ŞABLOANE DE PROIECTARE

16 octombrie 2023

Cuprins

1 Prezentare generală **1**
1.1 Ce este un şablon de proiectare? 1
1.2 De ce sunt utile? 3
1.3 Clasificarea şabloanelor 4

I Şabloane creaţionale **5**

2 Singleton **7**

II Şabloane structurale **11**

III Şabloane comportamentale **13**

3 Observer **15**
3.1 Scop şi motivare 15
3.2 Aplicare 16
3.3 Structură 17
3.4 Exemple 18
3.4.1 Exemplul 1 18
3.4.2 Exemplul 2 19
3.4.3 Exemplul 3 21

4 Template **25**
4.1 Scop şi motivare 25
4.2 Aplicare 26

4.3 Structură 26
4.4 Exemplu 28

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.

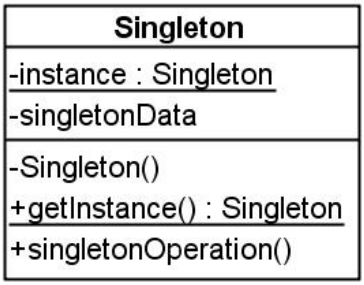

Figura 2.1: Structura şablonului Singleton

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();
  }
}
```

Partea II

Şabloane structurale

Partea III

Şabloane comportamentale

Capitolul 3

Observer

3.1 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 3.1.

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:

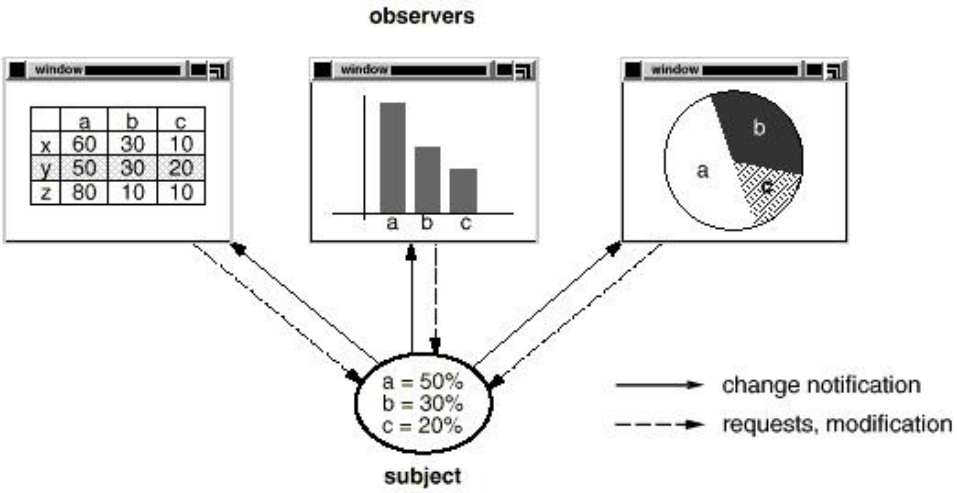

Figura 3.1: Motivare utilitate Observer

- 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.

3.2 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.

3.3 Structură

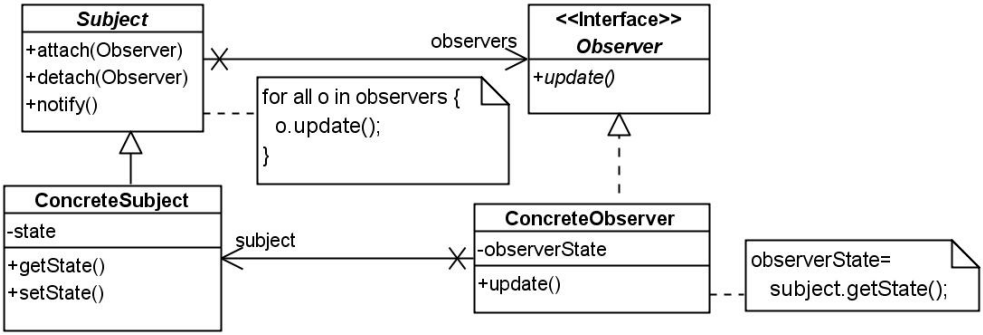

Figura 3.2: Structură şablon Observer

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.

3.4 Exemple

3.4.1 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.

3.4.2 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");
  }
}
```

3.4.3 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 4

Template

4.1 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 4.1.

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();
  }
```

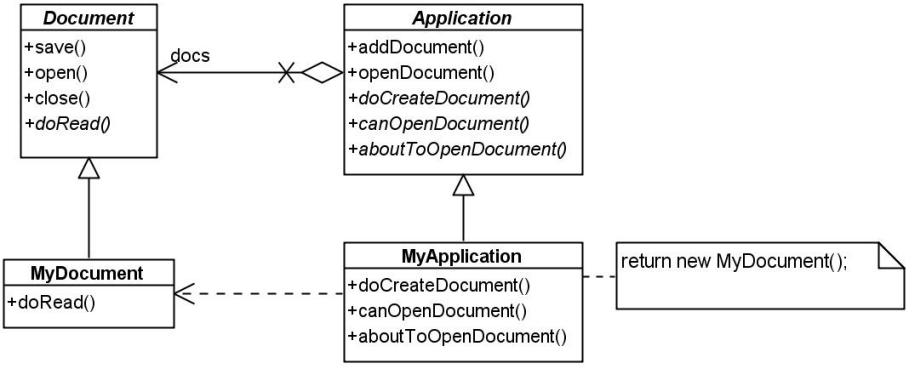

Figura 4.1: Motivare utilitate Template

}

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.

4.2 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.

4.3 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-

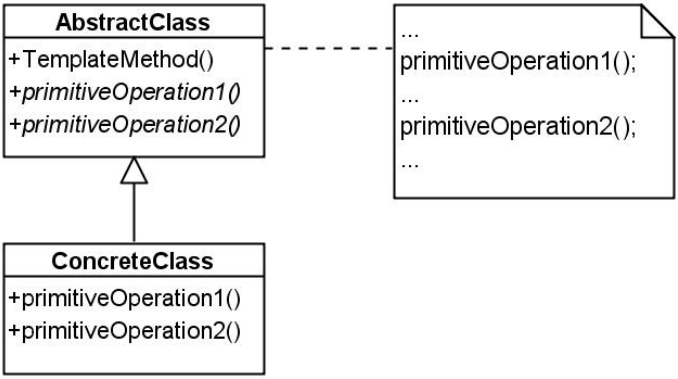

Figura 4.2: Structură şablon Template

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.

4.4 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));
  }
}
```