]+" ");
    }
}

```

Exemplul corespunzător C# este prezentat în figura 5.3.

Strategy Pattern with IComparer

```

class CoolComparer : IComparer
{
    #region IComparer Members

    public int Compare(object x, object y)
    {
        // TODO: implementation
        return 0;
    }

    #endregion
}

ArrayList items = new ArrayList();

items.Add("One");
items.Add("Two");
items.Add("Three");

items.Sort(); // Uses IComparable on string object

IComparer myComparer = new CoolComparer();
items.Sort(myComparer); // Delegate Comparison Method

```

Figura 5.3: Utilizare Strategy in C#

5.6 Exemplul al 3-lea

Problemă: Să se implementeze un joc de X&O folosind șablonul Strategy pentru stabilirea dificultății jocului.

```

public class TestGame {
    public static void main(String[] args) {
        XOGame game=new XOGame();
        game.setLevel(GameLevel.EASY);
        //playing the game
    }
}

public interface GameLevel {

```

```

void makeMove(int[] [] a, int playerNo);

final static GameLevel EASY=new GameLevel() {
    public void makeMove(int[] [] a, int playerNo) {
        //TODO de implementat metoda makeMove
    }
};

final static GameLevel MEDIUM=new GameLevel() {
    public void makeMove(int[] [] a, int playerNo) {
        //TODO de implementat metoda makeMove
    }
};

final static GameLevel DIFFICULT=new GameLevel() {
    public void makeMove(int[] [] a, int playerNo) {
        //TODO de implementat metoda makeMove
    }
};
}

public class XOGame {
    private GameLevel level;
    private int[] [] a;
    private int currentPlayer;

    public void setLevel(GameLevel level) {
        this.level=level;
    }

    public void makeComputerMove() {
        level.makeMove(a, currentPlayer);
        currentPlayer=3-currentPlayer;
    }

    //TODO other methods to write
}

```


Capitolul 6

Observer

6.1 Scop și motivare

Scop: Definește o dependență de la unul la mai mulți între obiecte astfel încât atunci când un obiect își schimbă starea, toate cele dependente de el să fie notificate și actualizate automat.

Motivare: Programarea obiect orientată conduce la crearea mai multor clase, între care există mai multe legături. De multe ori, atunci când conținutul unui obiect se schimbă mai multe alte obiecte care depind de acel obiect, trebuie anunțate.

Acest lucru trebuie realizat, pe cât posibil, astfel încât să nu fie cuplate foarte mult clasele.

Un astfel de exemplu este un document Excel în care avem mai multe reprezentări ale acelorași date. În momentul în care se modifică anumite date, automat toate reprezentările trebuie actualizate. Un exemplu este prezentat în figura 6.1.

În această figură există un subiect, care reține date, precum și mai mulți observatori care își schimbă reprezentările dacă datele din subiect se modifică.

Mai mult, numărul de observatori este nedefinit, ceea ce înseamnă că se pot adăuga oricând alți observatori.

Ca observație suplimentară, trebuie remarcat că observatorii pot fi de tipuri foarte variate, lucru care face și mai util acest șablon.

Practic, într-o aplicație care folosește acest șablon se întâmplă următoarele operații:

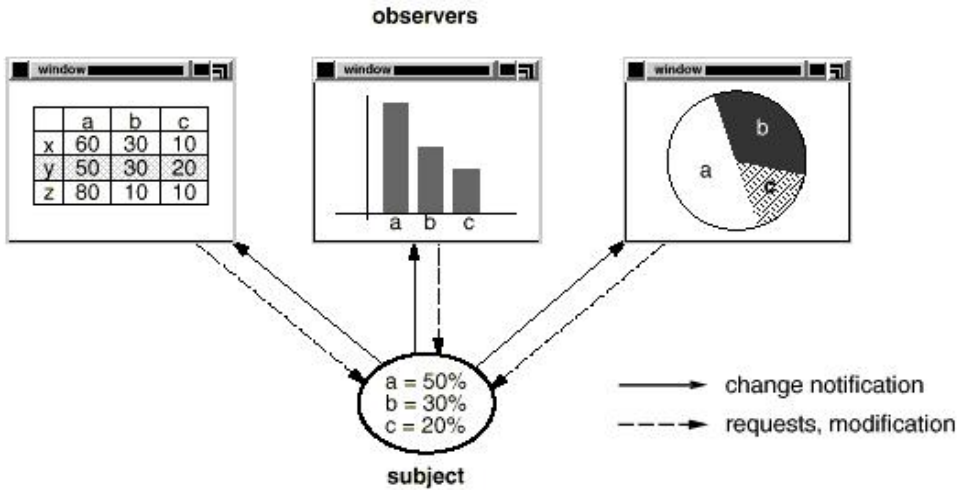


Figura 6.1: Motivare utilitate Observer

- Este creat un anumit obiect de tip subiect.
- Sunt create obiectele observator care se înregistrează în lista de observatori a subiectului.
- În momentul în care subiectul se modifică, automat toate obiectele care au fost înregistrate sunt anunțate.
- În plus, la orice moment, se poate să se înregistreze noi obiecte care din acel moment vor fi anunțate de modificări ale subiectului, sau să se dezînregistreze obiecte care nu vor mai fi anunțate din acel moment.

Evident, pentru a putea să-și actualizeze starea, fiecare observator reține o referință la obiectul model.

6.2 Aplicare

Se folosește acest șablon în una din următoarele situații:

- Când o abstracție are două aspecte, unul dependent de altul. Încapsularea acestor aspecte în obiecte separate permite varierea și reutilizarea lor independentă.

- Când o schimbare într-un obiect necesită schimbarea altora și nu se știe câte astfel de obiecte trebuie schimbate.
- Când un obiect ar trebui să fie capabil să notifice alte obiecte fără a putea face presupuneri despre cine sunt aceste obiecte. Cu alte cuvinte, vrem evitarea cuplării puternice între obiecte.

6.3 Structură

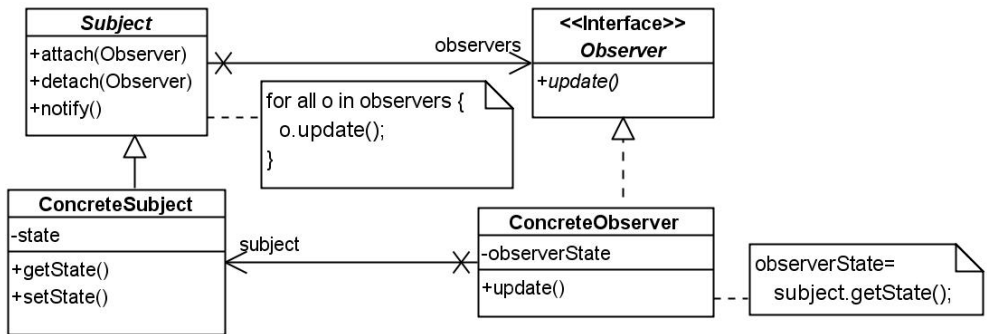


Figura 6.2: Structură șablon Observer

Folosirea acestui șablon are mai multe consecințe:

- Cuplarea între subiect și observator este una abstractă. Un subiect nu știe nici tipul concret al observatorilor, nici numărul acestora. Un astfel de design, poate facilita comunicarea între layer-uri diferite fără a strica designul aplicației.
- Avem o comunicare de tip broadcast. Subiectul nu transmite mesaje personalizate fiecărui obiect, ci transmite același mesaj tuturor, doar obiectele interesate tratând acel mesaj.
- Trebuie lucrat cu grijă, deoarece în urma modificărilor asupra subiectului, se pot produce mai multe actualizări ale observatorilor decât ne-am aștepta. Uneori, are sens să transmitem un mesaj care să descrie modificarea care s-a produs, pentru a fi mai ușor de tratat de către observatori.

- Uneori, dacă putem avea mai multe obiecte observate de către un observator, este necesar să putem să știm și cine este obiectul care a trimis notificarea.
- O problemă interesantă este cine ar trebui să stabilească când se fac notificările? Exista două soluții:
 - Se poate ca oricând se modifică ceva în starea subiectului, toți observatorii să fie anunțați. Această modalitate prezintă un avantaj și un dezavantaj.

Avantajul ar fi că nu mai trebuie ca utilizatorul să-și aducă aminte automat să apeleze o anumită metodă.

Dezavantajul ar fi că, în acest fel, s-ar putea ca mai multe actualizări consecutive să conducă la notificări în cascadă.
 - Se poate ca nici o notificare să nu se facă până când clientul nu cere acest lucru. Problema, în acest caz este că se poate ca clientul să uite să apeleze acea metodă.
- Există două modele de subiecți:
 - Cei care transmit foarte multe detalii legate de schimbarea de stare care s-a produs în interiorul lor (modelul **push**)
 - Cei care nu transmit aproape nici o informație despre schimbarea care s-a produs (modelul **pull**).
- Se poate ca observatorii să se înregistreze la un subiect și din punctul de vedere al aspectului de care sunt interesați. Astfel se elimină numeroase apeluri inutile.

6.4 Exemple

6.4.1 Exemplul 1

În unele limbaje fiecare componentă are o metodă specială `CompName.Click`, în care se poate pune codul pe care vrem să-l executăm atunci când se dă click pe acea componentă. Componenta răspunde unui număr fix de evenimente care nu poate fi schimbat.

În alte limbaje, gen limbajul C, trebuie ca programatorul să facă foarte multe lucruri, inclusiv o structură repetitivă în care să se tot verifice dacă s-au produs noi evenimente.

Mediul Java promovează o soluție intermediară. Programatorul controlează modul în care evenimentele sunt transmise de la *event source* la *listeners*. În Java orice obiect poate fi *listener*.

Acest model, *event delegation model* este un pic mai flexibil față de modelele anterioare, dar necesită, mai mult cod decât modul de lucru ca în prima situație prezentată.

Este un mod de lucru în care o sursă de evenimente poate avea înregistrați mai mulți observatori și de asemenea, un observator poate fi legat la mai multe surse de evenimente, eventual chiar diferite.

Informația despre eveniment este încapsulată într-un obiect *event*, derivat din `java.util.EventObject`.

Atunci când vrem să adăugăm un nou listener, în Java scriem un cod de genul următor:

```
eventSource.addEventListener(eventListenerObject);
```

Un listener este de fapt un observator în această situație, el trebuind să implementeze o interfață cum ar fi, de exemplu `ActionListener`.

Când se execută un click pe un buton, în interiorul lui se creează un eveniment de tipul `ActionEvent` și se apelează:

```
listener.actionPerformed(event);
```

Evident, putem avea oricâți listeneri, de tipuri foarte variate.

6.4.2 Exemplul 2

În cele ce urmează prezentăm un mic exemplu concret de implementare a șablonului *Observer*.

```
import java.util.*;

public class Subject {
    private ArrayList<Observer> observers;

    public Subject() {
        observers=new ArrayList<Observer>();
    }
}
```

```
}

public void attach(Observer o) {
    observers.add(o);
}

public void detach(Observer o) {
    observers.remove(o);
}

public void notifyAllObservers() {
    for(Observer o:observers)
        o.update(this);
}
}

public interface Observer {
    void update(Subject s);
}

import java.util.*;

public class Lista extends Subject {
    private ArrayList<String> persoane;

    public Lista() {
        persoane=new ArrayList<String>();
    }

    public void adaugaPersoana(String s) {
        persoane.add(s);
        notifyAllObservers();
    }

    public void stergePersoana(String s) {
        persoane.remove(s);
        notifyAllObservers();
    }

    public String toString() {
        StringBuilder sb=new StringBuilder();
        for(String s:persoane)
            sb.append(s).append(" ");
        return sb.toString();
    }
}
```

```

public class Observer implements Observer{
    private String name;
    private Lista subject;

    public Observer(String name, Lista subject) {
        this.name=name;
        this.subject=subject;
        subject.attach(this);
    }

    public void update(Subject s) {
        System.out.println(name+": S-a schimbat continutul listei...");
        System.out.println(name+": Noul continut: "+subject);
        System.out.println();
    }
}

public class TestObserver {
    public static void main(String[] args) {
        Lista lista=new Lista();
        Observer obs1=new Observer("OBS 1", lista);
        Observer obs2=new Observer("OBS 2", lista);
        lista.adaugaPersoana("Adi");
        lista.adaugaPersoana("Marcel");
        lista.detach(obs1);
        lista.stergePersoana("Adi");
    }
}

```

6.4.3 Exemplul 3

În cadrul platformei JDK există suport pentru implementarea acestui șablon. Modul de lucru cu clasa **Observable** precum și cu interfața **Observer**, ambele din pachetul **java.util** este prezentat în cele ce urmează.

```

import java.util.*;

public class Lista extends Observable{
    private ArrayList<String> persoane;

    public Lista() {

```

```
    persoane=new ArrayList<String>();
}

public void adaugaPersoana(String s) {
    persoane.add(s);
    setChanged();
    notifyObservers();
}

public void stergePersoana(String s) {
    persoane.remove(s);
    setChanged();
    notifyObservers();
}

public String toString() {
    StringBuilder sb=new StringBuilder();
    for(String s:persoane)
        sb.append(s).append(" ");
    return sb.toString();
}
}

import java.util.Observable;
import java.util.Observer;

public class Observator implements Observer{
    private String name;
    private Lista subject;

    public Observator(String name, Lista subject) {
        this.name=name;
        this.subject=subject;
        subject.addObserver(this);
    }

    public void update(Observable obs, Object o) {
        System.out.println(name+": S-a schimbat continutul listei...");
        System.out.println(name+": Noul continut: "+subject);
        System.out.println();
    }
}

public class TestObserver {
    public static void main(String[] args) {
        Lista lista=new Lista();
```

```
    Observer obs1=new Observer("OBS 1", lista);
    Observer obs2=new Observer("OBS 2", lista);
    lista.adaugaPersoana("Adi");
    lista.adaugaPersoana("Marcel");
    lista.deleteObserver(obs1);
    lista.stergePersoana("Adi");
}
}
```


Capitolul 7

Template

7.1 Scop și motivare

Scop: Definește scheletul unui algoritm într-o operație, lasând anumiți pași în sarcina subclaselor. **Template Method** lasă subclasele să redefinească anumiți pași ai unui algoritm fără a schimba structura algoritmului.

Motivare: Să presupunem că avem un framework în care avem clasele **Application** și **Document**. Clasa **Application** se ocupă, printre altele, cu deschiderea documentelor în timp ce clasa **Document** reprezintă documentul propriu-zis.

Când avem o aplicație particulară subclasăm fiecare din aceste subclase, pentru a rezolva problema respectivă. Am putea avea ceva corespunzător figurii 7.1.

Conținutul metodei `openDocument()` ar putea arăta astfel:

```
public void openDocument() {  
    if(!canOpenDocument())  
        return;  
    Document doc = doCreateDocument();  
    if(doc) {  
        docs.addDocument(doc);  
        aboutToOpenDocument(doc);  
        doc.open();  
        doc.doRead();  
    }  
}
```

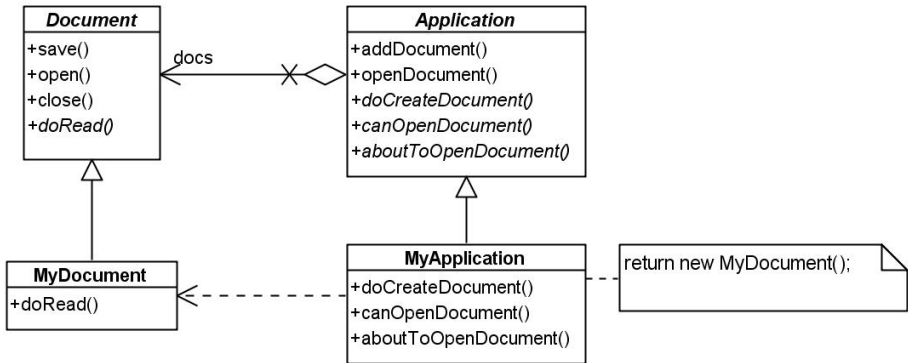


Figura 7.1: Motivare utilitate Template

}

În acest exemplu, `openDocument()` poartă numele de **metodă template**. Această metodă definește un algoritm și ordinea în care se produc diversele operații, permițând fiecărei subclase definirea anumitor operații.

7.2 Aplicare

Acest șablon de proiectare ar trebui utilizat:

- Pentru a implementa părțile invariante ale unui algoritm și a lăsa subclasele să implementeze părțile care variază.
- Atunci când comportamentul comun între subclase ar trebui factorizat și localizat într-o clasă comună pentru a evita duplicarea.
- Pentru a controla extensiile realizate cu ajutorul subclaselor. Se poate defini o metodă template care apelează operațiile "hook" în anumite puncte specifice, permițând extensia numai prin aceste puncte.

7.3 Structură

După cum se poate vedea ideea acestui șablon este de a avea o clasă de bază în care unele metode sunt definite și altele nu. Metodele abs-

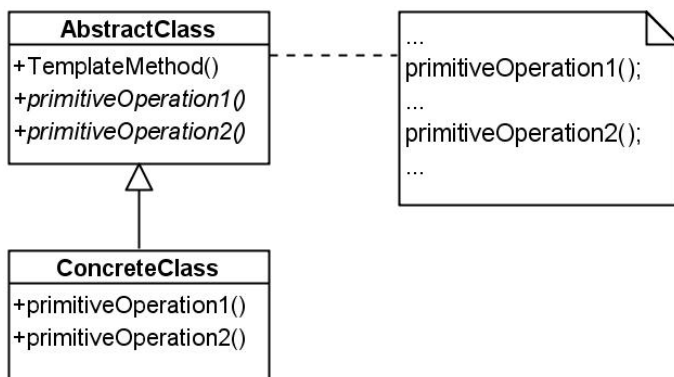


Figura 7.2: Structură șablon Template

trakte sunt folosite de celelalte metode, după care sunt definite ulterior în subclase.

Șablonul reprezintă o tehnică foarte utilă pentru reutilizarea codului și este folosit cu precădere în framework-uri.

De multe ori, este o ilustrarea a principiului ”don’t call us, we’ll call you” (principiul Hollywood), deoarece metodele din clasa de bază le apelează pe cele din clasa derivată, nu invers.

Metodele template pot apela mai multe tipuri de metode:

- operații concrete (posibil din subclase)
- operații concrete din clasa abstractă
- metode Factory
- ”hook operations” - sunt metode care oferă un comportament implicit pe care subclasele îl pot extinde dacă este necesar. În general este important să fie specificate foarte clar care sunt operațiile care pot fi suprascrise și care sunt cele care trebuie specificate neapărat.

Trebuie avute în vedere anumite lucruri care țin de implementare:

- În general, metodele care trebuie apelate pot fi declarate ca **protected**, ceea ce înseamnă că ele pot fi apelate doar de metoda template (și clasele din același pachet în Java).

- Aproape întotdeauna, metoda template este o metodă care nu poate fi suprascrisă în subclase. Ea poate fi o metodă nevirtuală (finală în Java și nevirtuală în C++).
- Cu cât numărul de metode care trebuie suprascrise este mai mic cu atât este mai bine, deoarece cel care va utiliza acea clasă o va utiliza cu mai mare ușurință.

7.4 Exemplu

```
public abstract class List {
    abstract public int count();
    abstract public Object getElem(int i);
    abstract public void add(Object o);

    public boolean contains(Object o) {
        for(int i=0;i<count();i++)
            if(getElem(i).equals(o))
                return true;
        return false;
    }

    public String toString() {
        StringBuilder s=new StringBuilder();
        for(int i=0;i<count();i++)
            s.append(getElem(i)).append(" ");
        return s.toString();
    }
}

public class ArrayList extends List {
    private Object[] data;
    private int count;
    public ArrayList(int dim) {
        data=new Object[dim];
    }

    public int count() {
        return count;
    }

    public Object getElem(int i) {
        return data[i];
    }
}
```

```
}

public void add(Object o) {
    data[count++]=o;
}
}

public class TestTemplate {
    public static void main(String args[]) {
        List list=new ArrayList(20);
        list.add("gigi");
        list.add("marian");
        System.out.println(list.contains("gigi"));
        System.out.println(list.getElem(0));
    }
}
```