The Visitor Pattern

Visitor Pattern represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Toni Sellarès
Universitat de Girona

Visitor Pattern: Motivation

Problem:

- Operations on the elements of an object structure may not apply to all objects, or apply differently to different objects.

Context:

- Object interfaces are fixed and diverse,
- Need to allow new operations, without coupling.

Solution:

- Represent operations to be performed as visitors, with the interface of every visitor representing the different kinds of objects.

Consequences:

- Can add new operations without changing objects,
- Visitors can traverse lists, trees, graphs, etc. of objects .

Visitor Pattern: Description

The Visitor pattern is a way of separating the operation from the object structure and a way of collecting together the different implementations of an operation for different kinds of elements in the object structure.

A Visitor class is created which knows how to perform a particular operation on the different kinds of elements in the object structure.

Each type of element in the structure defines an accept() method that can accept any kind of Visitor.

The Visitor is passed to each element in the structure in turn, by calling its accept() method and the Visitor then performs the operation on the visited element.

One important consequence of this separation of object structure and operation is that we can later add a new operation (a new kind of Visitor) without having to modify the element classes of the object structure.

Each type of Visitor defines several visit() methods, one for each kind of element.

The basic insight is that the precise set of instructions to execute (the function to call) depends on the run-time types of both the Visitor and the visited element.

Visitor Pattern: Definition

Visitor Pattern represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Elements can be composites (same interface) or not (different interface).

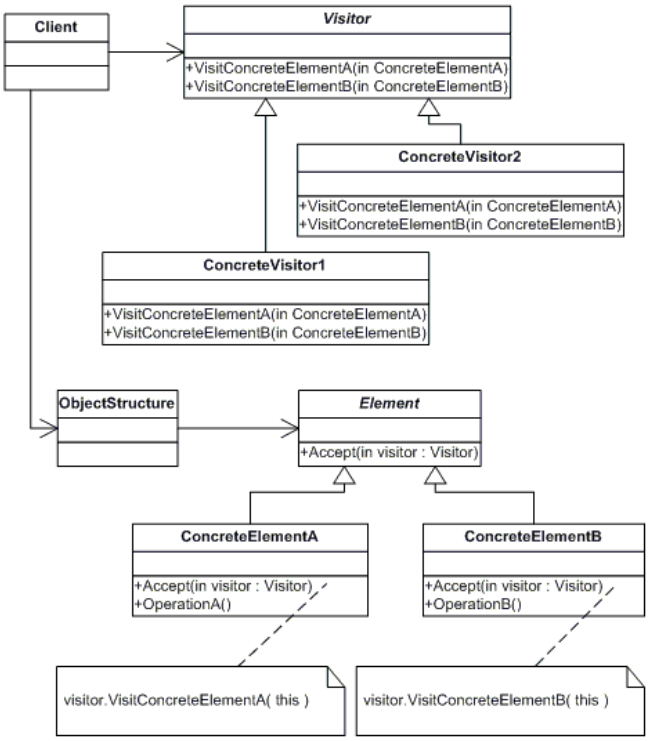
Visitor Pattern define two class hierarchies:

- One for elements being operated.
- One for visitors that define operations on the elements.

Double dispatch:

- An **accept** method in the hierarchy welcomes visitors polymorphically.
- The **accept** method calls the visitors operation also polymorphically.
- So the right combination of subject and visitor is invoked.

Create new operations by adding a new subclass to Visitor class hierarchy.



Visitor Pattern: Participants

Visitor: Declares a Visit operation for each class of ConcreteElement in the object structure.

ConcreteVisitor: Implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure.

Element: Defines an Accept operation that takes a visitor as an argument.

ConcreteElement: Implements an Accept operation that takes a visitor as an argument.

ObjectStructure:

- Can enumerate its elements,
- May provide a high-level interface to allow the visitor to visit its elements,
- May either be a Composite (pattern) or a collection such as a list or a set.

Visitor Pattern: Structural Code

```
public interface Visitor {
    void visitConcreteElementA( ConcreteElementA element );
    void visitConcreteElementB( ConcreteElementB element );
}

public interface Element {
    void accept( Visitor visitor );
}

public class ConcreteVisitor1 implements Visitor {
 void visitConcreteElementA( ConcreteElementA element ) { }
 void visitConcreteElementB( ConcreteElementB element ) { }
}

public class ConcreteVisitor2 implements Visitor {
 void visitConcreteElementA( ConcreteElementA element ) { }
 void visitConcreteElementB( ConcreteElementB element ) { }
}
```

```
public class ConcreteElementA implements Element {
        public void accept( Visitor visitor ){
            visitor.visitConcreteElementA( this );
        }
        public void operationA() {
        }
}

public class ConcreteElementB implements Element {
        public void accept( Visitor visitor ){
            visitor.visitConcreteElementB( this );
        }
        public void operationB() {
        }
}
```

Visitor Pattern: Example

Integer lists

```
interface List {}
class Nil implements List {}
class Cons implements List {
     int head;
     List tail;
}
```

What happens when we write a program which computes the sum of all components of a given List object?

First Attempt: Dedicated Methods

```
interface List {
   int sum();
}
class Nil implements List {
   public int sum() { return 0; }
}
class Cons implements List {
   int head;
   List tail;
   public int sum() {
       return head + tail.sum();
   }
}
```

Can compute the sum of all components of a given List-object l by writing l.sum().

Disadvantage: Every time we want to perform a new operation on List-objects, say, compute the product of all integer parts, then new dedicated methods have to be written for all the classes, and the classes must be recompiled.

Second Attempt: the Visitor Pattern

```
interface List {
     void accept(Visitor v);
}

class Nil implements List {
     public void accept(Visitor v) { v.visitNil(this); }
}

class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) { v.visitCons(this); }
}

interface Visitor {
  void visitNil(Nil x);
  void visitCons(Cons x);
}

class SumVisitor implements Visitor {
    int sum = 0;
    public void visitNil(Nil x) {}
    public void visitCons(Cons x){
        sum = sum + x.head;
        x.tail.accept(this); }
}

.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
.....
```

The interface Visitor declares a visit method for each of the basic classes, which must be instantiated before use.

Note that the visit methods describe both the action to be performed:

sum = sum + x.head;

and also the pattern of access:

x.tail.accept(this);

accesses the tail of the list.

The instance sv of a SumVisitor shows how to compute and print the sum of all components of a given List-object l.

The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes, provided that all objects must have an accept method.

Benefits and Drawbacks

- Benefits:
  - Allows you to add operations to a Object Structure without changing the structure itself,
  - Adding new operations is relatively easy,
  - The code for operations performed by the Visitor is centralized.
- Drawbacks when the Object Structure is a Composite:
  - The Composite classes' encapsulation is broken when the Visitor is used,
  - Because the traversal function is involved, changes to the Composite structure are more difficult.