CHAPTER 1

Manipulating ASTs with the Visitor Pattern

In the previous chapters we have seen how to create and manipulate AST nodes. The `Parser` class implements a parser of expressions and methods that returns AST nodes for the text given as an argument. With the AST manipulation methods we have seen before, we can already write queries on an AST. For example, counting the number of message-sends in an AST is as simple as the following loop.

```
count := 0.
aNode nodesDo: [ :n |
  n isMessage
    ifTrue: [ count := count + 1 ] ].
count
```

More complex manipulations, however, require more than an iteration and a conditional. When a different operation is required for each kind of node in the AST, potentially with special cases depending on how nodes are composed, one object-oriented alternative is to implement it using the Visitor design pattern.

The Pharo AST supports visitor: its nodes implement `acceptVisitor:` methods on each of the nodes. This means we can introduce operations on the AST that were not foreseen by the original developers. In such cases, it is up to us to implement a visitor object with the correct `visitXXX:` methods.

1.1 Introducing AST visitors: measuring the depth of the tree

To introduce how to implement an AST visitor on RBASTs, let's implement a visitor that returns the max depth of the tree. That is, a tree with a single node has a depth of 1. A node with children has a depth of 1 + the maximum depth amongst all its children. Let's call that visitor `DepthCalculatorVisitor`.

```
Object << #DepthCalculatorVisitor
  package: 'VisitorExample'
```

Pharo's AST nodes implement already the visitor pattern. They have an `acceptVisitor:` method that will dispatch to the visitor with corresponding visit methods.

This means we can already use our visitor but we will have to define some methods else it will break on a visit.

1.2 Visiting message nodes

Let's start by calculating the depth of the expression `1+1`. This expression is made of a message node, and two literal nodes.

```
expression := OCParser parseExpression: '1+1'.
expression acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand visitMessageNode:
```

If we execute the example above, we get a debugger because `DepthCalculatorVisitor` does not understand `visitMessageNode:`. We can then proceed to introduce that method in the debugger using the button **create** or by creating it in the browser. We can implement the visit method as follows, by iterating the children to calculate the maximum depth amongst the children and then adding 1 to it.

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ 1 + (aMessageNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

1.3 Visiting literal nodes

As soon as we restart the example, it will stop again with an exception again, but this time because our visitor does not know how to visit literal nodes. We know that literal nodes have no children, so we can implement the visit method as just returning one.

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
  ^ 1
```

1.4 Calculating the depth of a method

A method node contains a set of statements. Statements are either expressions or return statements. The example that follows parses a method with two statements whose maximum depth is 3. The first statement, as we have seen above, has a depth of 2. The second statement, however, has depth of three, because the receiver of the + message is a message itself. The final depth of the method is then 5: 1 for the method node, 1 for the sequence node, and 3 for the statements.

```
method := OCParser parseMethod: 'method
  1+1.
  self factorial + 2'.
method acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand visitMethodNode:
```

To calculate the above, we need to implement three other visiting methods: `visitMethodNode:`, `visitSequenceNode:` and `visitSelfNode:`. Since for the first two kind of nodes we have to iterate over all children in the same way, let's implement these similarly to our `visitMessageNode:`. Self nodes are variables, so they are leafs in our tree, and can be implemented as similarly to literals.

```
DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ 1 + (aMethodNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ 1 + (aSequenceNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ 1
```

1.5 Refactoring the implementation

This simple AST visitor does not actually require a different implementation for each of its nodes. We have seen above that we can differentiate the nodes between two kinds: leaf nodes that do not have children, and internal nodes that have children. A first refactoring to avoid the repeated code in our solution may extract the repeated methods into common ones: `visitNodeWithChildren:` and `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNodeWithChildren: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ self visitNodeWithChildren: aMessageNode


DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ self visitNodeWithChildren: aMethodNode
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNodeWithChildren: aSequenceNode
```

```
DepthCalculatorVisitor >> visitLeafNode: aSelfNode
  ^ 1
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ self visitLeafNode: aSelfNode
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitLeafNode: aLiteralValueNode
```

1.6 Second Refactoring

As a second step, we can refactor further by taking into account a simple intuition: leaf nodes do never have children. This means that `aNode children` always yields an empty collection for leaf nodes, and thus the result of the following expression is always a program that zero:

```
(aNode children
  inject: 0
  into: [ :max :node | max max: (node acceptVisitor: self) ])
```

In other words, we can reuse the implementation of `visitNodeWithChildren:` for both nodes with and without children, to get rid of the duplicated `1+`.

Let's then rename the method `visitNodeWithChildren:` into `visitNode:` and make all visit methods delegate to it. This will allow us also to remove the, now unused, `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNode: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ self visitNode: aMessageNode
```

```
DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ self visitNode: aMethodNode
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNode: aSequenceNode
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ self visitNode: aSelfNode
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitNode: aLiteralValueNode
```

1.7 Refactoring: A common Visitor superclass

If we take a look at our visitor above, we see a common structure has appeared. We have a lot of little visit methods per kind of node where we could do specific per-node treatments. For those nodes that do not do anything specific, with that node, we treat them as a more generic node with a more generic visit method. Our generic visit methods could then be moved to a common superclass named `BaseASTVisitor` defining the common structure, but making a single empty hook for the `visitNode:` method.

```
Object << #BaseASTVisitor
  package: 'VisitorExample'
```

```
BaseASTVisitor >> visitNode: aNode
  "Do nothing by default. I'm meant to be overridden by subclasses"
```

```
BaseASTVisitor >> visitMessageNode: aMessageNode
  ^ self visitNode: aMessageNode
```

```
BaseASTVisitor >> visitMethodNode: aMethodNode
  ^ self visitNode: aMethodNode
```

```
BaseASTVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNode: aSequenceNode
```

```
BaseASTVisitor >> visitVariableNode: aNode
  ^ self visitNode: aNode
```

```
BaseASTVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitNode: aLiteralValueNode
```

And our `DepthCalculatorVisitor` is then redefined as a subclass of it:

```
BaseASTVisitor << #DepthCalculatorVisitor
  package: 'VisitorExample'
```

```
DepthCalculatorVisitor >> visitNode: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

Fortunately for us, Pharo's ASTs already provide `ASTProgramNodeVisitor` a base class for our visitors, with many hooks to override in our specific subclasses.

1.8 Searching the AST for a Token

Calculating the depth of an AST is a pretty naïve example for a visitor because we do not need special treatment per node. It is however a nice example to introduce the concepts, learn some common patterns, and it furthermore forced us to do some refactorings and understand a complex visitor structure. Moreover, it was a good introduction for the `OCProgramNodeVisitor` class.

In this section, we will implement a visitor that does require a different treatment per node: a node search. Our node search will look for a node in the tree that contains a token matching a string. For the purposes of this example, we will keep it scoped to a **begins with** search, and will return all nodes it finds, in a depth-first in-order traversal. We leave as an exercise for the reader implementing variants such as fuzzy string search, traversing the AST in different order, and being able to provide a stream-like API to get only the next matching node on demand.

Let's then start to define a new visitor class `SearchVisitor`, subclass of `OCProgramNodeVisitor`. This class will have an instance variable to keep the token we are looking for. Notice that we need to keep the token as part of the state of the visitor: the visitor API implemented by Pharo's ASTs do not support additional arguments to pass around some extra state. This means that this state needs to be kept in the visitor. When the search matches the name of a variable or elements, we will store its name into a collection of matched node names.

```
OCProgramNodeVisitor << #SearchVisitor
  slots: { #token . #matchedNodeNames};
  package: 'VisitorExample'

SearchVisitor >> token: aToken
  token := aToken

SearchVisitor >> initialize
  super initialize.
  matchedNodeNames := OrderedCollection new
```

1.9 Searching in variables nodes

Let us define a little test

```
SearchVisitorTest >> testTokenInVariable

  | tree visitor |
  tree := OCParser parseMethod: 'one

| pharoVar |
pharoVar := 0.
pharoVar := pharoVar + 1.
^ pharoVar'.
```

```
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharoVar'
```

Implement the visit methods for variable nodes. A variable node matches the search if its name begins with the searched token.

```
SearchVisitor >> visitVariableNode: aNode

  (aNode name beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode name ]
```

Searching in message nodes

Message nodes will match a search if their selector begins with the searched token. In addition, to follow the specification children of the message need to be iterated in depth first in-order. This means the receiver should be iterated first, then the message node itself, and finally the arguments.

The following test checks that we identify message selectors.

```
SearchVisitorTest >> testTokenInMessage

  | tree visitor |
  tree := OCParser parseMethod: 'one

self pharo2 pharoVar: 11.
'.
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharo2'.
  self assert: visitor matchedNodeNames second equals: 'pharoVar:'
```

```
SearchVisitor >> visitMessageNode: aNode

  aNode receiver acceptVisitor: self.
  (aNode selector beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode selector ].
  aNode arguments do: [ :each | each acceptVisitor: self ]
```

1.10 Searching in literal nodes

Literal nodes contain literal objects such as strings, but also booleans or numbers. To search in them, we need to transform such values as string and then perform the search within that string.

```
SearchVisitor >> testTokenInLiteral

  | tree visitor |
  tree := OCParser parseMethod: 'one

^ #(''pharoString'')'.
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharoString'
```

```
SearchVisitor >> visitLiteralValueNode: aNode

  (aNode value asString beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode value asString ]
```

Another design would be to return the collection of matched nodes instead of storing it inside the visitor state. The visit methods should then return a collection with all matching nodes. If no matching nodes are found, an empty collection is returned.

We let you implement it as an exercise.

1.11 Exercises

Exercises on the AST Visitors

1. Implement an AST lineariser, that returns an ordered collection of all the nodes in the AST (similar to the pre-order exercise above).
2. Extend your AST lineariser to handle different linearisation orders: breadth-first, depth-first pre-order, depth-first post-order.
3. Extend the Node search exercise in the chapter to have alternative search orders. E.g., bottom-up, look not only if the strings begin with them.
4. Extend the Node search exercise in the chapter to work as a stream: asking `next` repeatedly will yield the next occurrence in the tree, or `nil` if we arrived at the end of the tree. You can use the linearisations you implemented above.

1.12 Conclusion

The visitor design pattern allows us to extend tree-like structures with operations without modifying the original implementation. The tree-like structure, in our case the AST, needs only to implement an accept-visit protocol. Opal Compiler's ASTs implement such a protocol and some handy base visitor classes.

Finally, we implemented two visitors for ASTs: a depth calculator and a node searcher. The depth calculator is a visitor that does not require special manipulation per-node, but sees all nodes through a common view. The search visitor has a common case for most nodes, and then implements special search conditions for messages, literals and variables.

In the following chapters, we will use the visitor pattern to implement AST interpreters: a program that specifies how to evaluate an AST. A normal evaluator interpreter yields the result of executing the AST. However, we will see that abstract interpreters will evaluate an AST in an abstract way, useful for code analysis.