Design Patterns Basics

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

- 1 Introducere
- 2 Singleton Pattern
  - 2.1 Utilizari
  - 2.2 Class Diagram
  - 2.3 Implementare
  - 2.4 Exemplu
- 3 Factory Pattern
  - 3.1 Exemplu
  - 3.2 Exemplu de utilizare impreuna cu Singleton
- 4 Observer Pattern
  - 4.1 Motivatie
  - 4.2 Structura
  - 4.3 Implementare
  - 4.4 Resurse utile
- 5 Command Pattern
  - 5.1 Structura
  - 5.2 Implementare
  - 5.3 Resurse utile
- 6 Exercitii

Introducere

Un design pattern este o solutie generala si reutilizabila a unei probleme comune in design-ul software. Un design pattern nu este un design in forma finala, ceea ce inseamna ca nu poate fi transformat direct in cod. Acesta este o descriere a solutiei sau un template ce poate fi aplicat pentru rezolvarea problemei. In general pattern-urile orientate obiect arata relatiile si interactiunile dintre clase sau obiecte, fara a specifica insa forma finala a claselor sau obiectelor implicate.

Design Pattern-urile fac parte din domeniul modulelor si interconexiunilor. La un nivel mai inalt se gasesc pattern-urile arhitecturale (Architectural Patterns) ce descriu pattern-ul global utilizat al intregului sistem.

Nu toate pattern-urile software sunt design patterns. De exemplu, algoritmii rezolva probleme computationale, nu probleme de design.

Design pattern-urile au fost initial grupate in urmatoarele categorii: *Creational patterns*, *Structural patterns*, si *Behavioral patterns* si au fost descrise folosind conceptele de delegare, agregare si consultare.

Creational Patterns sunt pattern-uri ce implementeaza mecanisme de creare a obiectelor. In aceasta categorie se incadreaza pattern-urile **Singleton** si **Factory**.

Structural Patterns sunt pattern-uri ce simplifica design-ul aplicatiei prin gasirea unei metode de a defini relatiile dintre entitati. In aceasta categorie se incadreaza pattern-ul **Decorator**.

Behavioral Patterns sunt pattern-uri ce definesc modul in care obiectele comunica intre ele. In aceasta categorie se incadreaza pattern-urile **Command**, **Visitor** si **Observer**.

Singleton Pattern

Pattern-ul Singleton este utilizat pentru a restrictiona numarul de instantieri ale unei clase la un singur obiect. **Atentie:** Acest pattern ingreuneaza testarea aplicatiei, deoarece introduce stari globale.

Utilizari

Pattern-ul Singleton este in general utilizat in urmatoarele cazuri:

1. Ca parte integrata in Pattern-urile Abstract Factory, Builder, Prototype
2. Obiectele ce reprezinta stari sunt deseori Singleton-uri.
3. Singleton este preferat variabilelor globale deoarece:
   - nu polueaza namespace-ul global cu variabile nenecesare.

- permite initializarea intarziata (lazy initialization) pentru a nu consuma inutil resursele sistemului.

Din punct de vedere al design-ului si testarii unei aplicatii de multe ori se evita folosirea acestui pattern. De exemplu daca avem nevoie sa mentinem o referinta catre un obiect in mai multe clase, putem sa il facem Singleton si sa obtinem acel obiect in fiecare componenta in care avem nevoie de el sau ca sa facem codul mai curat si cu un flow mai usor de urmarit, sa il instantiem doar intr-un singur loc si sa il transmitem ca argument. Incercati sa nu folositi in exces metode statice (mai mult pt functii utility) si componente Singleton.

Class Diagram

Implementare

La baza pattern-ului Singleton sta o metoda ce permite crearea unei noi instante a clasei daca aceasta nu exista deja. Daca instanta exista deja, atunci intoarce o referinta catre acel obiect. Pentru a asigura o singura instantiere a clasei, constructorul trebuie facut **protected** (un constructor privat impiedica reutilizarea sa sau accesul unei unitati de testare).

Diferenta dintre o clasa cu atribute si metode statice si un Singleton este aceea ca Singleton-ul permite **instantierea lazy**, utilizand memoria doar in momentul in care acest lucru este necesar deoarece instanta se creeaza atunci cand se apeleaza `getInstance()`. Inca un avantaj ar fi faptul ca o clasa Singleton poate fi extinsa si metodele ei suprascrise, insa intr-o clasa cu metode statice acestea nu pot fi suprascrise (overriden) (o discutie pe aceasta tema puteti gasi aici, si o comparatie intre static si dynamic binding aici

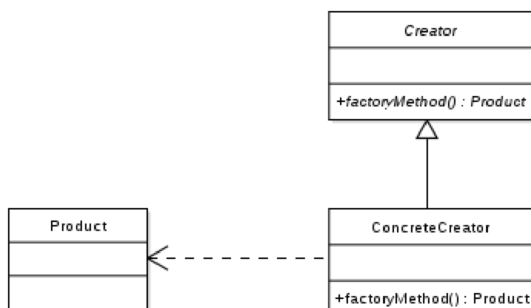
Exemplu

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  // Private constructor prevents instantiation from other classes
  private Singleton() {}

  public static Singleton getInstance() {
     return INSTANCE;
  }
}
```

Alt exemplu de implementare este exercitiul 4 de la laboratorul 3 (Organizarea surselor si controlul accesului).

Factory Pattern

Patternul Factory face parte din categoria *Creational Patterns* si ca atare rezolva problema crearii unui obiect fara a specifica exact clasa obiectului ce urmeaza a fi creat. Acest lucru este implementat prin definirea unei metode al carei scop este crearea obiectelor. Metoda va avea specificat ca parametru de intors in antet un obiect de tip parinte, urmand ca, in functie de alegerea programatorului, aceasta sa creeze si sa intoarca obiecte noi de tip copil.



Exemplu

Situatia cea mai intalnita in care se potriveste acest pattern este aceea cand trebuie instantiate multe clase care implementeaza o anumita interfata sau extind o alta clasa (eventual abstracta), ca in exemplul de mai jos. Clasa care foloseste aceste subclase nu

trebuie sa "stie" tipul lor concret ci doar pe al parintelui.

```
abstract class Pizza {
    public abstract double getPrice();
}

class HamAndMushroomPizza extends Pizza {
    public double getPrice() {
        return 8.5;
    }
}

class DeluxePizza extends Pizza {
    public double getPrice() {
        return 10.5;
    }
}

class HawaiianPizza extends Pizza {
    public double getPrice() {
        return 11.5;
    }
}

class PizzaFactory {
    public enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    }

    public static Pizza createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw new IllegalArgumentException("The pizza type " + pizzaType + " is not recognized.");
    }
}

class PizzaLover {
    /*
     * Create all available pizzas and print their prices
     */
    public static void main (String args[]) {
        for (PizzaFactory.PizzaType pizzaType : PizzaFactory.PizzaType.values()) {
            System.out.println("Price of " + pizzaType + " is " +
PizzaFactory.createPizza(pizzaType).getPrice());
        }
    }
}

Output:
Price of HamMushroom is 8.5
Price of Deluxe is 10.5
Price of Hawaiian is 11.5
```

Exemplu de utilizare impreuna cu Singleton

De ce am avea nevoie de aceasta combinatie?

- De obicei avem nevoie ca o clasa factory sa fie utilizata din mai multe componente ale aplicatiei. Ca sa economisim memorie este suficient sa avem o singura instanta a factory-ului si sa o folosim pe aceasta. Folosind pattern-ul Singleton putem face clasa factory un singleton, si astfel din mai multe clase putem obtine instanta acesteia si cu ajutorul acesteia sa obtinem/sa fie create instante ale unor resurse ale caror tip nu este cunoscut codului ce le utilizeaza.

Un exemplu ar fi Abstract Window Toolkit (AWT). `java.awt.Toolkit` este o clasa abstracta ce face legatura dintre componentele AWT si implementarile native din toolkit. Clasa Toolkit are o metoda factory Toolkit.getDefaultToolkit() ce intoarce subclasa de Toolkit specifica platformei. Obiectul Toolkit este un Singleton deoarece AWT are nevoie de un singur obiect pentru a efectua legaturile si deoarece un astfel de obiect este destul de costisitor de creat. Metodele trebuie implementate in interiorul obiectului si nu pot fi declarate statice deoarece implementarea specifica nu este cunoscuta de componentele independente de platforma.

Observer Pattern

Design Pattern-ul *Observer* defineste o relatie de dependenta 1..* intre obiecte astfel incat cand un obiect isi schimba starea, toti dependentii lui sunt notificati si actualizati automat. Implica existenta unui obiect denumit *subiect* care are asociata o lista de obiecte dependente, numite *observatori*, pe care le apeleaza automat de fiecare data cand se intampla o actiune.

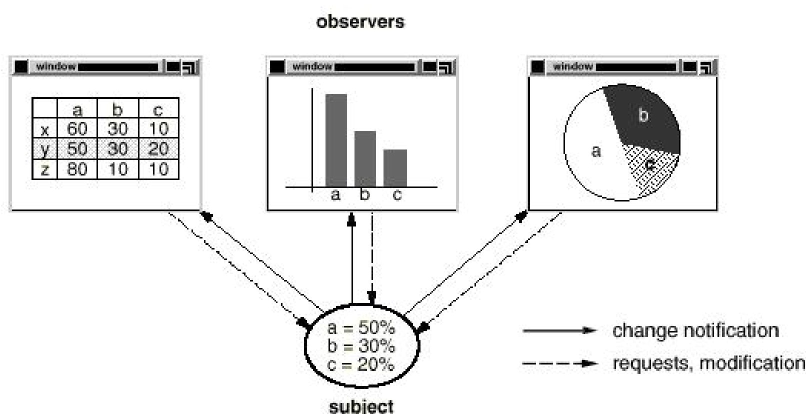
Acest pattern este de tip **Behavioral** (comportamental), deorece faciliteaza mai buna organizare a comunicatiei dintre clase in functie de rolurile/comportamentul acestora.

Motivatie

Design pattern-ul Observer se foloseste in cazul in care mai multe clase(*observatori*) depind de comportamentul unei alte clase(*subiect*), in situatii de tipul:

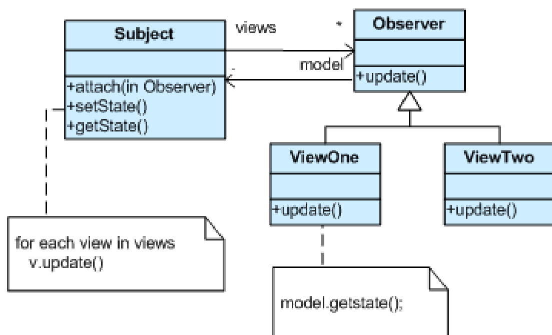
- o clasa implementeaza/reprezinta logica, componenta de baza, iar alte clase doar folosesc rezultate ale acesteia (monitorizare).
- o clasa efectueaza actiuni care apoi pot fi reprezentate in mai multe feluri de catre alte clase (view-uri ca in figura de mai jos).

Practic in toate aceste situatii clasele Observer **observa** modificarile/actiunile clasei Subject. Observarea se implementeaza prin **notificari initiate din metodele clasei Subject**. De exemplu in laboratorul 6, clasa MyArrayList era observata pentru evenimentele de adaugare si stergere din lista.



Structura

Figura de mai jos ofera un exemplu de baza de structurare al aplicatiei astfel incat sa se foloseasca observatori.



Subject

- cunoaste observatorii (numar arbitrar)
- subiectul nu trebuie sa stie ce fac observatorii, trebuie doar sa ii cunoasca si sa ii notifice.
- ca implementare:
  - trebuie sa ofere o metoda prin care primeste referinte catre obiecte de tip observator, eventual si o metoda prin care se pot deinregistreaza observatori
  - trebuie sa pastreze referintele observatorilor si sa apeleze metodele de notificare oferite de interfata acestora

Observer

- defineste o interfata de actualizare a obiectelor ce trebuie notificate de schimbarea subiectelor
- ca implementare:
  - toti observatorii pentru un anumit subiect trebuie sa implementeze aceasta interfata
  - metoda (metode) ce sunt invocate de subiect pentru notifica o schimbare. Ca argumente se poate primi chiar instanta subiectului sau obiecte speciale care reprezinta evenimentul ce a provocat schimbarea (ca in exemplul de la MyArrayList din laboratorul 6).

View

- implementeaza interfata Observator

Aceasta schema se poate extinde, in functie de scopul dorit - observatorii pot tine referinte catre subiect sau putem adauga clase speciale pentru reprezentarea evenimentelor, notificarilor. Un alt exemplu il puteti gasi aici

Implementare

Un exemplu de implementare este exercitiul 3 de la laboratorul 6 (Clase interne).

Tookit-ul Swing [1] pentru crearea interfetelor grafice foloseste design pattern-ul observer.

Resurse utile

Exemple simple 1 2 si explicatii despre acest pattern puteti gasi si aici.

Wikipedia

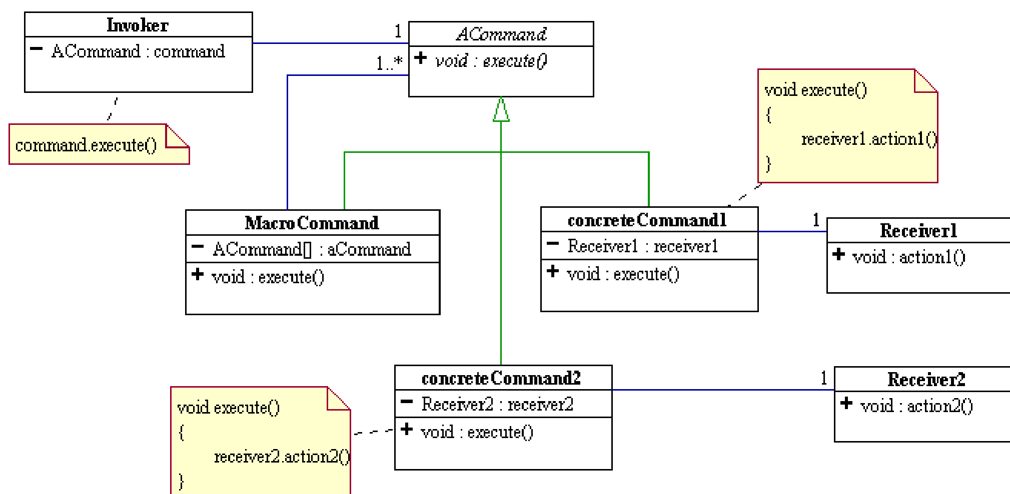
Command Pattern

Design pattern-ul *Command*

- incapsulează o cerere sub forma unui obiect;
- permite parametrizarea clienților cu cereri diferite;
- permite memorarea cererilor intr-o coada;
- suporta operatii reversibile (*undoable operations*)

Structura

Ideea principala este de a crea un obiect de tip **Command** care va retine parametrii pentru comanda. Comandantul retine o referinta la comanda si nu la comandat. Comanda propriu-zisa este anuntata obiectului comanda (de catre comandant) prin executia unei metode specificate asupra lui. Obiectul `Command` este apoi responsabil de trimiterea (dispatch) a comenzii la obiectele care o indeplinesc (comandati).



Tipuri de componente (**roluri**):

- **Invoker** - comandantul
  - apeleaza actiuni pe comenzi (invoca metode oferite de obiectele de tip Command)
  - poate mentine, daca e cazul, o *lista a tutoror comenzilor aplicate* pe obiectul(obiectele) comandate. Este necesara retinerea acestei liste de comenzi atunci cand implementam un comportament de undo/redo al comenzilor.
- **Receiver** - cel care "stie"/poate sa execute comenzile (comandatul)
- **Command** - obiectele pentru reprezentarea comenzilor implementeaza aceasta interfata/o extind daca este clasa abstracta
  - *concrete command* - ne referim la implementari/subclasele acesteia
  - de obicei contin metode cu nume sugestiv pentru executarea actiunii comenzii (e.g. `execute()`)
  - in cazul implementarii unor actiuni *undoable* adaugam metode pentru `undo` si/sau `redo`.
  - tin referinte catre comandati (receivers) pentru a aplica/invoca actiunea ce reprezinta acea comanda

In diagrama de mai sus, comandantul este clasa `Invoker` care contine o referinta la o instanta (command) a clasei (ACommand).

`Invoker` va apela metoda abstracta `execute()` pentru a cere indeplinirea comenzii.

Comenzile sunt instantele subclaselelor din `ACommand`: `ConcreteCommand1` si `ConcreteCommand2`.

- `ConcreteCommand2` reprezinta o clasa concreta care trimite comanda "comandatului" `Receiver2`.
- `ConcreteCommand1` extinde clasa abstracta `MacroCommand` care reprezinta o comanda care va fi indeplinita de mai multi "comandati"(receivers). Clasa `MacroCommand` extinde clasa abstracta `ACommand`.

Nota: In Java, se pot folosi si interfete in loc de clase abstracte, depinzand de problema concreta de rezolvat.

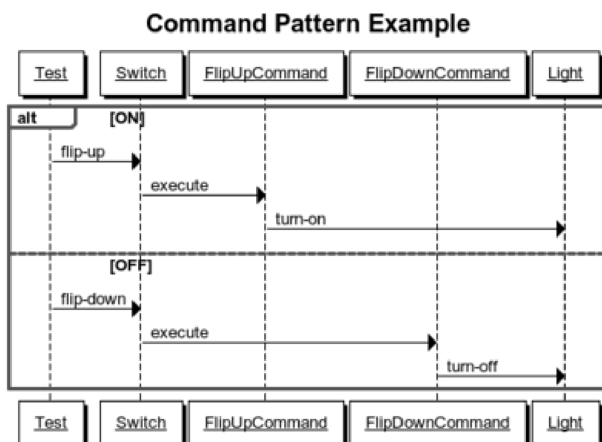
Implementare

Un exemplu de implementare care foloseste interfete in loc de clase abstracte este urmatorul:

```
/*the Invoker class*/
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
         this.flipUpCommand = flipUpCmd;
         this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
         flipUpCommand.execute();
    }

    public void flipDown() {
         flipDownCommand.execute();
    }
}

/*Receiver class*/
public class Light {

     public Light() {  }

     public void turnOn() {
        System.out.println("The light is on");
     }

     public void turnOff() {
        System.out.println("The light is off");
     }
}

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

/*the Command for turning on the light*/
public class FlipUpCommand implements Command {

   private Light theLight;

   public FlipUpCommand(Light light) {
        this.theLight=light;
   }

   public void execute(){
      theLight.turnOn();
   }
}

/*the Command for turning off the light*/
public class FlipDownCommand implements Command {

   private Light theLight;

   public FlipDownCommand(Light light) {
        this.theLight=light;
   }

   public void execute() {
      theLight.turnOff();
   }
}
```

```
/*The test class or client*/
public class PressSwitch {

   public static void main(String[] args) {
       Light lamp = new Light();
       Command switchUp = new FlipUpCommand(lamp);
       Command switchDown = new FlipDownCommand(lamp);

       // See criticism of this model above:
       // The switch itself should not be aware of lamp details (switchUp, switchDown) either directly or indirectly
       Switch s = new Switch(switchUp,switchDown);

       try {
           if (args[0].equalsIgnoreCase("ON")) {
                s.flipUp();
           } else if (args[0].equalsIgnoreCase("OFF")) {
               s.flipDown();
           } else {
               System.out.println("Argument \"ON\" or \"OFF\" is required.");
           }
       } catch (Exception e){
           System.out.println("Arguments required.");
       }
   }
}
```

Diagrama de secventa pentru acest examplu, pentru a intelege mai bine interactiunea dintre componente, este urmatoarea:



Resurse utile

- Descriere Command Pattern si exemplu pentru operatii undo/redo
- Learn how to implement the Command pattern in Java

Exercitii

1. (**4p**) Acest exercitiu are ca scop exemplificarea folosirii pattern-ului ***Observer***.
   - (2p) a) O clasa `MessageSet` tine o lista de mesaje (un text scurt) primite de la utilizator. Atunci cand se adauga un nou mesaj afisati acest eveniment. Decuplati primirea si stocarea mesajului de afisarea evenimentul folosind pattern-ul *Observer*. **Hint:** Un obiect *observator* face afisarea evenimentului, *subiectul* este `MessageSet`.
   - (2p) b) Extindeti exercitiul anterior astfel incat atunci cand se primeste un mesaj sa fie afisat in alte limbi. Folositi obiecte *observator* care fac traducerea si afisarea mesajului in cate o alta limba. Pentru a usura testarea (nu este importanta traducerea in sine, ci design-ul aplicatiei) fixati-va cateva (<10) cuvinte de test a caror traducere o mentineti in observatori in structuri de tip HashMap si testati pe mesaje continand doar acele cuvinte (daca un cuvant nu e in "dictionar" il afisati netradus).
     - clasa `MessageSet` trebuie sa aiba metode pentru:
       - inregistrare observator(i)
       - primire mesaj
       - notificare observator(i)
     - pastrati o *lista de observatori* in clasa `MessageSet`
     - conform descrierii pattern-ului, observatorii extind o clasa abstracta sau interfata ce reprezinta un observator.
2. (**3p**) In cadrul unei aplicatii de editare imagini pentru a reprezenta actiunile efectuate in timpul editarii folosim obiecte pentru fiecare tip de comanda ce modifica imaginea. De exemplu o comanda pentru 'resize', alta pentru 'crop', alta pentru pentru aplicarea unui filtru. Pentru a decupla logica construirii acestor obiecte putem folosi pattern-ul ***Factory***. Implementati o clasa `ImageCommandFactory` care creaza obiecte ce extind clasa abstracta `ImageCommand` (3 tipuri de comenzi, de exemplu ResizeCommand, BlurFilterCommad, CropCommand).
   - clasa `ImageCommand` ofera metoda `execute()` ce afiseaza un mesaj cu explicatia acelei comenzi.

  - clasele pentru comenzi pot sa aiba atribute diferite in functie de comanda.
  - unul din motivele pentru care avem nevoie de clase diferite pentru a reprezenta comenzile este ca sa vedem aplicarea lor ca o lista de actiuni diferite si undoable.

3. (**4p**) Extindeti exercitiul anterior astfel incat sa se poata face undo-redo ale acelor comenzi. Folositi pattern-ul ***Command***.
   - Modificati comenzile astfel incat sa aiba metodele de `undo/redo`. Unde faceti intai aceasta modificare?
   - Creati o clasa `Image` pe care se aplica aceste comenzi. Aceste comenzi doar modifica niste atribute, ca sa "afisati" imaginea trebuie de fapt sa afisati starea atributelor (de exemplu afisati: blurred: yes, size: 100x100 etc).
     - contine si o metoda care aplica comanda (e.g. action()) care va modifica atributele si eventual va afisa un mesaj informativ ca s-a executat.
   - Creati o clasa cu rolul de Invoker (comandant) care invoca comenzile
     - trebuie sa retina o lista de comenzi, iar cand se da `undo/redo` pe imagine se apeleaza `undo/redo` la rand pt comenzile aplicate.

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Design_Patterns_Basics"

- Ultima modificare 14:36, 5 decembrie 2012.
- Această pagină a fost vizitată de 8.209 ori.
- Conţinutul este disponibil sub GNU Free Documentation License 1.2.
- Politica de confidenţialitate
- Despre POO
- Termeni