The Visitor Pattern

Relevant Links

- OODesign[1]
- GoF Book[2], page 331

Notes

We examine here a very important pattern, the Visitor Pattern.

Classification

The Visitor Pattern is a Behavioral Pattern.

Intent

The Visitor Pattern is used to represent an operation to be performed on the elements of an object structure. The pattern lets you define new operations without changing the classes and elements on which it operates.

This is an example of the "open-closed" principle: Classes should be open for extension but closed for modification. The Visitor Pattern allows us to extend the behavior of our class structure by later adding new functionality.

Motivation

Imagine a "calculator" program that takes a series of arithmetic expressions and evaluates them. Each such arithmetic expression would be a tree of sorts. For instance the expression $2 + (3 * 4)$ can be represented with a node that stands for the addition, with a "left" child which is itself a "number node" with value 2, and a "right" child which is a multiplication node with children being number nodes.

So we have a tree structure made out of "nodes" of various types. We have number nodes, binary operator nodes, function nodes (log, exponential etc), maybe variable nodes etc. These will tend not to change much.

We also have various operations we might want to perform in such a structure: Evaluate it, do some type-checking in more complicated cases, print it, determine the set of variables used in it, check if any variable is used without a value assigned to it etc.

[1] http://www.oodesign.com/visitor-pattern.html

[2] http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/

Now we could put the code for these operations into the node class. But then in the future every time we want to add new operations, we would need to change all the node class implementations. This is the problem that the visitor pattern solves:

> The visitor pattern allows us to set up an interface where we can easily add operations to a fixed set of objects/classes without having to change the class implementations.
>
> On the other hand, changing the set of objects/classes by adding new types of objects is harder to do with the visitor pattern, as it requires reworking the implementation of all the visitors.
>
> In other words, the visitor pattern is good when you have a fixed set of classes of objects that need to support a varying set of operations on them.
>
> *It makes adding new operations easy.*

Participants / Implementation

Visitor The visitor interface declares a visit operation for each concrete class in the object structure (i.e. each type of node in our example).

It may also contain a generic visit operation that takes an Element as argument and calls its accept method. Typically however this belongs on the Object structure.

ConcreteVisitor Implements the specific visit operations to perform the needed task (e.g. evaluate the nodes). Each operation holds the fragment of the algorithm that is relevant for the given class structure. The ConcreteVisitor instance provides the context and local state storage for the algorithm (e.g. if variables can be "stored" in memory, this is where that would happen).

Element The interface of the elements. They need to implement an accept operation that takes an object of type visitor as input.

ConcreteElement The different objects implement the accept operation by simply calling the appropriate visit operation on the visitor.

ObjectStructure The object structure can enumerate its elements and provides some high-level interface to allow the visitor to visit its elements. May be a composite or a collection.

Here are the typical steps involved in a "visit":

- The object structure's visit method is called with a provided visitor.
- The structure starts traversing its elements. For each element, it calls its accept method, passing it the visitor.
- The element, which is from some concrete class, calls the visitor's specific "visit" method that corresponds to that class, passing itself as an argument so the visitor can use it. This is often called *double-dispatch*.

Javascript implementation A detailed example of a Javascript implementation can be seen in this file[3], which describes a small language for algebraic expressions. It contains the following:

- Number Nodes
- Variable Nodes
- Assignment Nodes (basically a way to say x=... for later use)
- Binary Operator Nodes (x+2)
- Function Nodes (ln(x))
- Sequence Nodes (allow a series of assignments for example)

Each of these nodes will support a accept function. Those all look like this:

```
VariableExp.prototype.accept = function accept(visitor) {
   return visitor.visitVariable(this);
};
FuncExp.prototype.accept = function accept(visitor) {
   return visitor.visitFunc(this);
};
...
```

So each different type of node calls a different function of the visitor. The visitor must implement all these different methods, 6 in total.

We implement three visitors: One that evaluates an expression, one that prints an expression, and one that determines if any variables are used before they have values assigned to them.

[3] ../../testPages/expressions.js