Design patterns (part 3)

CSE 331
University of Washington

Michael Ernst
Outline

✓ Introduction to design patterns
✓ Creational patterns (constructing objects)
✓ Structural patterns (controlling heap layout)
⇒ Behavioral patterns (affecting object semantics)
Composite pattern

Composite permits a client to manipulate either an atomic unit or a collection of units in the same way.

Good for dealing with part-whole relationships.
Composite example: Bicycle

• Bicycle
  – Wheel
    • Skewer
      – Lever
      – Body
      – Cam
      – Rod
      – Acorn nut
    • Hub
    • Spokes
    • Nipples
    • Rim
    • Tape
    • Tube
    • Tire
  – Frame
  – Drivetrain
  – ...
Methods on components

class BicycleComponent {
    int weight();
    float cost();
}

class Skewer extends BicycleComponent {
    float price;
    float cost() { return price; }
}

class Wheel extends BicycleComponent {
    float assemblyCost;
    Skewer skewer;
    Hub hub;

    float cost() {
        return assemblyCost
            + skewer.cost()
            + hub.cost()
            + ...;
    }
}
Composite example: Libraries

Library
  Section (for a given genre)
  Shelf
  Volume
  Page
  Column
  Word
  Letter

interface Text {
  String getText();
}
class Page implements Text {
  String getText() {
    ... return the concatenation of the column texts ...
  }
}
Traversing composites

Goal: perform operations on all parts of a composite
Representing Java code

\[ x = \text{foo} \times b + c / d; \]
Abstract syntax tree (AST) for Java code

```java
class PlusOp extends Expression {  // + operation
    Expression leftExp;
    Expression rightExp;
}
class VarRef extends Expression {  // variable reference
    String varname;
}
class EqualOp extends Expression {  // equality test a==b;
    Expression lvalue;  // left-hand side; "a" in "a==b"
    Expression rvalue;  // right-hand side; "b" in "a==b"
}
class CondExpr extends Expression {  // a?b:c
    Expression condition;
    Expression thenExpr;  // value of expression if a is true
    Expression elseExpr;  // value of expression if a is false
}
```
Object model vs. type hierarchy

• AST for "a + b":

• Class hierarchy for Expression:
Perform operations on abstract syntax trees

Need to write code in each of the cells of this table:

<table>
<thead>
<tr>
<th>Operations</th>
<th>CondExpr</th>
<th>EqualOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>typecheck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question: Should we group together the code for a particular operation or the code for a particular expression?

(A separate issue: given an operation and an expression, how to select the proper piece of code?)
Interpreter and procedural patterns

Interpreter: collects code for similar objects, spreads apart code for similar operations

Makes it easy to add objects, hard to add operations

Procedural: collects code for similar operations, spreads apart code for similar objects

Makes it easy to add operations, hard to add objects

The visitor pattern is a variety of the procedural pattern

<table>
<thead>
<tr>
<th>Objects</th>
<th>CondExpr</th>
<th>EqualOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>typecheck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects</th>
<th>CondExpr</th>
<th>EqualOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>typecheck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interpreter pattern

Add a method to each class for each supported operation

class Expression {
    ...
    Type typecheck();
    String prettyPrint();
}

class EqualOp extends Expression {
    ...
    Type typecheck() { ... }
    String prettyPrint() { ... }
}

class CondExpr extends Expression {
    ...
    Type typecheck() { ... }
    String prettyPrint() { ... }
}
Procedural pattern

Create a class per operation, with a method per operand type

```java
class Typecheck {
    // typecheck "a?b:c"
    Type tcCondExpr(CondExpr e) {
        Type condType = tcExpression(e.condition); // type of "a"
        Type thenType = tcExpression(e.thenExpr); // type of "b"
        Type elseType = tcExpression(e.elseExpr); // type of "c"
        if ((condType == BoolType) && (thenType == elseType)) {
            return thenType;
        } else {
            return ErrorType;
        }
    }

    // typecheck "a==b"
    Type tcEqualOp(EqualOp e) {
        ...
    }
}
```

How to invoke the right implementation?
Definition of tcExpression
(in procedural pattern)

```java
class Typecheck {
    ...
    Type tcExpression(Expression e) {
        if (e instanceof PlusOp) {
            return tcPlusOp((PlusOp)e);
        } else if (e instanceof VarRef) {
            return tcVarRef((VarRef)e);
        } else if (e instanceof EqualOp) {
            return tcEqualOp((EqualOp)e);
        } else if (e instanceof CondExpr) {
            return tcCondExpr((CondExpr)e);
        } else ...
        ...
    }
    ...
}
```

Maintaining this code is tedious and error-prone.
The cascaded if tests are likely to run slowly.
This code must be repeated in PrettyPrint and every other operation class.
Visitor pattern: a variant of the procedural pattern

Visitor encodes a traversal of a hierarchical data structure
Nodes (objects in the hierarchy) accept visitors
Visitors visit nodes (objects)

class SomeExpression extends Expression {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

class Visitor {
    void visit(SomeExpression n) {
        perform work on n
    }
}

n.accept(v) traverses the structure rooted at n, performing v's operation on each element of the structure

What happened to all the instanceof operations?
Visitor pattern: example

Objects

class EqualsOp extends Expression {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

class PlusOp extends Expression {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}

Operations

class TypeCheckVisitor implements Visitor {
    void visit(EqualsOp e) {
        perform work on n
    }
    void visit(PlusOp e) {
        perform work on n
    }
}

class PrettyPrintVisitor implements Visitor {
    void visit(EqualsOp e) {
        perform work on n
    }
    void visit(PlusOp e) {
        perform work on n
    }
}
Sequence of calls to accept and visit

```
a.accept(v)
b.accept(v)
d.accept(v)
v.visit(d)
e.accept(v)
v.visit(e)
v.visit(b)
c.accept(v)
f.accept(v)
v.visit(f)
v.visit(c)
v.visit(a)
```

Sequence of calls to visit: d, e, b, f, c, a
Implementing visitor

• You must add definitions of `visit` and `accept`

• `visit` might count nodes, perform typechecking, etc.

• It is easy to add operations (visitors), hard to add nodes (modify each existing visitor)

• Visitors are similar to iterators: each element of the data structure is presented in turn to the `visit` method
  – Visitors have knowledge of the structure, not just the sequence
Calls to visit cannot communicate with one another

Can use an auxiliary data structure

Another solution: move more work into the visitor itself

```java
class Node {
    void accept(Visitor v) {
        v.visit(this);
    }
}
class Visitor {
    void visit(Node n) {
        for each child of this node {
            child.accept(v);
        }
        perform work on n
    }
}
```

Information flow is clearer (if visitor depends on children)

Traversal code repeated in all visitors (acceptor is extraneous)