

Design Patterns

Singleton, Factory, Command, Observer, Visitor

Programare Orientată pe Obiecte



Proiectarea aplicațiilor

- Realizarea unui program presupune *etapa inițială de proiectare*
- Este posibil ca după implementarea proiectului inițial să se revină la această etapă
 - Îmbunătățire – refactorizare
 - Modificare esențială
- Faza de proiectare
 - Se stabilesc clasele, interfețele necesare și relațiile dintre ele
 - Clase noi specifice aplicației
 - Clase predefinite de uz general
 - Pentru un proiect mai complex: crearea de modele ce reprezintă relațiile dintre clase și obiecte, rolul fiecărui obiect etc. - UML (Unified Modeling Language)

Proiectarea aplicațiilor

- Proiectare ce are în vedere posibilitatea de a fi extins și adaptat ulterior într-un mod cât mai simplu și cât mai sigur – *design for change*
- Extindere simplă și sigură
 - Nu se va recurge la modificarea unor clase existente
 - Se vor defini clase noi care
 - Înlocuiesc clasele din programul existent
 - Se vor adăuga claselor existente
- Posibilitatea reutilizării unor clase din aplicație și în alte aplicații

Proiectarea aplicațiilor

- Clasele si obiectele necesare într-o aplicatie pot rezulta din:
 - Analiza aplicației concrete și modelarea obiectelor și acțiunilor din aplicație;
 - Analiza altor aplicații și a bibliotecilor standard Java =>
 - Clase cu caracter mai abstract pentru:
 - Grupare de obiecte
 - Intermediere între obiecte
 - Alt rol care nu este evident din descrierea aplicației
 - Cunoașterea unor soluții de proiectare deja folosite cu succes în alte aplicații – *design patterns*

Proiectarea aplicațiilor

- Mărirea flexibilității în extinderea și adaptarea unei aplicații:
 - prin mărirea numărului de clase și obiecte din aplicație!
- Un proiect ce conține numai clasele rezultate din analiza aplicației poate fi mai compact
 - dar nu este scalabil și adaptabil la alte cerințe apărute după realizarea prototipului aplicației!
- Sunt importante:
 - separarea părților susceptibile de schimbare de părțile care nu se mai modifică
 - evitarea cuplajelor "strânse" dintre clase

Design Patterns

- Recunoașterea și inventarierea unor scheme (șabloane) de proiectare “Design Patterns”:
 - grupuri de clase și obiecte care cooperează pentru realizarea unor funcții.
- În cadrul acestor scheme există clase care au un anumit rol în raport cu alte clase și care au primit un nume ce descrie acest rol;
- Exemple: clase iterator, clase observator, clase “fabrică” de obiecte, clase adaptor ș.a.
- Dincolo de detaliile de implementare se pot identifica clase cu aceleași responsabilități în diferite aplicații.

Design Patterns. Definiții

- Definitii posibile pentru schemele de proiectare folosite în aplicații cu clase:
- Soluții optime generale și reutilizabile ale unei probleme comune în design-ul software.
- Descrieri ale soluțiilor sau template-uri ce pot fi aplicate pentru rezolvarea problemei.
- Reguli pentru realizarea anumitor sarcini în proiectarea programelor cu obiecte.
- Abstractizări la un nivel superior claselor, obiectelor sau componentelor.
- Scheme de comunicare (de interacțiune) între obiecte.

Avantaje și recomandări

- Argumentul principal în favoarea studierii și aplicării schemelor de clase:
 - aplicarea acestor scheme conduce la programe mai ușor de modificat.
 - în general prin clase (obiecte) “slab” cuplate, care știu cât mai puțin unele despre altele.
- Principalele recomandări care rezultă din analiza schemelor de proiectare și a aplicațiilor:
 - Proiectarea cu interfețe și clase abstracte este preferată față de proiectarea cu clase concrete
 - permite separarea utilizării de implementare.
 - Este recomandată crearea de clase și obiecte suplimentare, cu rol de intermediari
 - pentru decuplarea unor clase cu roluri diferite.

Clasificare

Se disting trei categorii:

1. Creational patterns: scheme “creaționale”

- generează obiectele necesare
- mecanisme de creare a obiectelor
- Ex. *Singleton*, *Factory* etc

2. Structural patterns: scheme structurale

- definesc relații între entități
- grupează mai multe obiecte în structuri mai mari.
- *Decorator*, *Adapter*, *Composite*, *Proxy* etc.

3. Behavioral patterns: scheme de interacțiune

- definesc comunicarea între entități
- *Iterator*, *Visitor*, *Observer*, *Command*, *Mediator* etc.

Creational Patterns



- mecanisme de creare a obiectelor
 - **Singleton**
 - **Factory**

Singleton Pattern

- utilizat pentru a restrictiona numarul de instantieri ale unei clase la un singur obiect
- o metodă de a folosi o singură instanță a unui obiect în aplicație.

Utilizări

- subansamblu al altor pattern-uri:
 - împreună cu pattern-urile Abstract Factory, Builder, Prototype etc.
- obiectele care reprezintă stări
- în locul variabilelor globale.
 - Preferat variabilelor globale deoarece, printre altele, nu poluează namespace-ul global cu variabile care nu sunt necesare.

Implementare

- Implementarea unei metode statice, publice, ce permite crearea unei noi instanțe a clasei dacă aceasta nu exista deja.
- Dacă instanța există deja, atunci întoarce o referință către acel obiect.
- Pentru a asigura o singură instanțiere a clasei, constructorul trebuie să fie *private*
- clasa respectivă va fi instanțiată **lazy** (*lazy instantiation*), utilizând memoria doar în momentul în care acest lucru este necesar deoarece instanța se creează atunci când se apelează metoda ce întoarce instanța
- Avantaj față de clasele non-singleton, pentru care se face *eager instantiation* - se alocă memorie încă de la început, chiar dacă instanța nu va fi folosită

Implementare

- putem implementa o componentă de acest tip în mai multe feluri, inclusiv folosind enum-uri în loc de clase.
- trebuie avut în vedere contextul în care îl folosim, astfel încât să alegem o soluție care să funcționeze corect în toate situațiile ce pot apărea în aplicație
 - unele implementări au probleme atunci când sunt accesate din mai multe thread-uri sau când trebuie serializate

Utilizare:

- dacă avem nevoie să menținem o referință către un obiect în mai multe clase, putem să facem clasa Singleton și să obținem acel obiect în fiecare componentă în care avem nevoie de el
- cod mai curat și cu un flow mai ușor de urmărit, să îl instanțiem doar într-un singur loc și să îl transmitem ca argument.

Implementare - variantă

```
public class Singleton {  
  
    private Singleton instance = new Singleton();  
  
    private Singleton() { }  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
  
}
```

- Instanța singleton este *private*
- Constructorul definit este constructorul privat pentru a împiedica definirea constructorului default

Observație

- codul din exemplul de mai sus funcționează corect în context single-threaded
- în context multi-threading apar probleme legate de ce instanță ajung să obțină thread-urile
- aparent problema poate fi rezolvată folosind keyword-ul *synchronized*, însă nici acest lucru nu este suficient
- soluția este folosirea keyword-ului **static**

=> Toate thread-urile vor avea acces la rezultatele inițializării.

Implementare - variantă

```
public class Singleton {  
    private static Singleton singleton = new Singleton( );  
  
    // Constructor privat – alte clase nu pot instanția  
    private Singleton(){ }  
  
    // metoda statică ce întoarce instanța  
    public static Singleton getInstance( ) {  
        return singleton; }  
  
    // Alte metode protejate de singleton-ness  
    protected static void demoMethod( ) {  
        System.out.println("demoMethod for singleton"); }  
}  
  
class SingletonDemo {  
    public static void main(String[] args) {  
        Singleton tmp = Singleton.getInstance( );  
        tmp.demoMethod( ); }  
}
```


Implementare - variantă

```
public class ClassicSingleton {  
  
    private static ClassicSingleton instance = null;  
  
    protected ClassicSingleton() {  
        // doar pentru a nu permite instanțierea  
    }  
  
    public static ClassicSingleton getInstance() {  
        if(instance == null) {  
            instance = new ClassicSingleton();  
        }  
        return instance;  
    }  
}
```

Observații

- lazy instantiation – pentru crearea singleton-ului;
- instanța este inițial nulă
- instanța singleton nu este creată până când metoda getInstance() nu este apelată pentru prima oară.
- această tehnică asigură faptul că instanța este creată doar atunci când este nevoie de ea.

Tipuri de date enumerare

enum

```
public class CuloriSemafor {  
    public static final int ROSU = -1;  
    public static final int GALBEN = 0;  
    public static final int VERDE = 1;  
}
```

...

// Exemplu de utilizare

```
if (semafor.culoare == CuloriSemafor.ROSU)  
semafor.culoare = CuloriSemafor.GALBEN;
```

// Doar de la versiunea 1.5 !

```
public enum CuloriSemafor { ROSU, GALBEN, VERDE };  
// Utilizarea structurii se face la fel
```

Implementare - enum

- Enum furnizează suport pentru thread safety și este garantată o singură instanță
- Modalitate bună de a avea un singleton cu un minim de efort.

```
public enum EnumSingleton {  
    INSTANCE;  
    public void someMethod(String param) {  
        // some class member  
    }  
}
```

Aplicabilitate

- utilizat des în situații în care avem obiecte care trebuie accesate din mai multe locuri ale aplicației:
- obiecte de tip logger
- obiecte care reprezintă resurse partajate (conexiuni, sockets etc.)
- obiecte ce conțin configurații pentru aplicație
- pentru obiecte de tip Factory.

Exemple din API-ul Java:

- `java.lang.Runtime`
- `java.awt.Toolkit`

Factory

- Oferă o soluție legată de crearea obiectelor =>face parte din categoria *Creational Patterns*
- Folosite pentru obiecte care generează instanțe de clase înrudite (implementează aceeași interfață, moștenesc aceeași clasă abstractă).
- Acestea sunt utilizate atunci când dorim să izolăm obiectul care are nevoie de o instanță de un anumit tip de crearea efectivă a acesteia.
- Clasa care va folosi instanța nici nu are nevoie să specifice exact clasa obiectului ce urmează a fi creat, deci nu trebuie să cunoască toate implementările acelui tip, ci doar ce caracteristici trebuie să aibă obiectul creat.
- Situația cea mai întâlnită: trebuie instanțiate mai multe clase care implementează o anumită interfață sau extind o altă clasă (eventual abstractă).
- Clasa care folosește aceste subclase nu trebuie să “știe” tipul lor concret ci doar pe al părintelui.

Implementare

```
public class ProductFactory{

    public Product createProduct(String ProductID){
        switch (id){
            case ID1: return new OneProduct();
            case ID2: return new AnotherProduct();
            ... // la fel pt alte produse
            default: return null;
        }
    }
    ...
}
```

Exemplu

```
abstract class User{  
    public abstract double getRole();  
}
```

```
class Admin extends User {  
    public String getRole() {  
        return "ADMIN";  
    }  
}
```

```
class Student extends User {  
    public String getRole() {  
        return "STUDENT";  
    }  
}
```


Exemplu

```
class UserFactory {  
    public enum UserType {  
        Adm,  
        Stud  
    }  
  
    public static User createUser(UserType userType) {  
        switch (userType) {  
            case Adm:  
                return new Admin();  
            case Stud:  
                return new Student();  
        }  
        throw new IllegalArgumentException("The user type " +  
            userType + " is not recognized.");  
    }  
}
```

Exemplu

```
class UserCreation{
    /*
     * Create all available users
     */
    public static void main (String args[]) {
        for (UserFactory.UserType userType:
                UserFactory.UserType.values()) {
            System.out.println("Role of " + userType + " is " +
                    UserFactory.createUser(userType).getRole());
        }
    }
}
```

Output:

Role of Adm is ADMIN

Role of Stud is STUDENT

Utilizare Factory și Singleton

- *De ce am avea nevoie de această combinație?*
 - De obicei avem nevoie ca o clasă *factory* să fie utilizată din mai multe componente ale aplicației
 - Este suficient să avem o singură instanță a factory-ului și să o folosim pe aceasta.
 - Folosind pattern-ul Singleton putem face clasa *factory* un *singleton*
- => din mai multe clase putem obține instanța acesteia și cu ajutorul ei să fie create instanțe ale unor resurse ale căror tip nu este cunoscut codului ce le utilizează.

Exemplu - Java Abstract Window Toolkit (AWT)

- *java.awt.Toolkit*
- clasă abstractă ce face legatura dintre componentele AWT și implementările native din toolkit.
- are o metoda factory *Toolkit.getDefaultToolkit()* ce întoarce subclasa de Toolkit specifică platformei
- obiectul Toolkit este un Singleton deoarece AWT are nevoie de un singur obiect pentru a efectua legaturile și deoarece un astfel de obiect este destul de costisitor de creat.
- Metodele trebuie implementate în interiorul obiectului și nu pot fi declarate statice deoarece implementarea specifică nu este cunoscută de componentele independente de platformă.

Design pattern-ul *Command*

- *Behavioral pattern*: legat de interacțiunea dintre componente, de felul în care se efectuează apelurile.
- Este utilizat un obiect pentru a reprezenta și încapsula toate informațiile necesare pentru a apela mai târziu o anumită metodă:
 - Numele metodei, obiectul de care ține metoda și valorile parametrilor metodei.
- Permite:
 - acțiuni (comenzi) multiple
 - decuplarea alegerii operației executate de locul unde este emisă comanda.
- folosește o interfață generală, cu o singură funcție, de felul următor:

```
public interface Command {  
    public void execute(); // execută o acțiune nedefinită încă  
}
```

Command pattern – descriere

- Entități:
 - Command
 - Receiver
 - Invoker
 - Client
- Un obiect *command* are un obiect *receiver* și invocă o metodă a *receiver*-ului într-un mod specific clasei acelui *receiver*.
- *Receiver*-ul realizează acțiunea.
- Un obiect *command* este transmis separat unui obiect *invoker*, care invocă comanda, și opțional, realizează arhivarea execuției comenzii.
- Obiectul *invoker* poate primi orice obiect *command*.
- Atât un obiect *invoker* cât și obiectele *command* sunt păstrate de un obiect *client*.
- *Clientul* conține decizia legată de ce comandă să se execute și la ce moment. Pentru a executa o comandă, el transmite obiectul *command* obiectului *invoker*.

Tipuri de componente (roluri):

- **Invoker** – cere execuția unor comenzi
 - apelează acțiuni - metode oferite de obiectele de tip *Command*)
 - poate menține, o *listă a tuturor comenzilor aplicate* pe obiectul (obiectele) comandate.
 - primește clase *Command* pe care să le invoce
- **Receiver**
 - clasa asupra căreia se face apelul
 - conține implementarea efectivă a ceea ce se dorește executat
- **Command** - obiectele pentru reprezentarea comenzilor implementează această interfață sau o extind (dacă este clasă abstractă)
 - conțin metode cu nume sugestiv pentru executarea acțiunii comenzii - *execute()*
 - conțin referințe către *receivers* pentru a realiza comanda
 - *concrete command* - implementări/subclase
- **Client** – crează comenzile concrete și setează receiver-ul lor

Exemplu

- Un “switch” cu două comenzi: aprinderea/stingerea luminii.
- Acest switch poate fi folosit cu orice alt gen de acțiune pentru un alt aparat, nu doar pentru aprinderea-stingerea luminii, metoda Switch-ului poate primi orice subclasă Command ca parametru.
- Exemplu: îl putem configura pentru pornirea unui motor.

```
import java.util.List;  
import java.util.ArrayList;
```

```
/* interfata Command */  
public interface Command {  
    void execute();  
}
```


Exemplu

```
/* Invoker */
```

```
public class Switch {  
    private List<Command> history = new ArrayList<Command>();  
    public void storeAndExecute(Command cmd) {  
        history.add(cmd);        // optional  
        cmd.execute();  
    }  
}
```

```
/* Receiver */
```

```
public class Light {  
    public void turnOn() {  
        System.out.println("The light is on");  
    }  
    public void turnOff() {  
        System.out.println("The light is off");  
    }  
}
```

Exemplu

```
/* Comanda pentru aprinderea luminii – comanda concreta 1 */
public class FlipUpCommand implements Command {
    private Light theLight;
    public FlipUpCommand(Light light) {
        theLight = light;
    }
    public void execute(){
        theLight.turnOn();
    }
}

/* Comanda pentru stingerea luminii – comanda concreta 2 */
public class FlipDownCommand implements Command {
    private Light theLight;
    public FlipDownCommand(Light light) {
        theLight = light;
    }
    public void execute() {
        theLight.turnOff();
    }
}
```

Exemplu

```
/* Clasa test sau client */
public class PressSwitch {
    public static void main(String[] args){
        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);
        Switch mySwitch = new Switch();
        try {
            if ("ON".equalsIgnoreCase(args[0]))
                mySwitch.storeAndExecute(switchUp);
            else if ("OFF".equalsIgnoreCase(args[0]))
                mySwitch.storeAndExecute(switchDown);
            else
                System.out.println("\"ON\" sau \"OFF\" !!!");
        } catch (Exception e) {
            System.out.println("Introduceti argumentele!!");
        }
    }
}
```

Design pattern-ul *Command*

- In Swing există mai multe interfețe "ascultător" corespunzătoare interfeței "Command". Exemplu:

```
public interface ActionListener {  
    public void actionPerformed( ActionEvent ev);  
}
```

- Un obiect dintr-o clasă ascultător este un obiect "comandă" ce realizează o acțiune.
 - singura legătură dintre partea de interfață grafică a aplicației și partea de logică specifică aplicației (tratarea evenimentelor).
 - Se realizează astfel decuplarea celor două părți, ceea ce permite modificarea lor separată și chiar selectarea unei anumite acțiuni în cursul execuției.

Exemplu fără command pattern

```
...
JMenuBar mbar = new JMenuBar();
setJMenuBar (mbar);
JMenu mFile = new JMenu ("File");
mbar.add (mFile);
JMenuItem open = new JMenuItem ("Open");
    // optiune meniu vertical
JMenuItem exit = new JMenuItem ("Exit");
    // optiune meniu vertical
mFile.add (open);
mFile.addSeparator( );
mFile.add (exit);
open.addActionListener (this);
exit.addActionListener (this);
.....
// Tratarea evenimentelor
public void actionPerformed (ActionEvent e) {
    Object source = e.getSource(); // sursa evenimentului
    if ( source == open) fileOpen( );// actiune asociata lui "Open"
    else if (source == exit) System.exit(0); // actiune asociata lui "Exit"
}
```

Exemplul cu *command pattern*

- o interfață "*Command*"
public interface **Command** {
 public void execute();
}
- *definirea mai multor clase care implementează această interfață:*

```
class FileExitCmd extends JMenuItem implements Command {  
    public FileExitCmd (String optname) {  
        super(optname); // nume optiune (afisat in meniu)  
    }  
    public void execute () {  
        System.exit(0); // actiune asociata optiunii exit  
    }  
}
```

```
class FileOpenCmd extends JMenuItem implements Command  
{  
    public FileOpenCmd (String optname) {  
        super(optname); // nume optiune (afisat in meniu)  
    }  
}
```

Exemplul cu *command pattern*

```
public void execute () {  
    fileOpen();    // actiune asociata optiunii open  
}  
}
```

```
class CmdListener implements ActionListener {  
    public void actionPerformed (ActionEvent e) {  
        Command c = (Command) e.getSource();  
        c.execute();  
    }  
} // Invoker
```

```
...  
ActionListener cmd = new CmdListener();  
open.addActionListener (cmd);  
exit.addActionListener (cmd);
```

- Metoda "execute" : metodă polimorfică
 - selectarea unei implementări sau alta se face în funcție de tipul variabilei **c**, deci de tipul obiectului care este sursa evenimentului!

Exemplu utilizare Command pattern

- În programarea meniurilor si barelor de instrumente ("toolbar"), prin utilizarea obiectelor "actiune" - Action
- implementează interfata *Action*.
- O bară de instrumente contine mai multe butoane cu imagini (obiecte *JButton*), dar are acelasi rol cu o bară meniu: selectarea de actiuni de către operator.
- In loc să se adauge celor două bare obiecte diferite (dar care produc aceeasi actiune) se adaugă un singur obiect, de tip *Action*, care contine o metodă *actionPerformed*; câte un obiect pentru fiecare actiune selectată.
- Interfata *Action* ⇔ interfața *Command*
- metoda *actionPerformed* ⇔ metoda *execute*.
- Interfata *Action* extinde interfata *ActionPerformed* cu două metode *setValue* și *getValue*:
 - stabilirea si obtinerea proprietăților unui obiect actiune (text afisat în optiune meniu, imagine afisată pe buton din bara de instrumente, o scurtă descriere ("tooltip") a "instrumentului" afisat)

Utilizare



- Facilitează construirea de componente generale care au nevoie să delege, sau să execute apeluri de metode fără să fie necesar să cunoască clasa metodei sau parametrii metodei în momentul în care realizează aceasta!
- Utilizarea unui obiect invoker permite
 - păstrarea evidenței execuției comenzilor
 - implementarea mai multor tipuri de comenzi care sunt gestionate de invoker
- fără să fie necesar ca clientul să știe că există această arhivare sau aceste tipuri!

Observer Pattern

- Behavioral pattern (comportamental)
 - o organizare mai bună a comunicației dintre clase, funcție de rolurile/comportamentul lor.
- Subiect - Obiectul observat :
 - un obiect care poate suferi diverse modificări
 - conține o listă de referințe la obiecte *observer*
 - la producerea unui eveniment, apelează o anumită metodă a obiectelor *observer* înregistrate la el.
- Observatori - unul sau mai multe obiecte observator:
 - trebuie anuntate imediat de orice modificare în obiectul *observat*, pentru a realiza anumite acțiuni.
- Clasele Observer **observă** modificările/acțiunile clasei Subject

Observer Pattern

- Mecanism de decuplare a obiectelor observabile,
 - care generează evenimente
- de obiectele observator,
 - care recepționează evenimentele respective
- Relația dintre o clasă JFC generator de evenimente și o clasă ascultător (receptor) care reacționează la evenimente este similară relației dintre o clasă observată și o clasă observator!

Exemplu:

- un buton *JButton* are rolul de obiect *observat*
- o clasă care implementează interfața *ActionListener* (contine metoda *actionPerformed*) are rolul de *observator*.

Exemplu API Java

- Schema observat-observator a generat în Java
 - clasa *Observable*
 - interfața *Observer*
 - din pachetul *java.util*
- Programatorul va defini una sau mai multe clase cu rol de observator, compatibile cu interfața *Observer*.
- Motivul existenței acestei interfețe: în clasa *Observable* (pentru obiecte observate) există metode cu argument de tip *Observer* pentru menținerea unui vector de obiecte observator.

Exemplu:

```
public void addObserver(Observer o) {  
    //adauga un nou observator la lista  
    if (!observers.contains(o))  
        // "observers" este vector din clasa Observable  
        observers.addElement(o);  
}
```

Observer - Observable

Interfața *Observer*:

```
public interface Observer {  
    void update(Observable o, Object arg);  
    // o este obiectul observat  
}
```

- apelată de un obiect observat la o schimbare în starea sa care poate interesa obiectele observator înregistrate anterior

Clasa *Observable*:

- nu este abstractă dar nici nu este direct utilizabilă
- se va defini o subclasă a sa, adăugând o serie de metode specifice aplicației care apelează din superclasă metodele:
 - `setChanged`
 - `notifyObservers`
 - apelează metoda ***update*** pentru toți observatorii introduși în vectorul *observers*

Clasa Observable - fragment

```
public class Observable {  
    private boolean changed = false; //daca s-a produs o schimbare  
    private Vector obs;              // lista de obiecte observator  
    public void addObserver(Observer o) {  
        // adauga un observator la lista  
        if (!obs.contains(o))  
            obs.addElement(o);  
    }  
    public void notifyObservers(Object arg) {  
        // notificare observatori de schimbare  
        if (!changed) return;  
        for (int i = arrLocal.length-1; i>=0; i--)  
            ((Observer)obs.elementAt(i)).update(this, arg);  
    }  
    protected void setChanged() {  
        // comanda modificare stare obiect observat  
        changed = true;  
    }  
}
```

...

Exemplu utilizare Observable-Observer

- Exemplu de definire a unei subclase pentru obiecte observate:

```
class MyObservable extends Observable {  
    public void change() {    // metoda adaugata in subclasa  
        setChanged();        // din clasa Observable  
        notifyObservers();  
        // anunta observatorii ca a aparut o schimbare  
    }  
}
```

- Clasa observator: metoda *update* adaugă câte un caracter # pe ecran (la fiecare apelare de către un obiect MyObservable).

```
class ProgressBar implements Observer {  
    public void update( Observable obs, Object x ) {  
        System.out.print('#');  
    }  
}
```

Exemplu - continuare

- Utilizare a obiectelor observat-observator definite anterior:

```
public static void main(String[ ] arg) {  
    int n=1000000000, m=n/100;  
    ProgressBar bar= new ProgressBar();  
    MyObservable model = new MyObservable();  
  
    model.addObserver(bar);  
  
    for (int i=0;i<n;i++) // modifica periodic stare obiect  
        observat  
            if ( i%m==0)  
                model.change();  
}
```


Example JDK

- Interfețele *ActionListener*, *ItemListener* s.a.
 - rol similar cu interfața *Observer*
- metodele "actionPerformed" s.a.
 - corespund metodei "update".
- Programatorul de aplicație definește clase ascultător compatibile cu interfetele "xxxListener", clase care implementează metodele de tratare a evenimentelor
- Corespondența dintre metodele celor două grupe de clase:
- addObserver
 - addActionListener
 - addChangeListener
- notifyObservers
 - fireActionEvent
 - fireChangeEvent
- update
 - actionPerformed
 - stateChanged

Visitor pattern

- *Behavioral pattern* - comportamental
- Modalitate de a *separa un algoritm de structura* pe care acesta operează.

Putem adăuga noi:

- posibilități de prelucrare a structurii, fără să o modificăm
- funcții care realizează prelucrări asupra unei familii de clase, fără a modifica efectiv structura claselor

Structura:

- Element (sau *Observable*) și Visitor
- Element: are o metodă *accept* care poate primi ca argument un *Visitor*
 - Metoda *accept* apelează metoda *visit* din *Visitor*; *Element-ul* se transmite pe sine însuși ca argument al metodei *visit*

Exemplu



- Algoritm pentru parcurgerea elementelor unui graf
- Mai multe operații ce pot fi realizate în timpul acestei parcurgeri prin furnizarea a diferite tipuri de vizitatori care să interacționeze cu elementele grafului
- Are la bază tipul dinamic atât al elementelor grafului cât și al vizitatorilor – double dispatch!
- Double dispatch - Metoda apelată este determinată la *runtime* de doi factori!

Double Dispatch

Atunci când metoda *accept* e apelată în program, implementarea ei este determinată pe baza:

- Tipului dinamic (la execuție) al *Element-ului* și
- Tipului static al *Visitor-ului*.

Atunci când metoda asociată *visit* e apelată în program, implementarea ei este determinată pe baza:

- Tipului dinamic al *Visitor-ului* și
- Tipului static al *Element-ului* așa cum este el cunoscut din implementarea metodei *accept*, care înseamnă de fapt tipul dinamic al *Element-ului*

Prin urmare, implementarea metodei *visit* este determinată pe baza:

- Tipului dinamic al *Element-ului* și
- Tipului dinamic al *Visitor-ului*

Exemplul 1



- Afișarea conținutului unui arbore de noduri, care în acest caz descrie componentele unei mașini.
- În loc să scriem metoda "print" pentru fiecare subclasă (Wheel, Engine, Body și Car), avem o singură clasă (CarElementPrintVisitor) ce realizează acțiunea de afișare dorită.
- Deoarece diferitele subclase necesită acțiuni ușor diferite pentru a afișa corect, CarElementDoVisitor determină acțiunile pe baza clasei argumentului primit de către el.

Implementare

```
interface ICarElementVisitor {  
    void visit(Wheel wheel);  
    void visit(Engine engine);  
    void visit(Body body);  
    void visit(Car car);  
}
```

```
interface ICarElement {  
    void accept(ICarElementVisitor visitor);  
}
```

```
class Wheel implements ICarElement {  
    private String name;  
    public Wheel(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return this.name;  
    }  
}
```

Implementare

```
public void accept(ICarElementVisitor visitor) {  
    /*  
    * accept(ICarElementVisitor) din Wheel implementeaza  
    * accept(ICarElementVisitor) din ICarElement, astfel incat  
    * apelul lui accept se face prin dynamic binding (la execuție).  
    * Acesta poate fi considerat “the first dispatch”.  
    * Decizia de a apela visit(Wheel) (si nu visit(Engine) etc.)  
    * poate fi luată la compilare deoarece 'this' este cunoscut în  
    * momentul compilării ca fiind un Wheel. Pe lângă aceasta, fiecare  
    * implementare a lui ICarElementVisitor implementează  
    * visit(Wheel), care este o altă decizie ce este luată în momentul  
    * execuției. Acesta poate fi considerat “the second dispatch”.  
    */  
    visitor.visit(this);  
}  
  
class Engine implements ICarElement {  
    public void accept(ICarElementVisitor visitor) {  
        visitor.visit(this);  
    }  
}
```

Implementare

```
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car implements ICarElement {
    ICarElement[] elements;
    public Car() {
        elements = new ICarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }
    public void accept(ICarElementVisitor visitor) {
        for(ICarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}
```


Implementare

```
class CarElementPrintVisitor implements ICarElementVisitor {  
  
    public void visit(Wheel wheel) {  
        System.out.println("Visiting " + wheel.getName() + " wheel");  
    }  
  
    public void visit(Engine engine) {  
        System.out.println("Visiting engine");  
    }  
  
    public void visit(Body body) {  
        System.out.println("Visiting body");  
    }  
  
    public void visit(Car car) {  
        System.out.println("Visiting car");  
    }  
}
```

Implementare

```
class CarElementDoVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
    public void visit(Body body) {
        System.out.println("Moving my body");
    }
    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}

public class VisitorDemo {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Output



Visiting front left wheel

Visiting front right wheel

Visiting back left wheel

Visiting back right wheel

Visiting body

Visiting engine

Visiting car

Kicking my front left wheel

Kicking my front right wheel

Kicking my back left wheel

Kicking my back right wheel

Moving my body

Starting my engine

Starting my car

Observație

- O abordare mai flexibilă a acestui pattern este de a crea o *clasă wrapper* care implementează interfața care definește metoda *accept*.
- Această clasă conține o referință la `ICarElement` care poate fi inițializată prin constructor.
- Această abordare evită necesitatea ca fiecare element să implementeze o interfață.

```
class Wrapper implements interface ICarElement {  
    ICarElement el;  
    public Wrapper(ICarElement el){  
        this.el=el;}  
    public void accept(ICarElementVisitor visitor){ }  
}
```

Exemplul 2 - Implementare

```
interface Visitor {
    public void visit(Student e);
    public void visit(Profesor b);
}

interface Visitable {
    public void accept(Visitor v);
}

class Student implements Visitable {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class Profesor implements Visitable {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

Implementare

```
public class Test {  
    public static void main(String[] args) {  
        ...  
        Visitor v = new ConcreteVisitor();  
        for (Visitable e : persoane) // colectie de Persoane  
            e.accept(v);  
    }  
}
```

- vizitator ce afișează datele fiecărei persoane (Student sau Profesor):

```
public class ConcreteVisitor implements Visitor {  
    public void visit(Student s) {  
        System.out.println(s.nume() + " " + s.grupa());  
    }  
    public void visit(Profesor p) {  
        System.out.println(p.nume() + " " + p.materia());  
    }  
}
```