# ŞABLOANE DE PROIECTARE

20 noiembrie 2023

# Cuprins

# Capitolul 1

# Prezentare generală

Scopul şabloanelor de design este de a înregistra experienţă referitoare la programarea obiect-orientată. Fiecare astfel de şablon numeşte, explică şi evaluează un design în sisteme obiect orientate.

## 1.1 Ce este un şablon de proiectare?

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

El dă şi prima definiţie a ceea ce ar putea însemna un şablon de proiectare: *"Fiecare şablon descrie o problemă care tot apare în mediul nostru, după care descrie esenţa soluţiei acelei probleme, în aşa fel încât ea să poată fi folosită de mii de ori, fără a mai face acelaşi lucru de mai multe ori."*

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

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

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

conducând la soluţii mai complexe care agregă mai multe şabloane.

Momentul care a condus la popularizarea pe scară largă a noţiunii de şablon de proiectare în domeniul informaticii a fost apariţia în 1995 a cărţii *"Design patterns - Elements of Reusable Object Oriented Software"*, carte scrisă de E. Gamma, R. Helm, Johnsson şi Vlissides.

În această carte cei patru autori nu au meritul de a fi inventat şabloanele 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 şablon în industria software
- au catalogat şi descris fiecare şablon în parte
- au prezentat într-un mod logic, strategiile care stau la baza acestor şabloane de proiectare.

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

În semn de apreciere, autorii acesteia sunt cunoscuţi sub numele de "Gang of Four" (GoF).

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

O altă definiţie a unui şablon de design, care este dată chiar de către cei patru sună astfel: *"Şabloanele de proiectare sunt descrieri ale unor obiecte şi clase care comunică şi care sunt particularizate pentru a rezolva o problemă generală de design într-un context particular"*.

În continuare prezentăm o ultimă definiţie care îi aparţine tot lui Christopher Alexander. Chiar dacă ea a fost gândită referitor la arhitectură ea este foarte potrivită în domeniul informaticii: *"Fiecare şablon de proiectare este o regulă în trei părţi, care exprimă o relaţie între un context, o problemă şi o soluţie"*.

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

- *numele şablonului* - este important deoarece el ne ajută să comunicăm mai uşor cu cei cu care lucrăm
- *problema* - descrierea situaţiei în care se aplică
- *soluţia* - modalitatea de a rezolva acea problemă
- *consecinţele* - avantajele şi dezavantajele acelei abordări

## 1.2 De ce sunt utile?

Până acum am văzut ce sunt şabloanele de proiectare. O altă întrebare legitimă în acest moment ar fi, de ce sunt ele utile. Oare este neapărat necesar să înţelegem conceptul de şablon de proiectare? Este neapărat necesar să cunoaştem exemple de şabloane de proiectare?

Răspunsul autorului este că da, din mai multe motive:

- Utilizarea şabloanelor de proiectare conduce la reutilizarea unor soluţii care şi-au arătat de-a lungul timpului eficienţa. După cum se ştie una din cele mai importante probleme în domeniul informaticii ţine de reutilizabilitate. Dacă până acum am tot vorbit despre reutilizarea codului, oare nu se poate ca reutilizarea ideilor să fie la fel de importantă?
- Permit 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 şi designului sistemelor obiect orientate.
- Au ca principal obiectiv crearea de cod flexibil, uşor de modificat. Scopul lor este ca, în măsura în care este posibil, să permită adăugarea de noi funcţionalităţi fără a modifica cod existent.
- Odată înţelese bine, ele sunt nişte exemple foarte bune relativ la principiile de bază ale programării obiect orientate.
- Permit însuşirea unor strategii îmbunătăţite care ne pot ajuta şi atunci când nu lucrăm cu şabloanele de proiectare:

- Lucrul pe interfeţe nu pe implementări
- Favorizarea compoziţiei în dauna moştenirii
- Găsirea elementelor care variază şi încapsularea lor

## 1.3 Clasificarea şabloanelor

În funcţie de nivelul la care apar, şabloanele sunt de mai multe feluri:

- *idioms* - sunt primele care au apărut şi sunt dependente de anumite tehnologii (de exemplu, lucrul cu smart pointers în limbajul C++).
- *design patterns* - reprezintă soluţii 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 şabloane la nivel de sistem, adică sunt şabloane care sunt folosite pentru a descrie la nivel înalt arhitectura unui întreg sistem.

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

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

- creaţionale - abstractizeaza procesul de creare a obiectelor pentru a creşte 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 comparaţie între el şi şabloanele de proiectare.

# Partea I

# Şabloane creaţionale

# Capitolul 2

# Singleton

## Scop

*Asigură faptul că o clasă are o singură instanţă şi oferă un singur punct de acces global la ea.*

## Motivare

În anumite situaţii vrem ca o clasă să aibă o singură instanţă care să poată fi accesată de oriunde dintr-o aplicaţie.

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

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

## Aplicabilitate

Aplicăm acest şablon de proiectare atunci când:

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

- Când singura instanţă trebuie să poată fi extensibilă prin subclasare şi clienţii ar trebui să fie capabili să folosească o instanţă extinsă fără a-şi modifica codul.

## Structură

Structura şablonului de proiectare Singleton este prezentată în figura 2.1.

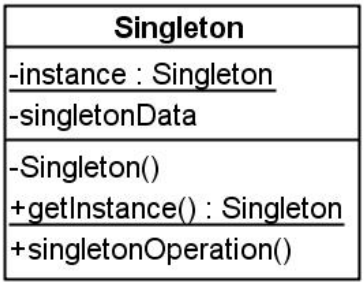


Figura 2.1: Structura şablonului Singleton

După cum se poate observa, pentru a rezolva această problemă trebuie să facem întâi şi 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 reţinut într-o instanţă 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 creea altul. Dacă el nu există, atunci este creat şi apoi returnat.

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

Folosirea acestui şablon are mai multe consecinţe:

- Prin faptul că nu se foloseşte o variabilă globală, se asigură nepoluarea spaţiului de nume cu o nouă denumire.
- Permite subclasarea unei clase de tip Singleton, astfel încât noua clasă să poată avea ea însăşi o singură existenţă.
- Cu anumite modificări se poate obţine 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 şablon de proiectare. Am construit în continuare o clasă `Printer` care simulează o imprimantă. Pentru această clasă trebuie să avem o singură instanţă la un moment dat, lucru care este asigurat de folosirea şablonului 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 şi 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 şi iniţializatorul ei static. Acesta este o construcţie de tipul următor, care poate iniţializa atribute statice şi eventual efectuează anumite operaţii în plus:

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

În cele ce urmează vom prezenta codul pentru o aplicaţie care foloseşte încărcarea dinamică a claselor şi iniţializatorul static.

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

public class TestDinamic {
  public static void main(String[] args) throws InstantiationExcept
      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

*Defineşte o interfaţă pentru crearea unui obiect, dar lasă subclasele să decidă care clasă trebuie instanţiată. Permite unei clase să transfere responsabilitatea instanţierii către subclase.*

## Motivare

În această secţiune vom prezenta ceea ce se întâmplă într-un framework în care dorim să creem mai multe tipuri de documente diferite. Totuşi, 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

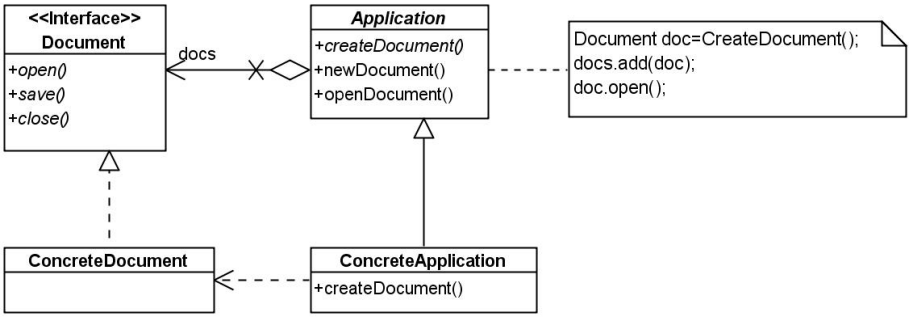


Figura 3.1: Motivare Factory Method

În această diagramă partea comună este reprezentată de interfaţa `Document` (partea comună tuturor documentelor) şi de clasa abstractă `Application` (partea comună tuturor aplicaţiilor).

## Aplicabilitate

Se foloseşte *Factory Method* atunci când:

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

## Structură

Structura acestui şablon 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 şi returna un anumit tip de obiect de tipul `Product`.

Această metodă este rescrisă într-o subclasă pentru a realiza o anumită operaţie.

Trebuie precizat faptul că acest şablon are mai multe consecinţe:

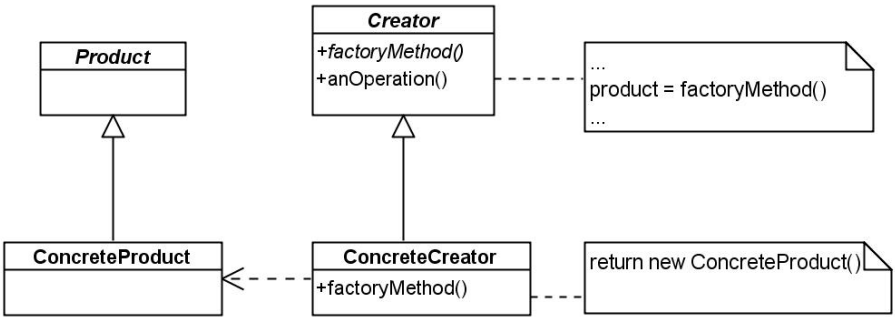


Figura 3.2: Structura şablonului Factory

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

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

## Exemplu

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

În primul rând definim o interfaţă `Shape`:

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

După aceasta definim o clasă `ShapeFactory` care ştie să creeze obiecte de tip `Shape` pe baza unui şir 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` şi `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 aplicaţie 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 interfaţă pentru crearea familiilor de obiecte înrudite sau dependente fără a specifica clasele lor concrete.*

## Motivare

Un astfel de şablon de proiectare se foloseşte 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 şi interfeţe folosite pentru lucrul cu baze de date în Java.

După cum se poate uşor observa, pentru a lucra cu baze de date se folosesc în codul pe care-l scriem mai mult interfeţe decât clase propriuzise. Avem astfel un exemplu care ilustrează lucrul pe interfeţe 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 nişte interfeţe, codul propriu zis care este apelat, pentru a realiza conexiuni cu serverul de baze de date şi pentru a efectua diverse operaţii fiind cuprins in driver-ul pe care îl folosim.

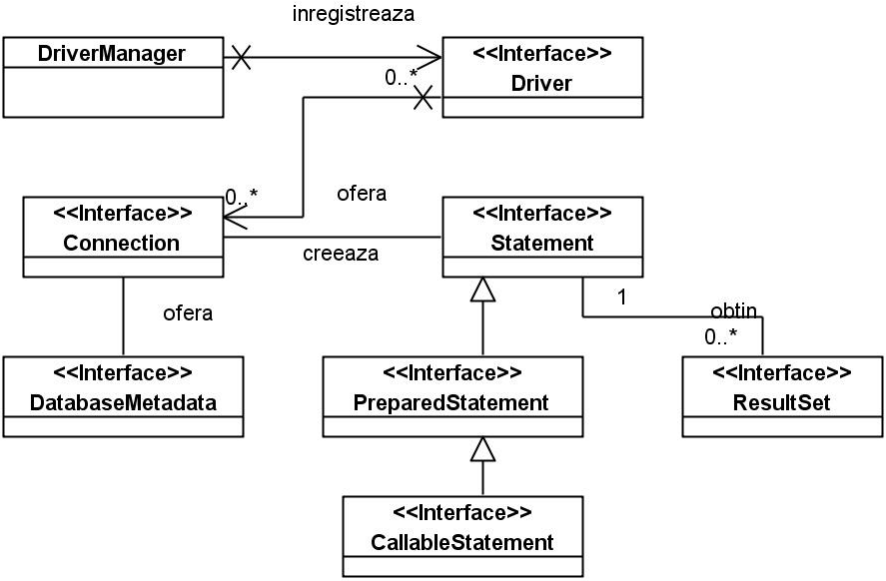


Figura 4.1: Arhitectura JDBC

Reamintim că un driver este un fişier JAR care trebuie inclus la momentul *rulării* aplicaţiei. El este responsabil cu tot ceea ce ţine 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-şi construiască un SGBD, poate face acest lucru şi, mai mult, după ce-şi construieşte 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 secţiunea dedicată exemplelor ce se întâmplă de fapt în acest cod.

## Aplicabilitate

Se foloseşte *Abstract Factory* atunci când:

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

## Structură

Structura acestui şablon este prezentată în figura 4.2

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

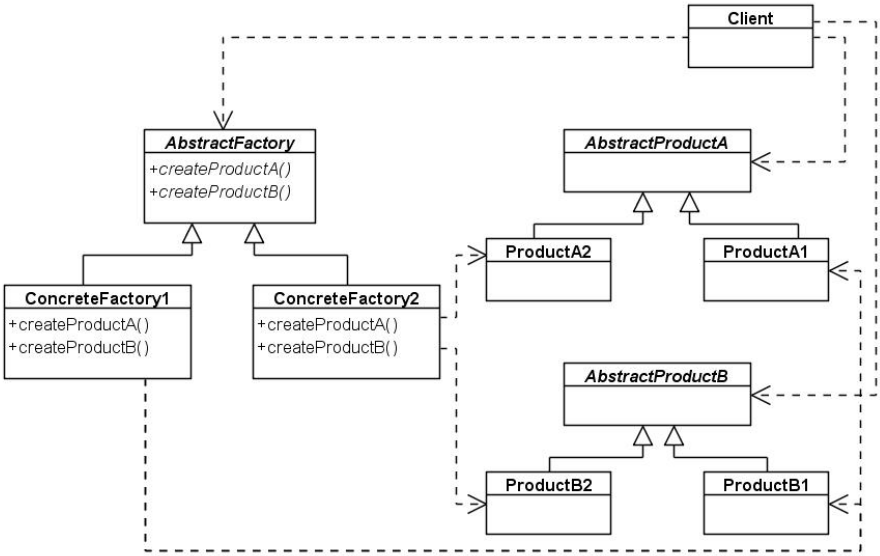


Figura 4.2: Structura şablonului Abstract Factory

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

Folosirea acestui şablon are mai multe consecinţe:

- Prin folosirea fabricii de obiecte se izolează clienţii de produsele efective pe care le folosesc. Acest lucru reprezintă o modalitate de asigurare a încapsulării.

- Folosirea Abstract Factory ajută la schimbarea uşoară a familiilor de obiecte. Pentru a realiza acest lucru trebuie schimbată o singură linie de cod.

- Se promovează consistenţa 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 lanţul de derivare al lui `AbstractFactory`) precum şi introducerea mai multor clase corespunzătoare noului produs.

## Exemplu

În secţiunea referitoare la acest şablon de proiectare vom prezenta modul în care ar trebui folosită o familie de produse precum şi 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 secţiuni 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 iniţializator static care este executat. În acest iniţializator este asociat un obiect de tip `Driver` corespunzător driverului utilizat (`com.mysql.jdbc.Driver` dacă lucrăm cu baze de date MySQL) unui şir de caractere care descrie acel driver (de ex "jcdb:mysql://“).

3. Când se apelează metoda `getConnection()` a clasei `DriverManager`, de fapt se parcurge lista de drivere, se selectează cel corespunzător şirului de caractere transmis şi cu ajutorul acestuia se creează o nouă conexiune, care este returnată (un fel ce `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 înţelege foarte bine corespondenţele între clasele existente în cazul lucrului cu baze de date şi a şablonului de proiectare Abstract Factory, este bine să vedem ce rol are fiecare clasă:

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

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

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

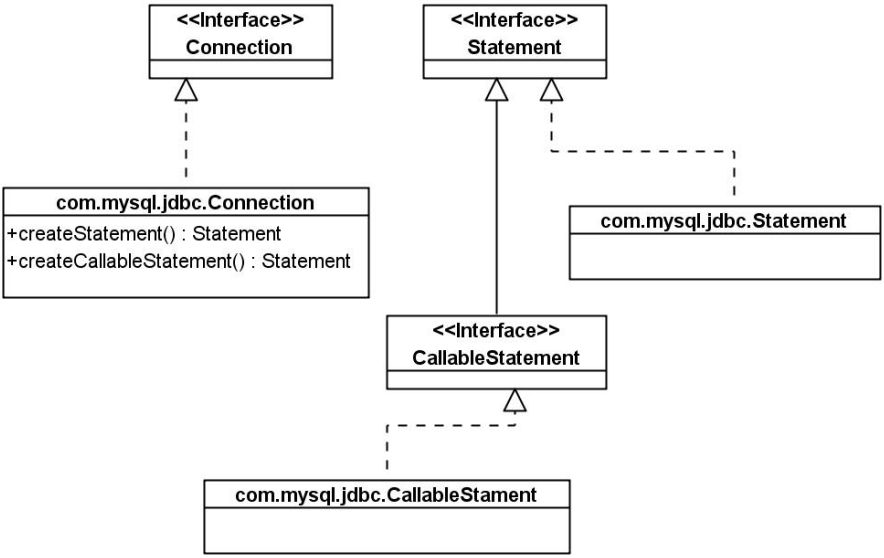


Figura 4.3: Ilustrare Abstract Factory în JDBC

# Partea II

# Şabloane structurale

# Capitolul 5

# Adapter

**Scop:** Converteşte interfaţa unei clase într-o altă interfaţă pe care clientul o aşteaptă. Permite conlucrarea unor clase care altfel n-ar putea lucra împreună din cauza interfeţelor incompatibile.

## Motivare

**Motivare:** Să presupunem că avem un program de desenare care poate desena mai multe figuri geometrice şi texte. Pentru fiecare tip de figură avem câte o clasă.

Evident, un pic mai complicată este clasa pentru desenarea textelor. Să presupunem că deja avem o astfel de clasă, dar interfaţa ei nu corespunde cu cea pe care o dorim, în cazul nostru `Shape`.

După cum se poate vedea în exemplu, soluţia este să definim o nouă clasă, `TextShape` care să adapteze clasa `TextView`.

Acest lucru este prezentat în figura 5.1.

După cum se poate vedea, clasa `TextShape` are un atribut de tipul `TextView`, atunci când este apelată o metodă a clasei, de fapt se apelează metoda (metodele) corespunzătoare din clasa `TextView`.

## Alte soluţii mai bune?

Ce alte posibilităţi mai avem?

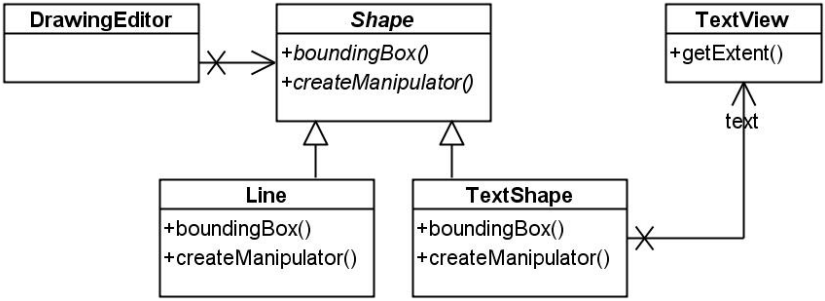


Figura 5.1: Motivare utilitate Adapter

- Am putea să construim o nouă clasă `TextShape` în care să rescriem toată funcţionalitatea, dar acest lucru ar putea fi foarte complex.

## Aplicare

Se foloseşte Adapter atunci când:

- Vrem să folosim o clasă existentă şi interfaţa ei nu se potriveşte cu cea pe care o aşteptăm.
- Vrem să creem o clasă reutilizabilă care cooperează cu clase neînrudite şi pe care nu le putem anticipa (clase care nu au interfeţe compatibile).
- (numai pentru object adapter) Trebuie folosite câteva subclase existente dar este nepractică adaptarea fiecăreia prin subclasare. Un object adapter poate adapta interfaţa clasei părinte.

## Structură

După cum se poate observa avem două soluţii:

- *Class Adapter* - se derivează clasa pe care vrem să o adaptăm şi se implementează interfaţa la care vrem să adaptam.
- *Object Adapter* - în clasa care realizează adaptarea reţinem o referinţă la obiectul adaptat. Clasa care face adaptarea trebuie să implementeze interfaţa la care se face adaptarea.

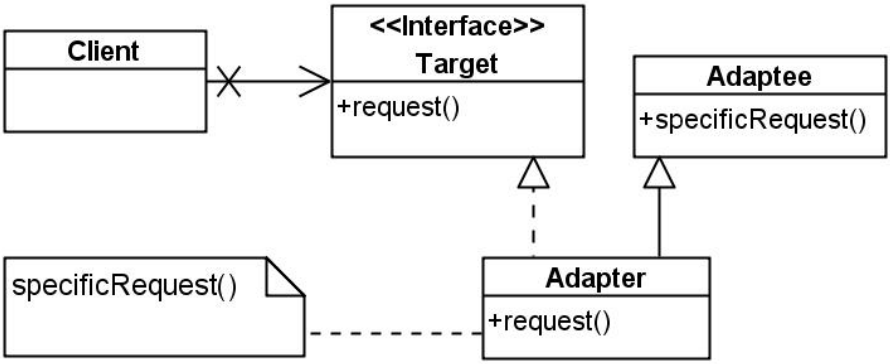


Figura 5.2: Structura Class Adapter

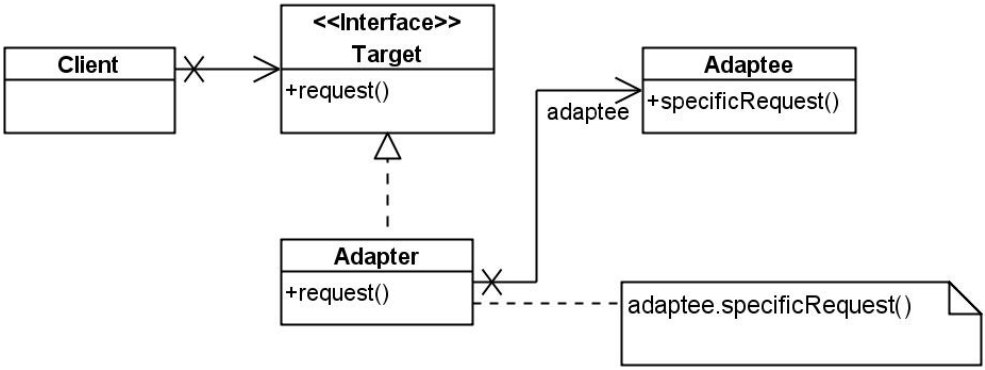


Figura 5.3: Structură Object Adapter

Observaţii:

- Dacă vrem să adaptăm mai multe clase care formează o ierarhie de clase, trebuie să folosim neapărat *object adapter* deoarece *class adapter* nu poate fi aplicat.

- Exista noţiunea de *pluggable adapters*. Aceste adaptoare sunt nişte clase care sunt deja concepute în interiorul unei clase pentru a putea introduce în această clasă funcţionalitate din alte clase, care pot fi foarte diferite.

  Ideea care stă în spatele acestor tipuri de adaptoare este aceea că o

clasă este cu atât mai generică cu cât se fac mai puţine presupuneri asupra obiectelor cu care clasa lucrează. Acesta este motivul pentru care, pentru anumite clase este util să se construiască o anumită adaptare a interfeţelor.

## Exemplu

Să presupunem că avem o clasă `Stiva` şi avem un program în care vrem să folosim obiecte care implementează interfaţa `Stack`. Putem proceda în modurile următoare:

```
public interface Stack {
  void push(int x);
  int top();
  int pop();
  boolean isEmpty();
}

public class Stiva {
  private int[] a;
  private int count;
  public Stiva(int dim) {
    a = new int[10];
  }
  public Stiva() {
    this(10);
  }
  public void adauga(int x) {
    a[count++] = x;
  }
  public int varf() {
    return a[count - 1];
  }
  public int scoate() {
    return a[--count];
  }
  public boolean esteGoala() { return count == 0; }
}

public class StackObjectAdapter implements Stack {
  private Stiva s;

  public StackObjectAdapter(int dim) {
```

```
    s = new Stiva(dim);
  }
  public StackObjectAdapter() { s = new Stiva(); }
  public void push(int x) {
    s.adauga(x);
  }
  public int top() {
    return s.varf();
  }
  public int pop() {
    return s.scoate();
  }
  public boolean isEmpty() {
    return s.esteGoala();
  }
}

public class StackClassAdapter extends Stiva implements Stack {
  public StackClassAdapter(int dim) {
    super(dim);
  }
  public StackClassAdapter() {
    super();
  }
  public boolean isEmpty() {
    return super.esteGoala();
  }
  public int pop() {
    return super.scoate();
  }
  public void push(int x) {
    super.adauga(x);
  }
  public int top() {
    return super.varf();
  }
}

public class TestStack {
  public static void main(String[] args) {
    // Stack s=new StackObjectAdapter();
    Stack s = new StackClassAdapter();
    s.push(3);
    s.push(4);
    System.out.println(s.pop());
    System.out.println(s.top());
```

```
    }
}
```

# Partea III

# Şabloane comportamentale

# Capitolul 6

# Command

## 6.1 Scop şi motivare

**Scop:** Încapsulează o cerere ca un obiect, permiţând parametrizarea clienţilor cu diferite cereri, punerea în coadă sau logging-ul cererilor, precum şi suportarea operaţiilor undoable.

**Motivare:** De multe ori, într-o aplicaţie avem un meniu prin intermediul căruia dorim să putem efectua diverse operaţii, fără ca meniul însuşi să ştie cine este cel care primeşte comanda sau care este comanda executată.

În aceste situaţii comanda este încapsulată într-un obiect, care are o metodă `execute()`, în interiorul căreia putem să definim ceea ce vrem să facem. Din punctul de vedere al componentei meniu este doar un obiect care are o metoda `execute()`. Acest lucru asigură faptul că am creat un meniu cât se poate de general.

Această situaţie este prezentată în figura 6.1.

După cum se poate vedea fiecare menu item este configurat cu o instanţă a interfeţei `Command`.

Când un utilizator execută click pe un meniu, automat se apelează metoda `execute()` a obiectului `Command` corespunzător, fără ca meniul să poată fi conştient de obiectul care a generat acţiunea sau de obiectul însărcinat cu efectuarea ei.

Se poate chiar combina şablonul *Command* cu şablonul *Composite* pentru a se crea macrocomenzi. Astfel, după cum ne aducem aminte, o macrocomandă ar putea fi alcătuită din mai multe macrocomenzi şi mai

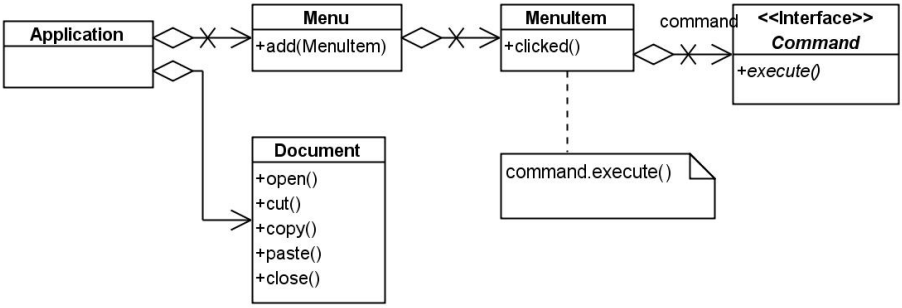


Figura 6.1: Motivare utilitate Command

multe comenzi simple.

## 6.2 Aplicare

Se foloseşte şablonul Command atunci când vrem să:

- Parametrizăm obiecte prin acţiuni pe care ele trebuie să le poată face.
- Să specificăm, punem în coadă şi să executăm cereri la momente diferite. Un obiect Command poate avea o viaţă independentă de cererea iniţială. Dacă primitorul cererii poate fi reprezentat într-un spaţiu de adresă independent, atunci se poate ca el să primească cererea într-un proces diferit şi să o rezolve acolo.
- Putem suporta operaţia *undo*. Operaţia `execute()` a lui `Command` poate stoca stare pentru a putea reface operaţia. Interfaţa `Command` ar trebui să aibă o metodă `unexecute()` care să aibă efectul invers lui `execute()`. Comenzile executate ar trebui stocate într-un *history list*.
- Suportăm jurnalizarea care ar permite ca schimbările să fie reaplicate în cazul unui sistem crash. Ar putea necesita îmbogăţirea interfeţei `Command` şi cu operaţii de salvare/ încărcare.
- Structurăm un sistem asupra operaţiilor la nivel înalt construite pe baza operaţiilor primitive. Această structură este comună în siste-

mele care dorim să poată permite utilizarea tranzacţiilor. `Command`-urile au toate aceeaşi interfaţă ceea ce ar permite modelarea la fel a tuturor tranzacţiilor.

## 6.3 Structură

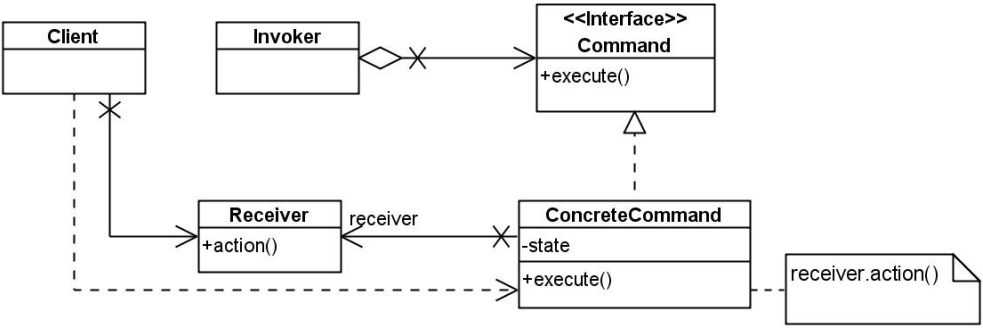


Figura 6.2: Structură şablon Command

Ca şi consecinţe ale acestui şablon, remarcăm:

- Decuplează obiectul care face o cerere de obiectul care ştie cum să trateze acea cerere.
- Este foarte uşor să se adauge noi comenzi, nefiind nevoie de schimbarea nici unei clase existente.
- Există creat cadrul pentru folosirea macrocomenzilor.
- Oferă cadrul pentru implementarea operaţiilor de *undo*.

  Pentru a putea face acest lucru, de multe ori este nevoie ca obiectele de tip `Command` să poată reţine informaţii de stare suplimentare.

  Mai mult, este nevoie să reţinem o listă de obiecte de tip `Command` cu ajutorul căreia să putem reface anumite operaţii. Acest lucru ar putea fi posibil dacă, în plus, obiectele de tip `Command` ar fi înregistrate şi cu o operatie `unexecute()`.

## 6.4 Exemplu

În final prezentăm un mic exemplu de utilizare al acestui şablon. Avem o aplicaţie care trebuie să tot execute mai multe task-uri, fiecare task având o anumită frecvenţă de execuţie.

Componenta care se ocupă de executarea task-urilor nu trebuie să cunoască tipul componentelor de tip `Task` pe care le reţine. Mai mult, aceasta componentă nu ştie nimic despre operaţiile care vor fi efectuate.

Prezentăm diagrama acestei aplicaţii in figura 6.3.

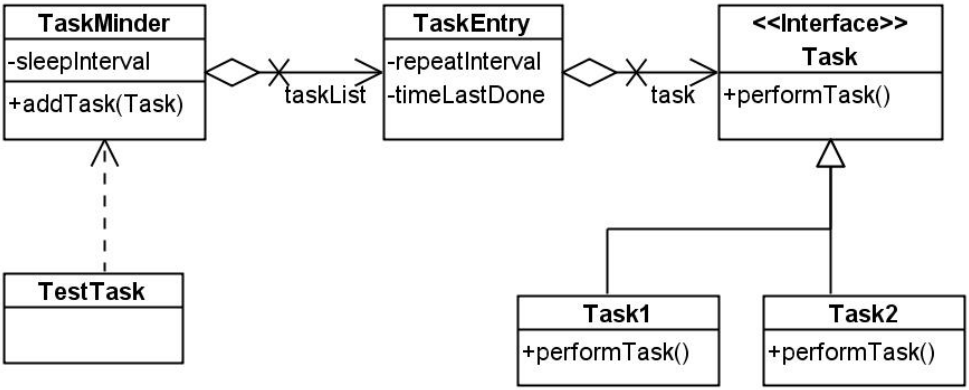


Figura 6.3: Diagrama aplicaţiei exemplu

Avem următorul cod:

```
public interface Task {
  public void performTask();
}

public class Task1 implements Task{
  public void performTask() {
    System.out.println("First task...");
  }
}

public class Task2 implements Task{
  public void performTask() {
    System.out.println("Second task...");
  }
}
```

```
public class TaskEntry {
  private Task task;
  private long repeatInterval; // How often task should be
  private long timeLastDone; // Time task was last done

  public TaskEntry(Task task, long repeatInterval) {
    this.task = task;
    this.repeatInterval = repeatInterval;
    this.timeLastDone = System.currentTimeMillis();
  }

  public Task getTask() {
    return task;
  }

  public void setTask(Task task) {
    this.task = task;
  }

  public long getRepeatInterval() {
    return repeatInterval;
  }

  public void setRepeatInterval(long ri) {
    this.repeatInterval = ri;
  }

  public long getTimeLastDone() {
    return timeLastDone;
  }

  public void setTimeLastDone(long t) {
    this.timeLastDone = t;
  }

  public String toString() {
    return (task + " to be done every " + repeatInterval
        + " ms; last done " + timeLastDone);
  }
}

import java.util.*;
public class TaskMinder extends Thread {
  private long sleepInterval;
  // How often the TaskMinder should
```

```
  // check for tasks to be run
  private ArrayList<TaskEntry> taskList; // The list of tasks

  public TaskMinder(long sleepInterval, int maxTasks) {
    this.sleepInterval = sleepInterval;
    taskList = new ArrayList<TaskEntry>(maxTasks);
    start();
  }

  public void addTask(Task task, long repeatInterval) {
    long ri = (repeatInterval > 0) ? repeatInterval : 0;
    TaskEntry te = new TaskEntry(task, ri);
    taskList.add(te);
  }

  public Iterator<TaskEntry> getTasks() {
    return taskList.iterator();
  }

  public long getSleepInterval() {
    return sleepInterval;
  }

  public void setSleepInterval(long si) {
    this.sleepInterval = si;
  }

  public void run() {
    while (true) {
      try {
        sleep(sleepInterval);
        long now = System.currentTimeMillis();
        Iterator<TaskEntry> e = taskList.iterator();
        for(TaskEntry te:taskList) {
          if (te.getRepeatInterval() + te.getTimeLastDone() < now) {
            te.getTask().performTask();
            te.setTimeLastDone(now);
          }
        }
      } catch (Exception e) {
        System.out.println("Interrupted sleep: " + e);
      }
    }
  }
}
```

```
public class TestTask {
  public static void main(String args[]) {
    TaskMinder tm = new TaskMinder(500, 100);
    Task1 t1=new Task1();
    tm.addTask(t1, 800);
    Task t2=new Task2();
    tm.addTask(t2, 3000);
  }
}
```

# Capitolul 7

# Strategy

## 7.1 Scop şi motivare

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

**Motivare:** Există numeroşi algoritmi pentru spargerea unui stream în linii. Introducerea algoritmului în clasa care-l foloseşte nu este cea mai bună soluţie din mai multe motive:

- Clienţii vor deveni cu atât mai complecşi cu cât numărul algoritmilor suportaţi va fi mai mare.
- Nu toţi algoritmi vor fi folosiţi de fiecare dată şi nu vrem să suportăm algoritmi care s-ar putea să nu fie folosiţi.
- Este dificil de adăugat noi algoritmi şi de a-i varia pe cei existenţi 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 funcţionalităţi).
- Un sistem trebuie să fie *inchis la modificări* (adăugarea noilor funcţionalităţi nu trebuie să necesite schimbarea codului existent).

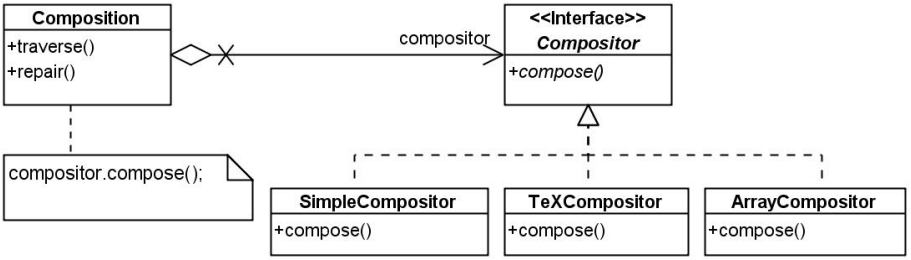


Figura 7.1: Motivare utilitate Strategy

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

Clasa care foloseşte algoritmul va reţine o referinţă la obiectul care va conţine algoritmul propriu-zis. Avantajul acestei abordări este că algoritmul poate varia la momentul rulării.

## 7.2 Aplicare

Se foloseşte şablonul 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 diferiţi din punctul de vedere al raportului spaţiu/timp. Strategiile pot fi folosite când aceşti algoritmi sunt implementaţi în ierarhii de clase.

- Un algoritm foloseşte date de care clienţii n-ar trebui să ştie. Se evită expunerea de structuri complexe specifice algoritmului.

- O clasă defineşte mai multe comportamente şi acestea apar apelate în urma mai multor instrucţiuni condiţionale complexe în interiorul operaţiilor clasei. Fiecare ramură a condiţiilor este mutată într-o clasă separată.

## 7.3 Structură

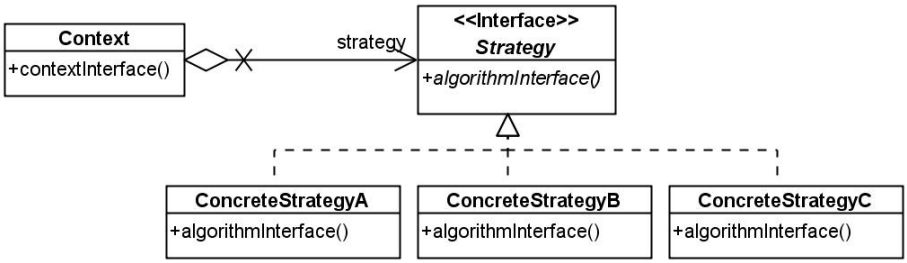


Figura 7.2: Structură şablon Strategy

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 referinţă la obiectul `Context` şi obiectul strategie să apeleze metode ale clasei `Context` pentru a obţine datele necesare.

Ca şi modalitate de lucru, în general, se parcurg următorii paşi:

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

Folosirea acestui şablon are mai multe consecinţe (avantaje şi 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 buna la subclasare. S-ar fi putut să derivăm direct clasa `Context` dar acest lucru nu ar fi fost aşa de bun din mai multe motive:

  - Ar fi făcut mai greu de înţeles clasa `Context`.
  - Nu ar fi permis varierea dinamică a strategiei utilizate.

- Folosirea strategiilor elimină instrucţiunile condiţionale complexe.
- Permite alegerea foarte uşoară a implementării dorite.
- Clienţii sunt conştienţi de diferitele strategii, ei fiind responsabili cu actualizarea şi stabilirea strategiei active.
- Uneori e greu de găsit o interfaţă comună între mai mulţi algoritmi, în funcţie de complexitatea fiecăruia. În aceste situaţii, uneori este necesar să realizăm o legătură mult mai strânsă între context şi strategie pentru a nu fi necesar să transmitem parametri inutil.
- Strategiile pot creşte foarte mult numărul de obiecte dintr-o aplicaţie, motiv pentru care uneori este mai bine ca ele să fie implementate sub forma unor obiecte fără stare care să fie partajate de toate obiectele din aplicaţie.

## 7.4 Exemplul 1

Problemă: Să se găsească o metodă prin care să putem sorta elementele unui şir cu obiecte de orice tip.

```
public interface Comparator <T> {
  int compare(T o1, T o2);
}

public class Arrays {
  public static <T> void sort(T[] a, Comparator<? super T> c)
  {
    for(int i=0;i<a.length-1;i++)
      for(int j=i+1;j<a.length;j++)
        if(c.compare(a[i], a[j])>0)
        {
          T aux=a[i];
          a[i]=a[j];
          a[j]=aux;
        }
  }
}
```

```
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]+" ");
  }
}
```

## 7.5 Exemplul 2

Problemă: Să se implementeze acelaşi lucru folosind facilităţile oferite de mediul Java şi 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 7.3.

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 7.3: Utilizare Strategy in C#

## 7.6 Exemplul al 3-lea

Problemă: Să se implementeze un joc de X&0 folosind şablonul Strategy pentru stabilirea dificultăţii 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 8

# Observer

## 8.1 Scop şi motivare

**Scop:** Defineşte o dependenţă de la unul la mai mulţi între obiecte astfel încât atunci când un obiect îşi schimbă starea, toate cele dependente de el să fie notificate şi actualizate automat.

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

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 aceloraşi date. În momentul în care se modifică anumite date, automat toate reprezentările trebuie actualizate. Un exemplu este prezentat în figura 8.1.

În această figură există un subiect, care reţine date, precum şi mai mulţi observatori care îşi schimbă reprezentările dacă datele din subiect se modifică.

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

Ca observaţie suplimentară, trebuie remarcat că observatorii pot fi de tipuri foarte variate, lucru care face şi mai util acest şablon.

Practic, într-o aplicaţie care foloseşte acest şablon se întâmplă următoarele operaţii:

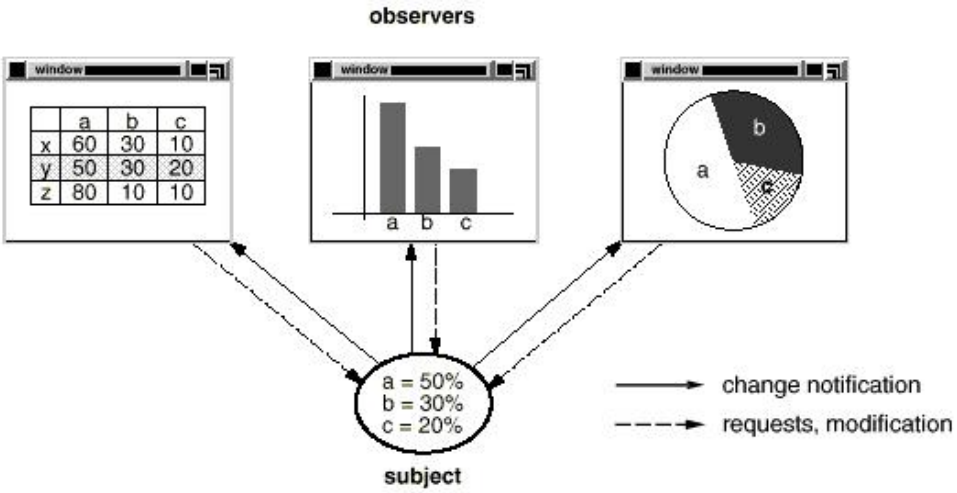


Figura 8.1: Motivare utilitate Observer

- 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 anunţate.
- În plus, la orice moment, se poate să se înregistreze noi obiecte care din acel moment vor fi anunţate de modificări ale subiectului, sau să se dezînregistreze obiecte care nu vor mai fi anunţate din acel moment.

Evident, pentru a putea să-şi actualizeze starea, fiecare observator reţine o referinţă la obiectul model.

## 8.2 Aplicare

Se foloseşte acest şablon în una din următoarele situaţii:

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

- Când o schimbare într-un obiect necesită schimbarea altora şi nu se ştie 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.

## 8.3 Structură

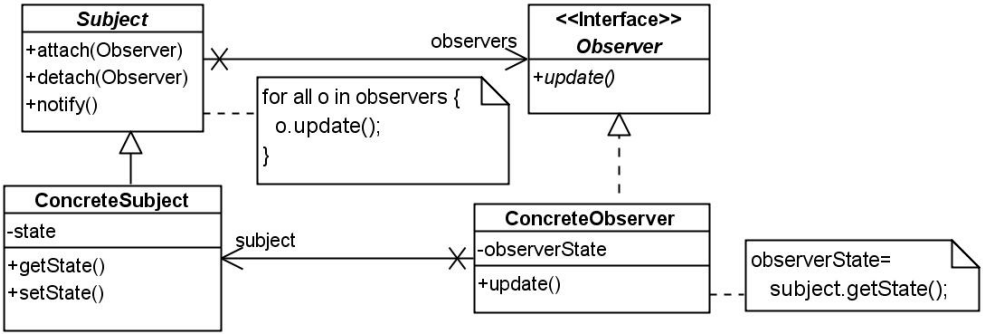


Figura 8.2: Structură şablon Observer

Folosirea acestui şablon are mai multe consecinţe:

- Cuplarea între subiect şi observator este una abstractă. Un subiect nu ştie nici tipul concret al observatorilor, nici numărul acestora. Un astfel de design, poate facilita comunicarea între layere diferite fără a strica designul aplicaţiei.

- Avem o comunicare de tip broadcast. Subiectul nu transmite mesaje personalizate fiecărui obiect, ci transmite acelaşi mesaj tuturor, doar obiectele interesate tratând acel mesaj.

- Trebuie lucrat cu grija, deoarece în urma modificărilor asupra subiectului, se pot produce mai multe actualizări ale observatorilor decât ne-am aştepta. Uneori, are sens să transmitem un mesaj care să şi descrie modificarea care s-a produs, pentru a fi mai uşor de tratat de către observatori.

- Uneori, dacă putem avea mai multe obiecte observate de către un observator, este necesar să putem să ştim şi cine este obiectul care a trimis notificarea.

- O problemă interesantă este cine ar trebui să stabilească când se fac notificările? Exista două soluţii:

  - Se poate ca oricând se modifică ceva în starea subiectului, toţi observatorii să fie anunţaţi. Această modalitate prezintă un avantaj şi un dezavantaj.

    Avantajul ar fi că nu mai trebuie ca utilizatorul să-şi 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 subiecţi:

  - 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 informaţie despre schimbarea care s-a produs (modelul **pull**).

- Se poate ca observatorii să se înregistreze la un subiect şi din punctul de vedere al aspectului de care sunt interesaţi. Astfel se elimină numeroase apeluri inutile.

## 8.4 Exemple

### 8.4.1 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 soluţie 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 faţă de modelele anterioare, dar necesită, mai mult cod decât modul de lucru ca în prima situaţie prezentată.

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

Informaţia 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 situaţie, el trebuind să implementeze o interfaţă cum ar fi, de exemplu `ActionListener`.

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

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

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

### 8.4.2 Exemplul 2

În cele ce urmează prezentăm un mic exemplu concret de implementare a şablonului *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 Observator implements Observer{
  private String name;
  private Lista subject;

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

### 8.4.3 Exemplul 3

În cadrul platformei JDK există suport pentru implementarea acestui şablon. Modul de lucru cu clasa `Observable` precum şi cu interfaţa `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();
```

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

# Capitolul 9

# Template

## 9.1 Scop şi motivare

**Scop:** Defineşte scheletul unui algoritm într-o operaţie, lasând anumiţi paşi în sarcina subclaselor. Template Method lasă subclasele să redefinească anumiţi paşi ai unui algoritm fără a schimba structura algoritmului.

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

Când avem o aplicaţie particulară subclasăm fiecare din aceste subclase, pentru a rezolva problema respectivă. Am putea avea ceva corespunzător figurii 9.1.

Conţinutul 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();
  }
```

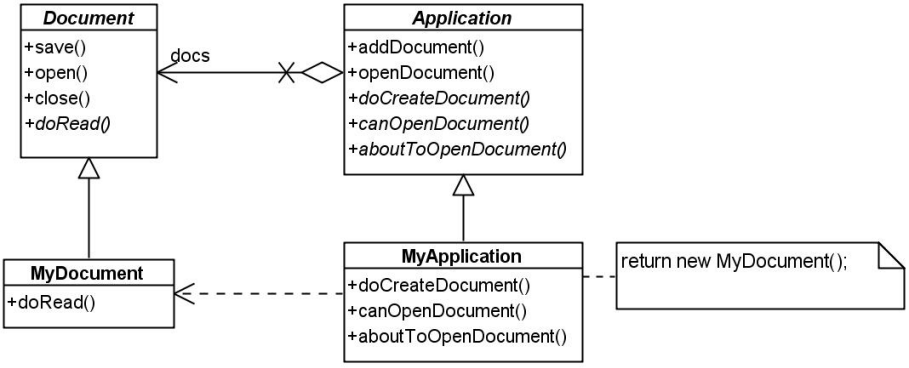


Figura 9.1: Motivare utilitate Template

}

În acest exemplu, `openDocument()` poartă numele de **metodă template**. Această metodă defineşte un algoritm şi ordinea în care se produc diversele operaţii, permiţând fiecărei subclase definirea anumitor operaţii.

## 9.2 Aplicare

Acest şablon de proiectare ar trebui utilizat:

- Pentru a implementa părţile invariante ale unui algoritm şi a lăsa subclasele să implementeze părţile care variază.
- Atunci când comportamentul comun între subclase ar trebui factorizat şi 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 operaţiile "hook" în anumite puncte specifice, permiţând extensia numai prin aceste puncte.

## 9.3 Structură

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

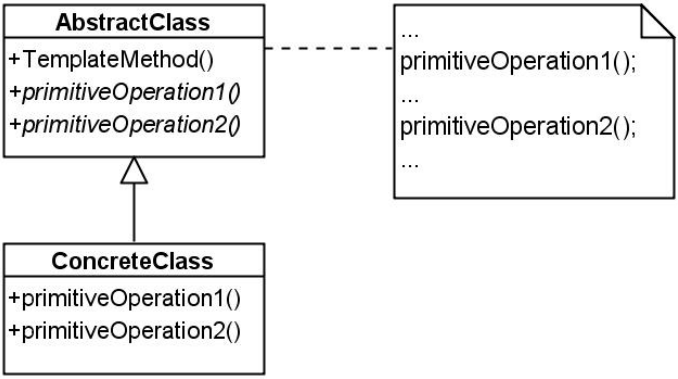


Figura 9.2: Structură şablon Template

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

Şablonul reprezintă o tehnică foarte utilă pentru reutilizarea codului şi 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:

- operaţii concrete (posibil din subclase)
- operaţii 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 operaţiile care pot fi suprascrise şi care sunt cele care trebuie specificate neapărat.

Trebuie avute în vedere anumite lucruri care ţin 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 (şi clasele din acelaşi 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 şi 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 uşurinţă.

## 9.4 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));
  }
}
```