The Visitor Pattern

Andrew P. Black

CS 410/510 Advanced Programming
Recap

• Recall the rows and columns diagram

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first: error</td>
</tr>
</tbody>
</table>

• Each row is a separate class
  ⇒ adding rows is easy

• Each column is a method in multiple classes
  ⇒ adding columns is hard
Visitor: Synopsis

- The Visitor pattern turns columns (hard to add) into rows (easy to add)
  - *i.e.*, operations are represented as *classes*, rather than as *methods*. 
Example: Arithmetic Expressions

- Represent arithmetic expressions like
  \[10 - (-4 + (5 \times -7))\]
• Class hierarchy:
  • operations like `numericValue` would normally be implemented by recursive traversal of the expression tree
  • e.g.

```plaintext
Difference ▸ numericValue
  ⊢ left numericValue – right numericValue
```

Expression
  BinaryExpression
    Difference
    Product
    Quotient
    Sum
  Primary
    Factor
      Literal
        IntegerLiteral
        RealLiteral
    Negation
Solution: turn operation into a class

1. Create *NumericEvaluator* class

   - give it methods called *visitDifference:*, *visitSum:*, that do the appropriate thing on Difference and Sum nodes, *e.g.*:

   ```ruby
   NumericEvaluator >> visitDifference: diff
     ↑ diff left numericValue –
        diff right numericValue
   ```
Solution: turn operation into a class

1. Create *NumericEvaluator* class

   • give it methods called `visitDifference:`, `visitSum:`, that do the appropriate thing on Difference and Sum nodes, e.g.:

   ```
   NumericEvaluator >> visitDifference: diff
   ↑ diff left numericValue –
   diff right numericValue
   ```

Compare:
Solution: turn operation into a class

1. Create *NumericEvaluator* class

   • give it methods called *visitDifference:*, *visitSum:*, that do the appropriate thing on Difference and Sum nodes, e.g.:

   ```ruby
   NumericEvaluator >> visitDifference: diff
   \[ \text{diff left numericValue} - \text{diff right numericValue} \]
   ```

   ```ruby
   Difference >> numericValue
   \[ \text{left numericValue} - \text{right numericValue} \]
   ```
2. Every class Foo in the hierarchy gets a method *accept: aVisitor* defined as follows:

```
Foo >> accept: aVisitor
  \aVisitor visitFoo: self
```

Note how the selector of the message tells the visitor what kind of node it is visiting.
3. At the top of the hierarchy, add a single method that provides a client interface:

Expression >> numericValue
↑ self accept: NumericEvaluator new

★ all of the code that implements numeric evaluation is now outside of the Expression classes
Let's look …
Consequences

• External code (in the visitor) must have access to the internals of the visited objects
  ➡ all significant state must be public
• Is this object-oriented?
• New operations can be added without changing the Expression classes
  • Why is this a big deal?
A very good resource... follows format of GoF book

The Design Patterns Smalltalk Companion

by

Sherman R. Alpert, Kyle Brown, Bobby Woolf

Foreword by Kent Beck

Addison-Wesley, 1998.