Separation of Concerns in Compiler Development Using Aspect-Oriented
The Problem: Modularization of Compiler Phases

Compiler implementation is often an intricate task due to the complexity and interconnected nature of various stages within the compiler.
Pure Object-Oriented Design

Node
TypeAnalysis() SemanticAnalysis() CodeGeneration()

SubNode 1
TypeAnalysis() SemanticAnalysis() CodeGeneration()

SubNode 2
TypeAnalysis() SemanticAnalysis() CodeGeneration()

Problems
1. Code scatters all over the classes.
2. Hard to maintain and evolve.
The Goal: Separation of Concerns

The key challenge is to provide exceptional modularity that assists in properly separating several crosscutting concerns, which not only helps the developer to divide-and-conquer the complexity, but also improves the readability, reusability and extensibility of the compiler implementation.

Syntax analysis
Type checking
static analysis
Code generation
Popular Solution: the Visitor Pattern

![Diagram of the Visitor Pattern]

**Problems**

1. Introduce a lot of extra code.
2. Forces all concrete visitors share the same interface.
3. New semantics are always introduced by traversal of the whole tree.
4. Cannot access non-public members of a node class.
New solution: Aspect-Oriented Programming

• Aspect-Oriented Programming provides special language constructs called aspects to modularize crosscutting concerns in conventional program structures.
  – Introduction (inter-type declarations)
  – Interception (join-points)

AOP is an excellent programming technology to apply to semantic analysis

AOP: handling crosscutting concerns

Semantic phase: a concern that crosscuts various AST nodes
Aspect-Oriented Semantics Implementation

- Each concern is modularized as an aspect
  - An independent semantic pass
  - A group of action codes
- Semantic pass (i.e., a visitor)
  - Implemented as introductions of the AST classes
- Crosscutting actions applied to a group of nodes
  - Weaved into AST classes as interceptions.
Introduction

TypeAnalysisAspect
Node.TypeAnalysis()
SubNode1.TypeAnalysis()
SubNode2.TypeAnalysis()

SemanticAnalysisAspect
Node.SemanticAnalysis()
SubNode1.SemanticAnalysis()
SubNode2.SemanticAnalysis()

CodeGenerationAspect
Node.CodeGeneration()
SubNode1.CodeGeneration()
SubNode2.CodeGeneration()
Benefits (1)

- Aspect-orientation can isolate crosscutting semantic behavior in an explicit way.
  - Each semantic aspect can be freely attached to (generated) AST nodes without “polluting” the parser or AST node structure.
  - Different aspects can be selectively plugged in for different purposes at compile time.
  - Since each aspect is separated with other aspects, developers can always come back to the previous phase while developing a later phase.
Benefits ( II )

• Inter-type declaration (Introduction)
  – Defined within class scope, direct access to AST node class members (include private members)
  – Preserve the object-oriented inheritance relationship.
• Join-point model (Interception)
  – Provides flexibility to insert semantic behaviors into AST nodes or parser
  – Avoid code duplication
  – Trace facility
• Introduction + Interception
  – Tree traversal
  – Phase combination
Inter-Type Declaration Example

class UnparseVisitor extends Visitor{
    protected PrintStream out = System.out;
    public Object print(Node node){
        // ...
    }
    // Other utility routines
    // ...
    public Object visit(Node node){
        return print(node);
    }
    public Object visit(
            ASTCompilationUnit node){
        return print(node);
    }
    // same visit methods for another 83
    // nodes.
    // ...
}

aspect Unparse{
    protected static PrintStream out = System.out;
    public static void print(Node node){
        // ...
    }
    // other utility routines
    // ...
    public void Node.unparse(){
        Unparse.print(this);
    }
}
Join-Point Model Example

pointcut scopeEvaluate(): target(ScopeNode+) && call (* *.evaluate()) ;

before() : scopeEvaluate(){
    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;
}

after() : scopeEvaluate(){
    currentSymTab = (SymbolTable)symTabs.pop();
}
before() : scopeEvaluate(){
    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;
}

after() : scopeEvaluate(){
    currentSymTab = (SymbolTable)symTabs.pop();
}

// node is an instance of ScopeNode
symTabs.push(currentSymTab);
SymbolTable tmp = currentSymTab;
currentSymTab = new SymbolTable();
currentSymTab.parentScope = tmp;

node.evaluate();

currentSymTab = (SymbolTable)symTabs.pop();

...
Summary

- The major difficulty in compiler construction is separation of concerns.
- Traditional object-oriented design and the Visitor pattern cannot address the problem adequately.
- Using aspect-orientation to describe semantics supersedes both approaches without generating side-effects. Various AOP language features fulfill the computational needs of tree traversal.
- The proposed approach improves the overall modularization of the system and provides flexibility for future evolution of the compiler.
Thank you!

http://www.cis.uab.edu/wuxi/
wuxi@cis.uab.edu