

Visitor [GoF]

Intent

Parameterize behavior of elements of an object structure.

Motivation

Hard-coding the behavior of an object structure such as an abstract syntax tree requires re-writing the nodes' classes. Consider for example the binary node of a tree for Boolean expressions:

```
public class AndExp extends BooleanExp {
    public boolean interpret(Context ctx) { ... }
    public void dump(int level) { ... }
    // add new methods here when needed...
}
```

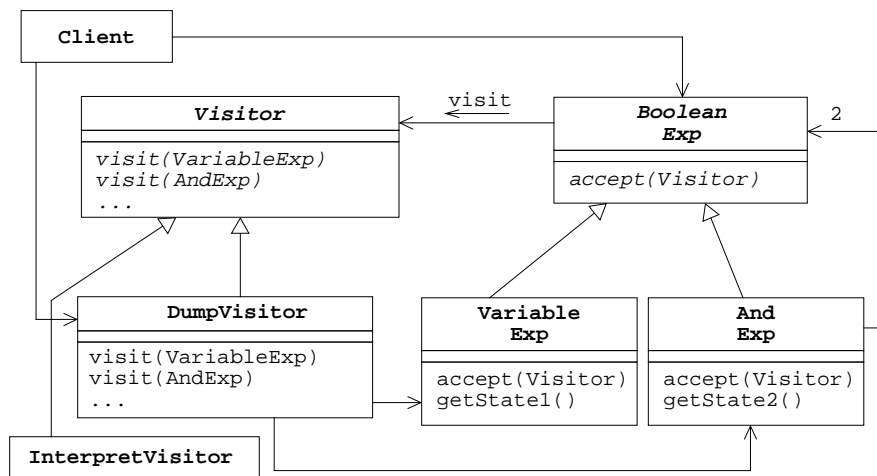
If the structure of your classes for your object structure is fairly stable, but behavior changes, then you might prefer to pull-out the behavior from the nodes of your structure. This is where the Visitor pattern comes in.

For visiting the “and” node of a tree of a Boolean expression, method `accept` of class `AndExp` looks like:

```
public class AndExp extends BooleanExp {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

For example, an in-order traversal strategy of a tree can be programmed in a specific `DumpVisitor` class:

```
public class DumpVisitor extends Visitor {
    ...
    public void visit(AndExp exp) {
        exp.getLeftExp().accept(this);
        exp.getState();
        exp.getRightExp().accept(this);
    }
}
```



A post-order traversal strategy of a tree can be programmed in a specific `DumpVisitor` class:

```
public class DumpVisitorPO extends Visitor {
    ...
    public void visit(AndExp exp) {
        exp.getLeftExp().accept(this);
        exp.getRightExp().accept(this);
        exp.getState();
    }
}
```

In the two variants above, the responsibility traversing the tree is in the visitors's visit methods.

Applicability

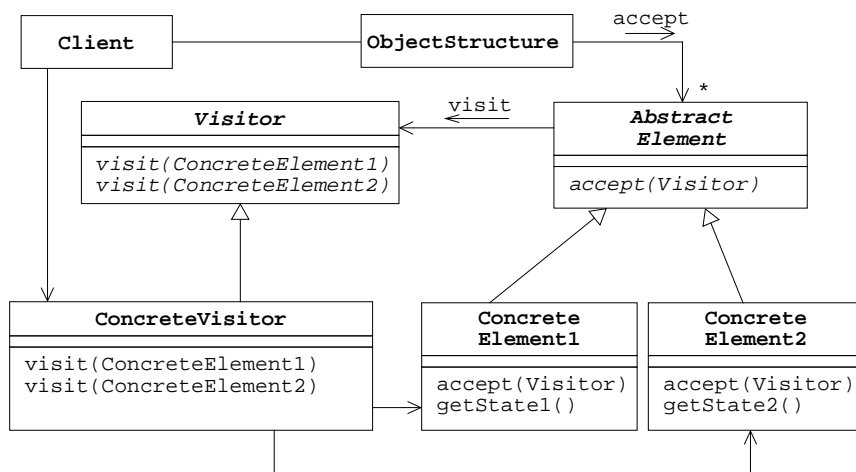
Use the Visitor when

- the object structure contains many classes, and the operations you perform depends on their concrete classes;
- you have many distinct and unrelated operations such as `interpret` and `dump`.

Participants

- Visitor:**
 - declares abstract, overloaded `visit` methods for each concrete element
- ConcreteVisitor:**
 - implements the overloaded `visit` methods
 - a concrete class is required for each terminal symbol of the grammar
- AbstractElement:**
 - defines an abstract `accept` method that takes `Visitor` as an argument
- ConcreteElement:**
 - implement the `accept` method that takes `Visitor` as an argument
- ObjectStructure:**
 - enumerates its elements
 - collection or a `Composite`

Structure

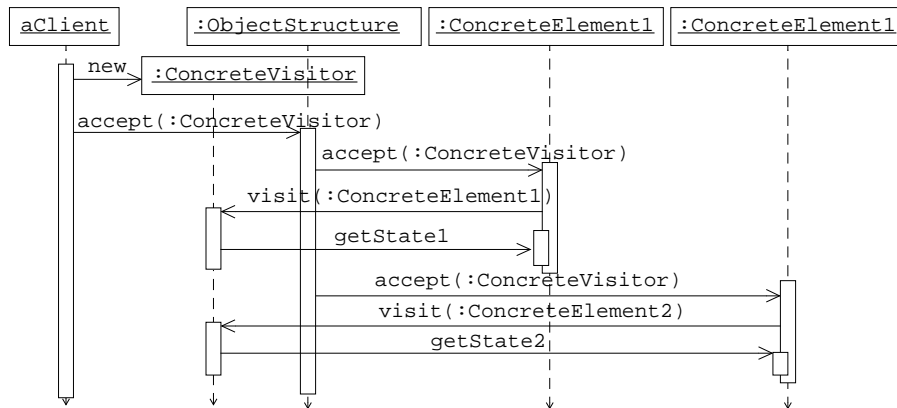


Collaborations

With using traversal responsibility at an `ObjectStructure` instance:

- First, a client creates a `ConcreteVisitor` object, and then traverses the object structure by first calling the `accept` method on the root object.
- When a `ConcreteElement` object is visited, it invokes `visit` on the `ConcreteVisitor` object that has been given as argument, and passes itself as argument (this).

Interaction between a ConcreteVisitor object, and ObjectStructure, and two ConcreteElement objects:



Implementation

- Each object structure will have its associated Visitor class. This abstract class or interface declares *for each* ConcreteElement class the corresponding visit method.
- Instead using method overloading, you can also write methods with different names.
- Traversal responsibility: The responsibility for traversing the object structure can be placed in any of three places: in the object structure, in the visitor, or in a separate iterator.

Consequences

- It is easy to add new operations on an object structure by adding a new ConcreteVisitor class.
- It is difficult to add new ConcreteElement classes. Each new class gives rise to a new abstract method visit on Visitor.
- Visitor can invoke accept on ConcreteElement object that do not necessarily have a common base class.
- Visitor can accumulate state. Without a Visitor, state would be passed as an extra argument.
- ConcreteElement classes need to offer accessor methods to allow inspection by the ConcreteVisitor.

Sample Code

Given some the classes that form the AST of a boolean expression, object structure-controlled post-order traversal can be realized as follows:

```

public abstract class BooleanExp {
    public abstract void accept(Visitor v);
}

public class AndExp extends BooleanExp {
    private BooleanExp left, right;
    public void accept(Visitor v) {
        v.visit(this);
    }
    public BooleanExp getLeftExp() {...}
    public BooleanExp getRightExp() {...}
}
  
```

```

public class Variable extends BooleanExp {
    public void accept(Visitor v) {
        v.visit(this);
    }
    public String getName() { ... }
}
// other classes omitted...

```

The abstract visitor class might look like:

```

public abstract class Visitor {
    public abstract void visit(OrExp exp);
    public abstract void visit(AndExp exp);
    public abstract void visit(NotExp exp);
    public abstract void visit(Variable exp);
    public abstract void visit(Constant exp);
}

```

Note above that an overloaded method exists for each node (ConcreteElement) in the AST. A concrete visitor class dumping the AST to the output in RPN form might look like:

Related Patterns

- Composite: Visitors can be used to traverse the composition of the Composite pattern, e.g., an AST.
- Interpreter: Visitor may be used to perform the interpretation.

```

public void class RPNVisitor extends Visitor {
    public void visit(AndExp exp) {
        exp.getLeft().accept(this);
        exp.getRight().accept(this);
        System.out.print(" *");
    }

    public void visit(OrExp exp) {
        // Left as an exercise
    }

    public void visit(NotExp exp) {
        // Left as an exercise
    }

    public void visit(Variable exp) {
        // Left as an exercise
    }
    public void visit(Constant exp) {
        // Left as an exercise
    }
}

```