Interlude: ASTs, Modularity, and the Visitor Pattern

Hal Perkins

Winter 2009
Modularity

- Classic slogans:
  - Do one thing well
  - Minimize coupling, maximize cohesion
  - Isolate operations/abstractions in modules
  - Hide implementation details

- OK, so where’s the typechecker module in MiniJava?
Operations on ASTs

- In a typical compiler, we may want to do these things with the AST:
  - Print a readable dump of the tree (pretty printing)
  - Do static semantic analysis
    - Type checking
    - Verify that things are declared and initialized properly
    - Etc. etc. etc. etc.
  - Perform optimizing transformations on the tree
  - Generate code from the tree, or
  - Generate another IR from the tree for further processing (often flatten to a linear IR)
Where do the Operations Go?

- Pure “object-oriented” style
  - Smart AST nodes
  - Each node knows how to perform every operation on itself

```java
public class WhileNode extends StmtNode {
    public typeCheck(…);
    public generateCode(…);
    public prettyPrint(…);
    ...
}
```

- Basically the organization in our MiniJava project
Critique

- This is nicely encapsulated – all details about a WhileNode are hidden in that class.
- But there are issues with modularity.
- What happens if we want to add a new operation?
  - Have to open up every node class.
- Furthermore, it means that the details of any particular operation (printing, type checking) are scattered across the node classes.
Modularity Issues

- Smart nodes make sense if the set of operations is relatively fixed, particularly if we expect to need flexibility to add new kinds of nodes.

- Example: graphics system
  - Operations: draw, move, iconify, highlight
  - Objects: textbox, scrollbar, canvas, menu, dialog box, plus new objects defined as the system evolves.
Modularity in a Compiler

- Abstract syntax does not change frequently over time
  - Kinds of nodes are relatively fixed
- As a compiler evolves, it is more common to modify or add operations
  - Can we modularize each operation (type check, code gen) so its components are together?
  - Can we avoid having to change node classes when we modify or add an operation?
## Two Views of Modularity

<table>
<thead>
<tr>
<th></th>
<th>Type check</th>
<th>Optimize</th>
<th>Generate x86</th>
<th>Flatten</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>exp</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>while</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>if</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Binop</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>draw</th>
<th>move</th>
<th>iconify</th>
<th>highlight</th>
<th>transmogrify</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>text</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>canvas</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>scroll</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>dialog</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

...
Visitor Pattern

- Idea: Package each operation in a separate class
  - Contains separate methods for each AST node kind
  - Examples: type check class, flatten class, print class
- Create one instance of this visitor class
  - Sometimes called a “function object”
- Include a generic “accept visitor” method in every node class
- To perform the operation, pass the “visitor object” around the AST during a traversal
  - This object contains separate methods to process each AST node type
Avoiding instanceof

- Next issue: we’d like to avoid huge if-elseif
  nests to check the node type in the visitor
  
  ```java
  void checkTypes(ASTNode p) {
    if (p instanceof WhileNode) { … }
    else if (p instanceof IfNode) { … }
    else if (p instanceof BinExp) { … } …
  }
  ```

- Solution: Include an overloaded “visit”
  method for each node type and get the node
  to call back to the correct operation for that
  node(!)
  
  - “Double dispatch”
One More Issue

- We want to be able to add new operations easily, so the nodes shouldn’t know anything specific about the actual visitor class(es)

- Solution: an abstract Visitor interface
  - AST nodes include “accept visitor” method for the interface
  - Specific operations (type check, code gen) are implementations of this interface
interface Visitor {
    // overload visit for each AST node type
    public void visit(WhileNode s);
    public void visit(IfNode s);
    public void visit(BinExp e);
    ...
}

- Aside: The result type can be whatever is convenient, doesn’t have to be void
Specific class TypeCheckVisitor

// Perform type checks on the AST
public class TypeCheckVisitor implements Visitor {
    // override operations for each node type
    public void visit(BinExp e) {
        e.exp1.accept(this); e.exp2.accept(this);
        // do additional processing on e before or after
    }
    public void visit(WhileNode s) { ... }
    public void visit(IfNode s) { ... }
    ...
Visitor Method in AST Nodes

- Add a new method to class ASTNode (base class or interface describing all AST nodes)

```java
public abstract class ASTNode {
    ...
    // accept a visit from a Visitor object v
    public abstract void accept(Visitor v);
    ...
}
```
Override Accept Method in Each Specific AST Node Class

- **Example**

```java
public class WhileNode extends StmtNode {
    ...
    // accept a visit from a Visitor object v
    public void accept(Visitor v) {
        v.visit(this); // dynamic dispatch on “this” (WhileNode)
    }
    ...
}
```

- **Key points**
  - Visitor object passed as a parameter to WhileNode
  - WhileNode calls visit, which dispatches to visit(WhileNode) automatically – i.e., the correct method for this kind of node
Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - \(\because\) May need to expose more state than we might do to otherwise
- Overall a good tradeoff – better modularity
  - (plus, the nodes are relatively simple data objects anyway)
Composite Objects

- If the node contains references to subnodes, we often visit them first (i.e., pass the visitor along in a depth-first traversal of the AST)
  
  ```java
  public class WhileNode extends StmtNode {
    Expr exp;  Stmt stmt;  // children
    ...
    // accept a visit from Visitor object v
    public void accept(Visitor v) {
      this.exp.accept(v);
      this.stmt.accept(v);
      v.visit(this);
    }
    ...
  }
  
  Other traversals can be added if needed
Visitor Actions

- A visitor function has a reference to the node it is visiting (the parameter)
  - ∴ can access subtrees via that node

- It’s also possible for the visitor object to contain local instance data, used to accumulate information during the traversal
  - Effectively “global data” shared by visit methods

```java
public class TypeCheckVisitor extends NodeVisitor {
    private <visitor local state shared by methods>;

    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <visitor local state shared by methods>;
}
```
Responsibility for the Traversal

- Possible choices
  - The node objects (as done above)
  - The visitor object (the visitor has access to the node, so it can traverse any substructure it wishes)
  - Some sort of iterator object
- In a compiler, the first choice can handle many common cases
Does it have to be this complicated?

What we’re trying to do: 2-level dispatch during generic traversal
  - First on the kind of operation (type check, print)
  - Second on the type of the node

If our language supports double-dispatch we could express this directly
  - But in Java and conventional O-O languages, only the first parameter (receiver) controls dispatch

One solution: multimethods. Research at UW, see papers by Chambers and colleagues
References

- For Visitor pattern (and many others)
  
  *Design Patterns: Elements of Reusable Object-Oriented Software*
  
  Gamma, Helm, Johnson, and Vlissides
  
  Addison-Wesley, 1995

- Good explanation of how to use visitors in compilers in Appel’s *Modern Compiler Implementation in Java*