CHAPTER 1

Understanding visitors

In a previous chapter, you built a simple mathematical expression interpreter. You were able to build an expression such as `(3 + 4) * 5` and then ask the interpreter to compute its value. In this chapter we will introduce Visitors. A Visitor is a way to represent an action on a structure (often a tree) as its own object. The action can be complex and need its own specific state. What is nice about a visitor is that it embeds its own state and behavior which would be otherwise mixed with the ones of the structure and other actions. In addition we can have multiple visitors visiting the same structure without mixing their concerns. Finally a visitor is modular because you may execute one and not another one or even load another one.

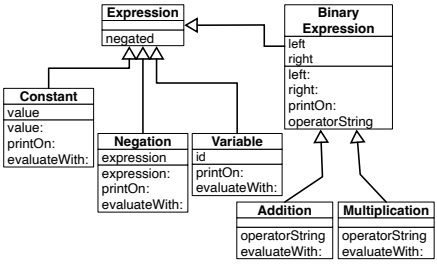

Figure 1-1 A simple hierarchy of expressions.

You will build two simple visitors that evaluate and print an expression.

Let us start with the previous situation.

1.1 Existing situation: expression trees

Figure 1-1 shows the simple hierarchy of expressions that we developed in a previous chapter. We basically have the different possible parts of an expression (variable, addition, value...) represented by their own node. Each node holds some state and in addition specifies how it computes its value. This is often done by a recursive call sending message `evaluateWith:` to subexpressions.

Note that expression trees are similar to the ones that are used to manipulate programs. For example, compilers and code refactorings as offered in Pharo and many modern IDEs are doing such manipulation with trees representing code (often called Abstract Syntax Trees).

In the rest of this chapter we will introduce step by step a visitor and we will incrementally replace the recursive calls by calls to the a visitor. Doing so we will make sure that all the tests still pass.

1.2 Visitor's key principle

The previous solution is using a simple recursive process to compute the value of an expression. Now we will define the evaluation using a visitor.

The key principle about visitor is the following one: a visitor declares to a structure that it wants to visit it (i.e., apply a treatment to it) and then the structure replies by indicating to the visitor how this visitor should visit it. This interaction is a double dispatch: it means that given a visitor and a structure, the correct method will be executed without having to explicitly test the class of the structure.

You do not have to deeply understanding this now. This interaction will emerge from the exercise.

Here is a typical illustration: The class `EConstant` defines the method `accept:` to say to the visitor that it should visit the expression using the message `visitConstant:`.

```
EConstant >> accept: aVisitor
  ^ aVisitor visitConstant: self
```

The visitor defines the specific action that he will perform:

```
EEvaluatorVisitor >> visitConstant: aConstant
  ^ aConstant value
```

Here is how the interaction starts: We ask the structure to accept a visitor.

```
| constant |
constant := EConstant value: 5.
constant accept: EEvaluatorVisitor new.
```

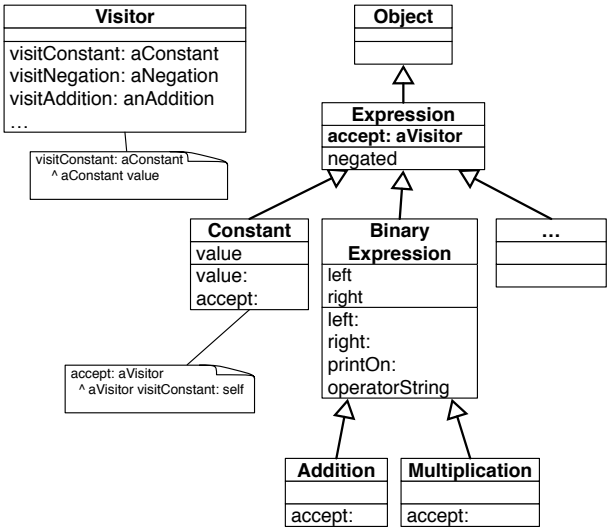

Figure 1-2 Visitor principle.

Let us step by step implement an evaluating visitor.

1.3 Introducing an evaluating Visitor

We start by adding an abstract method `accept:` in the `Expression` class to document that any expression can *welcome* a visitor and tells it how to react.

Here is the definition of the the abstract method `accept:`:

```
EExpression >> accept: aVisitor

  self subclassResponsibility
```

Now we take a concrete node expression: we start with constant expressions. When the visitor visit a constant, the constant tells the visitor that it should visit the constant as a constant. This is literaly what the following method is doing.

```
EConstant >> accept: aVisitor

  ^ aVisitor visitConstant: self
```

Defining the visitor class

Now it is time to define class representing the evaluating visitor.

```
Object subclass: #EEvaluatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

Once the class is created we can define what is it to visit a constant expression. This is simple, it is just to return the constant value. We define the `visitConstant:` as follows:

```
EEvaluatorVisitor >> visitConstant: aConstant

  ^ aConstant value
```

Adding a test class

To make sure that we control what we are doing, we add a test class.

```
TestCase subclass: #EEvaluatorVisitorTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Test'
```

We are ready to write our first test

```
EEvaluatorVisitorTest >> testVisitConstantReturnsConstantValue

  | constant result |
  constant := EConstant value: 5.
  result := constant accept: EEvaluatorVisitor new.
  self assert: result equals: 5
```

We can rewrite the old method `evaluateWith:` method to invoke the visitor.

```
EConstant >> evaluateWith: anObject

  ^ self accept: EEvaluatorVisitor new
```

You can execute your new and old tests and both should work. Note that once the visitor is in place, we will remove this method and only define it once in the superclass.

1.4 Now handling addition

We will do the same with adddition. First we define a new `accept:` method on the `Addition` class to say to the visitor which method it should execute on the structure.

```
EAddition >> accept: aVisitor

  ... Your code ...
```

Notice again that the visitor announces itself and that the addition tells it that it should be treated this time as an addition. This pattern is key to the visitor logic. You will see that we will repeat again and again. Each expression will declare how it should considered by the visitor.

Adding a new test

Now we can define a new test to validate that the execution of an addition is correct.

```
EEvaluatorVisitorTest >> testVisitAdditionReturnsAdditionResult

  | expression result |
  expression := EAddition
    left: (EConstant value: 7)
    right: (EConstant value: -2).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: 5
```

We create the accessors `left` and `right`.

```
EBinaryExpression >> left
  ^ left
```

```
EBinaryExpression >> right
  ^ right
```

Defining visitAddition:

Now we are ready to define the method `visitAddition:` so that it adds the value returned by each sub expression:

```
EEvaluatorVisitor >> visitAddition: anEAddition
  ... Your code ...
```

The method `visitAddition:` should pass the visitor to each subexpression. And once each value is known the visitor will perform the addition.

We also redefine the method `evaluateWith:` to use the visitor. As you recognize it, it is the same as in the class `EConstant`. We will remove it later.

```
EAddition >> evaluateWith: anObject
  ^ self accept: EEvaluatorVisitor new
```

Again all your new and old tests should pass.

1.5 Supporting negation

We will focus on the negation. Again we start by defining a test method.

```
EEvaluatorVisitorTest >> testVisitNegationReturnsNegatedConstant

  | expression result |
  expression := (EConstant value: 7) negated.
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: -7
```

We follow the same process. We define the `accept:` method for the negation.

```
ENegation >> accept: aVisitor
  ... Your code ...
```

We add the `expression` accessor.

```
ENegation >> expression
  ^ expression
```

Defining visitNegation:

We define the `visitNegation:` as follows:

```
EEvaluatorVisitor >> visitNegation: anENegation
  ... Your code ...
```

What you should see is that again the method `visitNegation:` is invoking the visitor on a subexpression, here the negated expression.

Again redefining evaluateWith:

We redefine the `evaluateWith:` method on a negation to invoke the visitor.

```
ENegation >> evaluateWith: anObject
  ^ self accept: EEvaluatorVisitor new
```

1.6 Supporting Multiplication

You start to get it and we will do exactly the same for multiplication.

Adding a test

```
EEvaluatorVisitorTest >>
    testVisitMultiplicationReturnsMultiplicationResult

  | expression result |
  expression := EMultiplication
    left: (EConstant value: 7)
    right: (EConstant value: -2).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: -14
```

Defining the accept: method

We define the `accept:` method on the `Multiplication` class.

```
EMultiplication >> accept: aVisitor

  ... Your code ...
```

Defining the visitMultiplication

We are not ready to define the method `visitMultiplication:` on the evaluating visitor. Its logic is similar to the one of the addition: get the value of the children and multiplying it.

```
EEvaluatorVisitor >> visitMultiplication: anEMultiplication

  ... Your report ...
```

Figure 1-3 describes the situation.

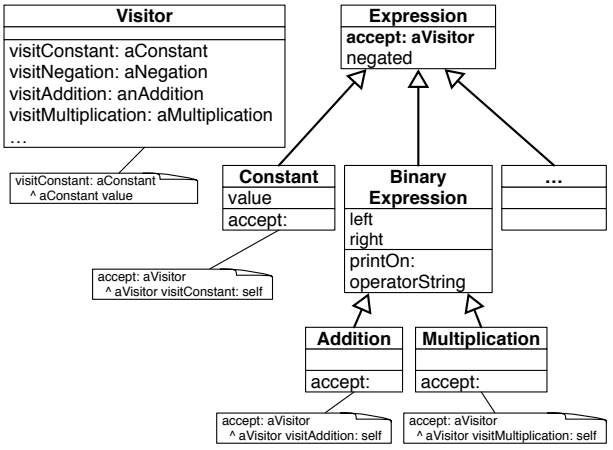

Figure 1-3 Visitor at work.

1.7 Supporting Division

As you can guess the logic is exactly the same to support division. You should start to get the pattern.

First two tests

```
EEvaluatorVisitorTest >> testVisitDivisionReturnsDivisionResult

  | expression result |
  expression := EDivision
    numerator: (EConstant value: 6)
    denominator: (EConstant value: 3).
  result := expression accept: EEvaluatorVisitor new.
  self assert: result equals: 2
```

```
EEvaluatorVisitorTest >> testVisitDivisionByZeroThrowsException

  | expression result |
  expression := EDivision
    numerator: (EConstant value: 6)
    denominator: (EConstant value: 0).
  self
    should: [expression accept: EEvaluatorVisitor new]
    raise: EZeroDenominator
```

Improving the creation API

We introduce the class message `numerator:denominator:` to ease division creation.

```
EDivision class >> numerator: aNumeratorExpression denominator:
    aDenominatorExpression

  ^ self new
    numerator: aNumeratorExpression;
    denominator: aDenominatorExpression;
    yourself
```

We define accessors so that the visitor can access to subexpression.

```
EDivision >> numerator
  ^ numerator
```

```
EDivision >> denominator
  ^ denominator
```

Defining accept:

Then we define the method `accept:` for divisions.

```
EDivision >> accept: aVisitor

  ... Your code ...
```

Defining the visitDivision:

We define the `visitDivision:` method as follows. It is similar to others. In addition here we prevent division by Zero and raise an exception instead.

```
EEvaluatorVisitor >> visitDivision: aDivision
  ... Your code ...
```

1.8 Moving up evaluateWith:

Since we get bored to always redefine the method `evaluateWith:` we define it in the superclass, the class `Expression` and we remove it from all the subclasses except `Variable` since we will still have to transform it.

```
EExpression >> evaluateWith: anObject

  ^ self accept: EEvaluatorVisitor new
```

1.9 Supporting variables

Now we can focus on supporting variable in the expression. The following test show that we can have an expression which is a variable (here named `answerToTheQuestion`) and that we can set the value of this variable using the message `at:put:`. The test then shows that when we are evaluating the expression we should get the corresponding value, (here 42).

```
EEvaluatorVisitorTest >> testVisitVariableReturnsVariableValue
  | expression result visitor |
  expression := EVariable id: #answerToTheQuestion.

  visitor := EEvaluatorVisitor new.
  visitor at: #answerToTheQuestion put: 42.

  result := expression accept: visitor.
  self assert: result equals: 42
```

Extending the visitor state

To support variable the visitor should hold a kind of environment with the value of each variable. We introduce an instance variable named `bindings`. This is a good example that shows that a visitor is the natural place to store state about the specific behavior it represents.

```
Object subclass: #EEvaluatorVisitor
  instanceVariableNames: 'bindings'
  classVariableNames: ''
  package: 'Expressions-Model'
```

We initialize this variable to a dictionary.

```
EEvaluatorVisitor >> initialize

  super initialize.
  bindings := Dictionary new
```

We define a little helper to set the value of a variable.

```
EEvaluatorVisitor >> at: anId put: aValue
  bindings at: anId put: aValue
```

We define a class method `id:` to name a variable.

```
EVariable class >> id: anId

  ^ self new id: anId; yourself
```

Visiting a variable

We have to define a method `accept:` on the class `EVariable`.

```
EVariable >> accept: aVisitor
  ... Your code ...
```

Now we are ready to define the meaning of evaluating a variable. The method `visitVariable:` of the `EEvaluatorVisitor` is responsible of this.

```
EEvaluatorVisitor >> visitVariable: aVariable

  ... Your code ...
```

1.10 Redefine evaluateWith:

We modify the method `evaluateWith:` to make sure that the initial bindings are stored in the visitor.

```
EExpression >> evaluateWith: anEnvironment

  | visitor |
  visitor := EEvaluatorVisitor new.
  visitor bindings: anEnvironment.
  ^ self accept: visitor.
```

```
EEvaluatorVisitor >> bindings: aDictionary

  bindings := aDictionary
```

1.11 A new visitor

Using a visitor is particularly interesting when we have multiple behavior that we want to encapsulate. Such behaviors are applied on a structure without mixing the state of the structure with the state of the behavior or mixing multiple behaviors together.

Now that each kind of expression is declaring in its respective methods how a visitor should visit it, other visitors can be easily expressed. And this is what we will show now.

Defining a new visitor

Now we show how we can have another visitor, an expression printer. Let us define the following class.

```
Object subclass: #EPrinterVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

Define some tests to make sure that you are getting the correct results. We let you do it.

```
TestCase subclass: #EPrinterVisitorTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Expressions-Model'
```

1.12 Visiting methods

We start by defining some typical visit methods as follows:

```
EPrinterVisitor >> visitConstant: aConstant
  ^ aConstant value asString
```

```
EPrinterVisitor >> visitMutiplication: aMultiplication

  | left right |
  left := aMultiplication left accept: self.
  right := aMultiplication right accept: self.
  ^ '(', left , ' * ', right, ')'
```

Now you should be in position to finish the implementation.

```
EPrinterVisitor >> visitAddition: anAddition
  ... Your code ...
```

```
EPrinterVisitor >> visitDivision: aDivision
  ... Your code ...
```

```
EPrinterVisitor >> visitNegation: aNegation
  ... Your code ...
```

```
EPrinterVisitor >> visitVariable: aVariable
  ... Your code ...
```

1.13 Conclusion

In this chapter we show how you can pass from a behavior inside a class hierarchy to a separate object and how once this architecture is in place (basically the `accept:` methods) other visitors can be easily expressed.

The visitor pattern is a nice design. It supports encapsulate behavior on complex structure. In addition it lets users develop their own functionality indepently of others.

Now you should pay attention not to over use it. It is also more suitable for systems whose domain does not change because else each time you add a kind of object in your composite (here the expression) you would have to touch each visitor.