The Visitor Pattern

Andrew P. Black

Recap

• Recall the rows and columns diagram

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ConsList (x: i)</td>
</tr>
<tr>
<td>EmptyList (y: i)</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 (or impossible)

Visitor: Synopsis

• The Visitor pattern turns columns (hard to add) into rows (easy to add)
  • *i.e.*, it turns columns (methods) into rows (classes)
  • operations are represented as classes, rather than as methods.

Example: Arithmetic Expressions

• Represent arithmetic expressions like
  10 - (-4 + (5 * -7))

Class hierarchy:

- operations like numericValue would normally be implemented by recursive traversal of the expression tree
- *e.g.*

  - Difference `\text{visitDifference}(\text{diffNode left numericValue - diffNode 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.*

Problem: each operation (prettyPrint, typeCheck, etc) is dispersed over a dozen classes
Solution (continued)

2. Every concrete class $Foo$ in the Expression 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
- Do this for $Foo = \text{Difference, Product, Quotient, Sum, etc.}$

Let's look …

<table>
<thead>
<tr>
<th>All</th>
<th>All NumericEvaluator</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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?

Solution (continued)

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
- It's in the NumericEvaluator class

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.