CHAPTER 1

# Reminder: the Visitor Pattern

In this chapter, we review the visitor pattern. The main purpose of the Visitor pattern is to externalize an operation from a data structure. In addition, it supports the modular definition of operation (independent from each other) and encapsulating their own data.

## 1.1 A Mini Filesystem

For example, let's consider a file system implemented with the composite pattern, where nodes can be files or directories. This composite forms a tree, where file nodes are leaf nodes and directory nodes are non-leaf nodes.

```
Object << #FileNode
  slots: { #size };
  package: 'VisitorExample'
```

```
Object << #DirectoryNode
  slots: { #children};
  package: 'VisitorExample'
```

Using this tree, we can take advantage of the Composite pattern and the polymorphism between both nodes to calculate the total size of a node implementing a polymorphic `size` method in each class.

```
FileNode >> size [
  ^ size

DirectoryNode >> size
  ^ children sum: [ :each | each size ]
```

Now, let's consider the users of the file system library want to extend it with their own operations. If the users have access to the classes, they may extend them just by adding methods to them. However, chances are users do not have access to the library classes. One way to open the library classes is to implement the visitor **protocol**: each library class will implement a generic `acceptVisitor: aVisitor` method that will perform a re-dispatch on the argument giving information about the receiver. For example, when a `FileNode` receives the `acceptVisitor:` message, it will send the argument the message `visitFileNode:`, identifying itself as a file node.

```
FileNode >> acceptVisitor: aVisitor
  ^ aVisitor visitFileNode: self

DirectoryNode >> acceptVisitor: aVisitor
  ^ aVisitor visitDirectoryNode: self
```

In this way, we can re-implement a `SizeVisitor` that calculates the total size of a node in the file system as follows. When a size visitor visits a file, it asks the file for its size. When it visits a directory, it must iterate the children and sum the sizes. However, it cannot directly ask the `size` of the children, because only `FileNode` instances do understand it but directories do not. Because of this, we need to make a recursive call and re-ask the child node to accept the visitor. Then each node will again dispath on the size visitor.

```
SizeVisitor >> visitFileNode: aFileNode
  ^ aFileNode size

SizeVisitor >> visitDirectoryNode: aDirectoryNode
  ^ aDirectoryNode children sum: [ :each | each acceptVisitor: self ]
```

## 1.2 Exercises

### Exercises on the Visitor Pattern

1. Implement mathematical expressions as a tree: define the classes `OpPlus`, `OpMinus`, and `OpNumber`.

Then model expressions like `1 + 8 / 3`, and two operations on them using the composite pattern: (a) calculate their final value, and (b) print the tree in pre-order. For example, the result of evaluating the previous expression is `3`, and printing it in pre-order yields the string `'/ + 1 8 3'`.

1. Re-implement the code above using a visitor pattern.
2. Add a new kind of node to our expressions: raised to by defining the class `OpRaiseTo`. Implement it in both the composite and visitor implementations.

3. About the difference between a composite and a visitor. What happens to each implementation if we want to add a new operation? And what happens when we want to add a new kind of node?

## 1.3 Conclusion

In this chapter we have reviewed the visitor design pattern first on a simple example, then on ASTs. The visitor design pattern allows us to extend tree-like structures with operations without modifying the original implementation.