Visitor Pattern

Introducción y nombre

Visitor. De Comportamiento. Busca separar un algoritmo de la estructura de un objeto.

Intención

Este patrón representa una operación que se aplica a las instancias de un conjunto de clases. Dicha operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

También conocido como

Visitante.

Motivación

Si un objeto es el responsable de mantener un cierto tipo de información, entonces es lógico asignarle también la responsabilidad de realizar todas las operaciones necesarias sobre esa información. La operación se define en cada una de las clases que representan los posibles tipos sobre los que se aplica dicha operación, y por medio del polimorfismo y la vinculación dinámica se elige en tiempo de ejecución qué versión de la operación se debe ejecutar. De esta forma se evita un análisis de casos sobre el tipo del parámetro.

Solución

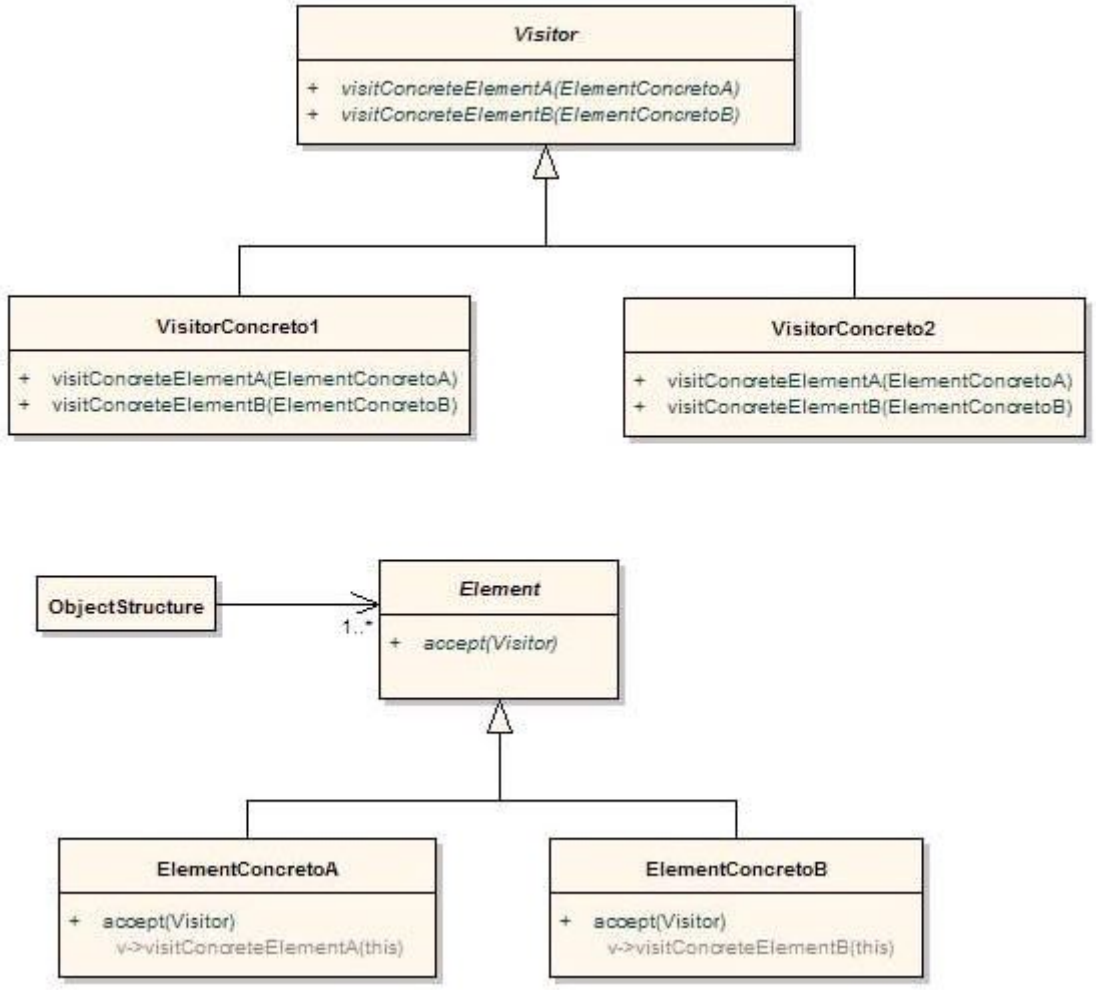
Este patrón debe utilizarse cuando:

Una estructura de objetos contiene muchas clases de objetos con distintas interfaces y se desea llevar a cabo operaciones sobre estos objetos que son distintas en cada clase concreta.

Se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases.

Las clases que definen la estructura de objetos no cambian, pero las operaciones que se llevan a cabo sobre ellas.

Diagrama UML



Participantes

Visitor: declara una operación de visita para cada uno de los elementos concretos de la estructura de objetos. Esto es, el método visit().

VisitorConcreto : implementa cada una de las operaciones declaradas por Visitor.

Element: define la operación que le permite aceptar la visita de un Visitor.

ConcreteElement: implementa el método accept() que se limita a invocar su correspondiente método del Visitor.

ObjectStructure: gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los Visitor visitar a sus elementos.

Colaboraciones

El Element ejecuta el método de visitar y se pasa a sí mismo como parámetro.

Consecuencias

Facilita la inclusión de nuevas operaciones.

Agrupa las operaciones relacionadas entre sí.

La inclusión de nuevos ElementsConcretos es una operación costosa.

Posibilita visitar distintas jerarquías de objetos u objetos no relacionados por un padre común.

Implementación

En patrón Visitor posee un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de

estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitador (visitor) como argumento. El visitador es una interfaz que tiene un método visit() diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: visitorClase1, visitorClase2... visitorClaseN. El método accept() de una clase elemento llama al método visit de su clase. Los visitadores concretos pueden entonces ser escritas para hacer una operación en particular.

Cada método visit() de un visitador concreto puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y clase elemento particular. Así el patrón visitor simula el envío doble (en inglés éste término se conoce como Double-Dispatch).

Código de muestra

En Argentina todos los productos pagan IVA. Algunos productos poseen una tasa reducida. Utilizaremos el Visitor para solucionar este problema.

[code]

```
public interface Visitable {
    public double accept(Visitor visitor);
}

public class ProductoDescuento implements Visitable {
    private double precio;

    @Override
    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
}

public class ProductoDescuento implements Visitable {
    private double precio;

    @Override
    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
}

public class IVA implements Visitor {
    private final double impuestoNormal = 1.21;
    private final double impuestoReducido = 1.105;

    @Override
    public double visit(ProductoNormal normal) {
        return normal.getPrecio() * impuestoNormal;
    }
```

```
    @Override
    public double visit(ProductoDescuento reducido) {
        return reducido.getPrecio() * impuestoReducido;
    }
}
```

Probemos como funciona este ejemplo:

```
public class Main {
    public static void main(String[] args) {
        ProductoDescuento producto1 = new ProductoDescuento();
        producto1.setPrecio(100);
        ProductoNormal producto2 = new ProductoNormal();
        producto2.setPrecio(100);

        IVA iva = new IVA();
        double resultado1 = producto1.accept(iva);
        double resultado2 = producto2.accept(iva);

        System.out.println(resultado1);
        System.out.println(resultado2);
    }
}
```

La salida por consola es:
110.5
121.0
[/code]

Cuándo utilizarlo

Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere “contaminar” a dichas clases.

Patrones relacionados

Interpreter: el visitor puede usarse para realizar la interpretación.

Composite: el visitor puede ayudar al Composite para aplicar una operación sobre la estructura del objeto definido en el composite.