Visitor Pattern

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Introducere

Design pattern-ul ***Visitor*** ofera o modalitate de a **separa** un **algoritm** de **structura** pe care acesta opereaza. **Avantajul** consta in faptul ca putem **adauga** noi posibilitati de prelucrare a structurii, **fara** sa o modificam. Extrapoland, folosind *Visitor*, putem adauga noi functii care realizeaza prelucrari asupra unei familii de clase, fara a modifica structura claselor.

Motivatie

Before

Fie ierarhia de mai jos, ce defineste un **angajat** (`Employee`) si un **sef** (`Boss`), vazut, de asemenea, ca un angajat:

```
class Employee {

        String  name;
        float   salary;

        public Employee(String name, float salary) {
                this.name       = name;
                this.salary     = salary;
        }

        public String getName() {
                return name;
        }

        public float getSalary() {
                return salary;
        }
}

class Boss extends Employee {

        float bonus;

        public Boss(String name, float salary) {
                super(name, salary);
                bonus = 0;
        }

        public float getBonus() {
                return bonus;
        }

        public void setBonus(float bonus) {
                this.bonus = bonus;
        }
}
```

```
public class Test {

	public static void main(String[] args) {
		Boss becali;
		List<Employee> employees = new LinkedList<Employee>();

		employees.add(new Employee("Costel", 15));
		employees.add(new Employee("Gigel", 20));
		employees.add(becali = new Boss("Becali", 1000));
		becali.setBonus(100);

	}
}
```

Presupunem ca ne intereseaza sa interogam toti angajatii nostri asupra **venitului** lor total. Observam ca:

- **anagajatii** obisnuiti au salariul ca unic venit
- **sefii** poseda, pe langa salariu, un posibil bonus

Varianta la indemana ar fi sa definim, in **fiecare** din cele doua clase, cate o metoda, `getTotalRevenue()`, care intoarce salariul pentru **angajati**, si suma dintre salariu si bonus pentru **sefi**:

```
class Employee {
	...
	public float getTotalRevenue() {
		return salary;
	}
	...
}

class Boss extends Employee {
	...
	@Override
	public float getTotalRevenue() {
		return salary + bonus;
	}
	...
}
```

Sa consideram acum ca dorim sa calulam **procentul mediu** pe care il reprezinta bonusul din venitul sefilor, luandu-se in considerare doar bonusurile pozitive. Avem doua posibilitati:

- **definim** cate o metoda, `getBonusPercentage()`, care in `Employee` intoarce mereu `0`, iar in `Boss` raportul real. **Dezavantajul** consta in **sporirea** interfetelor claselor cu functii prea specializate, de detaliu
- parcurgem lista de angajati, **testam**, la fiecare pas, tipul angajatului, folosind `instanceof`, si calculam, doar pentru sefi, raportul solicitat. **Dezavantajul** este tratarea intr-o maniera **neuniforma** a structurii noastre, cu evidentierea **particularitatilor** fiecarei clase.

Datorita acestor **particularitati** (in cazul nostru, modalitatile de calcul al venitului, respectiv procentului mediu), constatam ca este foarte utila **izolarea** implementarilor specifice ale alogritmului (in cazul nostru, scrierea unei functii in **fiecare** clasa). Acest lucru conduce, insa, la **introducerea** unei metode noi in **fiecare** din clasele antrenate in prelucrari, de **fiecare** data cand vrem sa punem la dispozitie o **noua** operatie. Obtinem urmatoarele **dezavantaje**:

- in cazul unui numar mare de operatii, **interfetele** claselor se **aglomereaza** excesiv, obscurizandu-se functionalitatea de baza a acestora
- **codul** din **interiorul** clasei, care servea functionalitatii **primare** a acesteia, poate ajunge **amestecat** cu cod necesar algoritmilor de **prelucrare**, devenind mai greu de parcurs si intretinut
- in cazul in care **nu** avem **acces** la codul claselor, singura modalitate de adaugare de functionalitate este **extinderea**.

In final, tragem concluzia ca este de dorit sa **izolam** algoritmii de clasele pe care le prelucreaza. O prima idee se refera la utilizarea metodelor **statice**. **Dezavantajul** acestora este ca **nu** pot retine, intr-un mod elegant, informatie de **stare** din timpul prelucrarii. De exemplu, daca structura noastra ar fi arborescenta (**recursiva**), in sensul ca o instanta `Boss` ar putea tine referinte la alte instante `Boss`, ce reprezinta sefii ierarhic inferiori, o functie de prelucrare ar trebui sa mentina o informatie partiala de **stare** (precum suma procentelor calculate pana intr-un anumit moment, si numarul de procente nenule, conform cerintelor de mai sus) sub forma unor **parametri** furnizati apelului recursiv:

```
class Boss extends Employee {
	...
	public float getPercentage(float sum, int n) {
		float f = bonus / getTotalRevenue();
		if (f > 0)
			return inferiorBoss.getPercentage(sum + f, n + 1); // forward query to inferior boss

		return inferiorBoss.getPercentage(sum, n);
	}
	...
}
```

O idee mai buna se dovedeste:

- **conceperea** claselor cu posibilitatea de **primire** a unor obiecte algoritm, care definesc operatiile ce se doresc realizate
- definirea unor clase **algoritm** care vor **vizita** structura noastra de date, vor efectua **prelucrarile** specifice fiecarei clase, avand, totodata, posibilitatea de **incapsulare** a unor informatii de **stare** (cum sunt suma si numarul din exemplul anterior).

After

Sa privim modificarile pe care le sufera programul nostru pentru a se alinia cu observatia de mai sus:

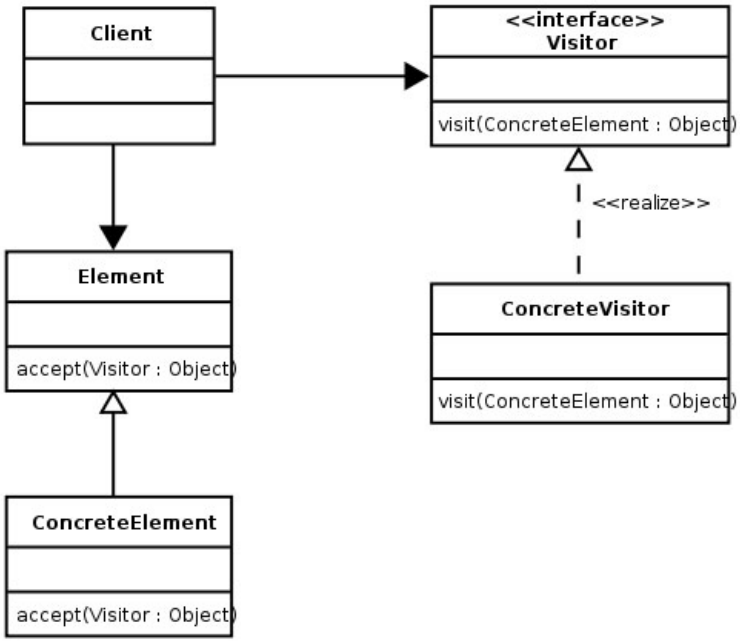
```
// (1)
interface Visitor {
        public void visit(Employee e);
        public void visit(Boss b);
}

// (2)
interface Visitable {
        public void accept(Visitor v);
}

class Employee implements Visitable {
        ...
        // (3)
        @Override
        public void accept(Visitor v) {
                v.visit(this);
        }
        ...
}

class Boss extends Employee {
        ...
        // (3')
        @Override
        public void accept(Visitor v) {
                v.visit(this);
        }
        ...
}

public class Test {

        public static void main(String[] args) {
                ...
                Visitor v = new SomeVisitor();        // init visitor with a concrete visitor class
                for (Employee e : employees)
                        e.accept(v);
                ...

        }
}
```

Secventa de mai sus defineste (urmariti numerotarea din cod):

1. o interfata, `Visitor`, ce reprezinta un **algoritm** oarecare, ce va putea **vizita** orice clasa. Observati definirea cate unei metode `visit` pentru **fiecare clasa** ce va putea fi vizitata
2. o interfata, `Visitable`, a carei metoda `accept(Visitor)` permite rularea unui **algoritm** pe structura curenta
3. implementari ale metodei `accept(Visitor)`, in cele doua clase, care, pur si simplu, solicita **vizitarea** instantei curente de catre vizitator. `Employee` implementeaza acum `Visitable`.

Diagrama de clase

Diagrama de clase pentru design pattern-ul **Visitor** este urmatoarea:



In diagrama de mai sus, putem identifica:

- `Element` - `Visitable`
- `ConcreteElement` - `Employee`, `Boss`

Aparent, folosirea lui `accept` este artificiala. De ce nu declansam vizitarea unui obiect, apeland **DIRECT** `v.visit(e)` atunci cand dorim vizitarea unui obiect oarecare? Ce se intampla insa, cand dorim sa vizitam o structura complexa de obiecte? (lista, arbore, graf etc):

- **declansarea** vizitarii se va face printr-un apel `accept` pe un prim obiect (primul element din lista, radacina arborelui etc)
- elementul **curent** este vizitat, prin apelul `v.visit(this)`
- pe langa vizitarea elementului curent, este necesar sa declansam vizitarea tuturor elementelor **accesibile** din elementul curent (spre ex: urmatorul element din lista, nodurile copil din arbore etc). Realizam acest lucru apeland `accept` pe fiecare dintre aceste elemente. Acest lucru **nu** este **obligatoriu**, depinzand de logica structurii.

Exemplu

Iata cum poate arata un **vizitator**, ce determina venitul total al fiecarui angajat, si il afiseaza:

```
class RevenueVisitor implements Visitor {

        @Override
        public void visit(Employee e) {
                System.out.println(e.getName() + " " + e.getSalary());

        }

        @Override
        public void visit(Boss b) {
                System.out.println(b.getName() + " " + (b.getSalary() + b.getBonus()));

        }

}
```

Bineinteles, variabila `v` din `main` trebuie initializata cu o instanta a clasei de mai sus. Se observa ca metoda `visit` este **supraincarcata**, pentru fiecare clasa pe care o viziteaza.

Discutie

Aplicabilitate

Pattern-ul **Visitor** este **util** cand:

- se doreste **prelucrarea** unei structuri **complexe**, ce cuprinde mai **multe** obiecte, avand tipuri **diferite**

- se doreste **definirea** de operatii **distincte** pe **aceeasi structura**, pentru a **preveni** *poluarea* interfetelor claselor implicate, cu multe detalii apartinand unor algoritmi diferiti. In acest fel, se **centralizeaza** aspectele legate de **acelasi** algoritm intr-un singur loc, dar, in acelasi timp, se **separa** detaliile ce tin de algoritmi **diferiti**. Acest lucru conduce la **simplificarea** atat a **claselor** prelucrate, cat si a **vizitatorilor**. Orice **date** specifice algoritmului rezida in **vizitator**.
- **clasele** ce se doresc prelucrate se schimba **rar**, in timp **operatiile** de prelucrare se definesc **des**. Daca clasele prelucrate apar des, atunci este necesara modificarea vizitatorilor existenti, pentru adaugarea unei metode `visit` ce stie sa prelucreze clasa respectiva. In exemplul nostru, daca extindem clasa `Employee` cu clasa `ProjectManager`, trebuie sa imbogatim interfata `Visitor` cu metoda `visit(ProjectManager)`.

Dezavantajul intrebuintarii unui **vizitator** provine din faptul ca, asemenea oricarei alte clase, nu are acces decat la membrii **publici** ai obiectului prelucrat (**nu** poate accesa membrii **privati**, care ar putea fi relevanati in conturarea starii interne a obietului). Necesitatea expunerii acestor informatii (in forma publica) ar putea conduce la **ruperea incapsularii**.

Decizia de **utilizare** a pattern-ului **Visitor** este in stransa legatura cu **stabilitatea** ierarhiilor de clase prelucrate: daca noi clase copil sunt adaugate **rar**, atunci se **poate** aplica acest pattern (intr-o maniera eficienta).

Un **iterator** poate fi privit ca un **vizitator**, care poseda **limitarea** de a nu putea gestiona decat elemente de **acelasi tip**. Un vizitator, in cazul general, **nu se supune** acestei constrangeri, putand prelucra elemente de **orice tipuri** (care nu sunt nici macar inrudite!)

Double-dispatch

Mecanismul din spatele pattern-ului **Visitor** poarta numele de **double-dispatch**. Acesta este un concept raspandit, si se refera la faptul ca **efectul** unei operatii este **determinat** de **doi** factori. Luand cazul nostru, efectul vizitarii, solicitate prin apelul `e.accept(v)`, depinde de:

- tipul elementului **vizitat**, `e` (`Employee` sau `Boss`), pe care se invoca metoda
- tipul **vizitatorului**, `v` (`RevenueVisitor`), care contine implementarile metodelor `visit`

Acest lucru contrasteaza cu un simplu apel `e.getTotalRevenue()`, pentru care efectul este hotarat doar de tipul anagajatului. Acesta este un exemplu de **single-dispatch**.

Traversarea structurii

Traversarea structurii poate fi realizata in 3 moduri:

- de catre **structura**
- in cadrul **vizitatorului**, in cazul unor parcurgeri cu o logica mai complexa
- in conjunctie cu un **iterator**, care dicteaza ordinea de vizitare

Exercitii

1. (**3p**) Construiti programul corespunzator situatiei prezentate in laborator, si rulati. Ulterior, inlaturati definitia metodei `accept`, din clasa `Boss`, si explicati rezultatul.
2. (**7p**) Adaptati ierarhia `Employee`-`Boss`, astfel incat sa obtineti o structura **arborescenta** de obiecte:
   - Fiecare `Boss` va tine referinte catre angajatii aflati sub raspunderea lui directa (ce pot fi alti sefi, sau salariati obisnuiti)
   - In radacina se va afla CEO-ul companiei (`Boss`)
   - Adaptati mecansimul de vizitare in vederea parcurgerii intregului arbore. Propagati vizitatorul de afisare a venitului.
   - Definiti celalalt vizitator mentionat in enunt, care calculeaza procentul mediu. Afisati-l!

- **Solutii**

Navigare

- Pagina principală
- Portalul comunităţii
- Discută la cafenea
- Schimbări recente
- Pagină aleatorie
- Ajutor

Caută

Du-te Caută

Trusa de unelte

- Ce se leagă aici
- Modificări corelate
- Trimite fişier
- Pagini speciale
- Versiune de tipărit
- Legătură permanentă
- Print as PDF