A Two-Dimensional Separation of Concerns for Compiler Construction

Xiaoqing Wu, Suman Roychoudhury, Barrett R. Bryant and Jeffrey G. Gray
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
{wuxi, roychous, bryant, gray}@cis.uab.edu

Marjan Mernik
Faculty of Electrical Engineering and Computer Science
University of Maribor
2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

ABSTRACT
During language evolution, compiler construction is usually performed along two dimensions: defining new abstract syntax tree (AST) classes, or adding new operations. In order to facilitate such changes, two software design patterns (i.e., the inheritance pattern and the visitor pattern) are widely used to help modularize the language constructs. However, as each design pattern is only suitable for one dimension of extension, neither of these two patterns can independently fulfill the evolution needs during the compiler construction process. In this paper, we analyze two dimensions of concerns in compiler construction and develop a paradigm allowing compiler evolution across these two dimensions using both object-orientation and aspect-orientation. Moreover, this approach provides an ability to perform pattern transformation based on pluggable aspects. A simple implementation of an expression language and its possible extension is demonstrated using Java and AspectJ.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Compilers, Incremental compilers.*

Keywords
Separation of concerns, aspect-oriented programming, pattern transformation, compiling

1. INTRODUCTION
An important objective in software engineering is to modularize a system into several components such that a change in a design decision is isolated to one location of the system. During language evolution, a compiler designer may make numerous changes to define a new abstract syntax tree (AST) class, or add a new operation. To facilitate such changes, compiler designers usually use one of the two object-oriented design patterns, the inheritance pattern [1] and the visitor pattern [2], to improve modularity in compiler construction. It is well-known that each design pattern has its own usage conditions as well as certain limitations. As an illustration, the pure syntax-directed inheritance pattern is useful for adding new AST classes in a compiler, but is unsuitable for adding new semantic operations. In contrast, the visitor pattern provides an appropriate facility to add new semantic functionalities to the existing AST node classes, but the addition of a new AST node can prove problematic. Since these two patterns can't be used together, neither of them could fulfill the evolution needs during the compiler construction process.

Aspect-Oriented Programming (AOP) [3] provides special language constructs that modularize concerns which crosscut conventional program structures (e.g., class hierarchies of object-oriented programs). Most object-oriented design patterns are generally defined as collaborations between several objects, which characterize a special case of crosscutting known as code scattering. The applicability of AOP toward modularizing object-oriented design patterns is a topic of intense investigation [4, 5].

This paper introduces the concept of a compiler matrix, which leads to a two-dimensional approach toward compiler construction that combines object-oriented design pattern principles with aspect orientation. The contribution of the paper is a demonstration of interchangeability between these dimensions based on the technique of pluggable aspects. The approach highlights the benefits derived from the inheritance and visitor patterns, while eliminating their limitations.

The next section introduces the compiler matrix and provides a high-level overview of pattern transformation. Section 3 illustrates the significance of this approach by describing it with a simple expression language. Details pertaining to the implementation of this approach are discussed in Section 4. Section 5 cites related work, followed by a conclusion that presents future work to broaden this idea to other design pattern transformations.

2. THE COMPILER MATRIX AND PATTERN TRANSFORMATION
In this section, the essence of the compiler matrix is explained by exploring the usage and limitations of the inheritance and visitor patterns. The section also presents how AOP techniques can help make these two patterns interchangeable.

2.1 Inheritance Pattern
A straightforward way to build a compiler in an object-oriented approach is to use the inheritance pattern [1] (a variation of the interpreter pattern [2]), which can be characterized as follows:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.
Copyright 2005 ACM 1-58113-964-0/05/0003…$5.00.

1. Declare an abstract super class for all AST nodes, and declare virtual methods for node operations (e.g., type checking, code generation, pretty printing) inside the class.
2. Define a class for each AST node and implement the methods that inherit from the super node class.

As an example, a partial node class hierarchy is shown in Figure 1 (a). An advantage of this approach is that during the grammar or language evolution stage, any new symbols that are added to the base grammar can be incorporated without extensive changes to the existing class hierarchy.

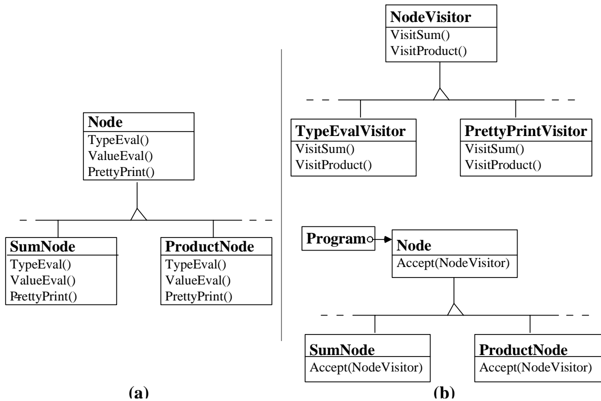

Figure 1. Inheritance pattern vs. visitor pattern

However, an inherent problem in this approach is that the semantic operations (defined as methods within each node class) crosscut the various other class boundaries, thereby leading to a system that is hard to comprehend and maintain. Moreover, adding a new operation requires an invasive change throughout the existing class hierarchy. It would be better if each new semantic function could be added separately, and the node classes were independent of the operations that apply to them.

2.2 Visitor Pattern

To resolve the problem that occurs with the inheritance pattern, recent compiler construction techniques have shown the benefits of using the visitor pattern. In the visitor pattern, all the methods pertaining to one operation of the nodes are encapsulated into a single visitor class, which is independent of other node classes and can be freely added or deleted from the system.

2.2.1 Object-Oriented Implementation

The conventional implementation of the visitor pattern uses object-oriented principles as shown in Figure 1 (b). The semantic operations on each AST node are created by invoking iteratively the accept method (i.e., the visitor function) within every node class throughout the class hierarchy. This approach is widely accepted in some compiler generators such as JavaCC[1] (used together with JJTree[2]).

However, since object-orientation describes a system by a collection of objects rather than a collection of operations, it is clear that object-orientation is not a natural specification of programs based on the visitor pattern. The complicated implementation of this design pattern makes the code hard to understand and maintain [5].

2.2.2 Aspect-Oriented Implementation

Current research in AOP and design patterns has indicated that the visitor has basic AOP characteristics: without it the structure and behavior characteristics are scattered throughout the code base instead of being isolated in single separate classes. Aspect-orientation when applied to the visitor pattern can isolate crosscutting behavior in a more explicit way [5]. By applying AOP concepts, the visitor operations can be written such that the accept methods defined in each separated node class are no longer needed (as will be shown in Section 4).

However, an important observation is that, since each operation crosscuts each visitor class, adding a new node to the existing class hierarchy will cause an invasive change to all of the visitor classes resulting in a maintenance nightmare. Therefore, no matter if it is a pure object-oriented or aspect-oriented implementation, the visitor pattern is applicable only under conditions when the node structure is static and does not change frequently. However, this requirement is hard to satisfy during language evolution because there is always a need to modify the grammar or extend the grammar during early language design stages.

2.3 Compiler Matrix

The inheritance and visitor patterns have clear benefits and limitations. The inheritance pattern assists in flexibly adding new AST nodes, but is unsuitable for adding semantic actions; the visitor pattern is useful for adding semantic operations, but inappropriate for adding new AST nodes. Therefore, the ideal solution is to synergistically combine the usefulness of both design patterns while addressing their limitations.

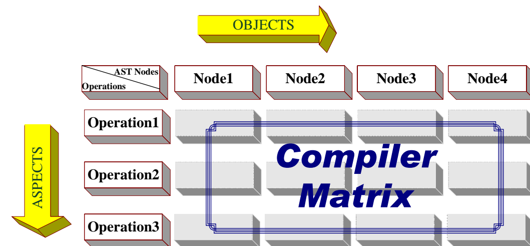

Figure 2. The compiler matrix

As reflected by the above two patterns, the abstraction of all the necessary constructs in the compiler design phase can be considered as a compiler matrix (as in Figure 2). The columns of the matrix represent the nodes of an AST, and the rows represent the operations on the AST. Each node has several operations to be performed and each operation crosscuts every other node. From an orientation point of view, each column represents an object and each row represents an aspect. If all of the artifacts are modularized vertically, an instance of the inheritance pattern emerges, which could be easily realized using a pure object-oriented approach. Correspondingly, if the matrix artifacts are modularized horizontally (row-major), an instance of the visitor

[1]JavaCC: Java Compiler Compiler, http://javacc.dev.java.net/

[2] Introduction to JJTree. http://www.j-paine.org/jjtree.html

pattern emerges, which can be ideally implemented using an aspect-oriented approach.

2.4 Pattern Transformation Approach

The compiler matrix expresses the essence of compiler construction, which reflects that an ideal solution to compiler design should provide two-dimensional separation of concerns [6] and a facility to make the two dimensions transferable. As a result, the pattern transformation based compiler construction approach is developed in which the inheritance pattern is implemented using pure Java and the visitor pattern is implemented using Java and AspectJ [7].

Since each pattern has the same set of semantic operations as in the compiler matrix and they both use Java code in method implementation, the transformation between two patterns can be achieved simply by relocation of all the methods, i.e. from AspectJ aspects to Java classes (aspect weaving) or from Java classes to AspectJ aspects (aspect unweaving). The whole compiler construction process is outlined by the following steps (illustrated in Figure 3):

1. Initially generate the parser and AST nodes in the form of Java classes or interfaces. The super node is generated as an interface to be implemented by all the AST node classes.
2. Add the semantic operations on nodes using the visitor pattern based on the AOP approach. The abstract methods added on the super node ensure that all the Java classes implement all of the required semantic operations (the checking happens at compile time).
3. If the grammar needs to be extended later, transform the visitor pattern to the inheritance pattern by weaving the operations in each aspect into the corresponding class node. After weaving, since no more aspects are needed, all the aspect specifications are removed automatically (the transformation result is shown in the lower part of Figure 3), and add the new AST nodes using the inheritance pattern.
4. If more operations need to be added, transform the inheritance pattern to the visitor pattern by unweaving the operation methods of each class node into individual aspect specifications (the transformation result is shown in the upper part of Figure 3), and add the semantic operations on nodes using the visitor pattern.

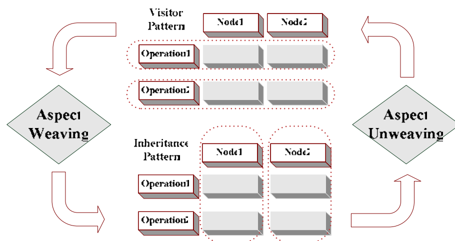

Figure 3. Pattern transformation overview

3. CASE STUDY

To illustrate the usage of the matrix construction process and the benefits of using pattern transformation, consider a simple language for arithmetic expression calculation. The grammar for the expression language is provided in Listing 1.

```
1   expression ::= term | binary_expression
2   binary_expression ::= sum | difference
3   sum ::= expression '+' expression
4   difference ::= expression '-' expression
5   term ::= integer_literal | real_literal
```

Listing 1. Context-free grammar for the expression language

```
1   class Sum implements Binary_expression, ASTNode{
2     public Expression expression1;
3     public Expression expression2;
4     public Sum (Expression expression1 , Expression expression2){
5       this.expression1 = expression1;
6       this.expression2 = expression2;
7     };
8   }
```

Listing 2. Java class for grammar symbol Sum

```
1   aspect ValueEval {
2     public abstract Double ASTNode.valueEval();
3     public Double Real_literal.valueEval(){
4       return Double.valueOf(lexeme);
5     };
6     public Double Integer_literal.valueEval(){
7       return Double.valueOf(lexeme);
8     };
9     public Double Difference.valueEval(){
10      Double value1 = expression1.valueEval();
11      Double value2 = expression2.valueEval();
12      return new Double
13         (value1.doubleValue()-value2.doubleValue());
14    };
15    public Double Sum.valueEval(){
16      Double value1 = expression1.valueEval();
17      Double value2 = expression2.valueEval();
18      return new Double
19         (value1.doubleValue()+value2.doubleValue());
20    };
21  }
```

Listing 3. AspectJ specification for value evaluation

The compiler matrix for this particular language is composed of 7 nodes (2 terminals and 5 nonterminals) and 3 operations that can be applied to those nodes. While generating the appropriate Java classes, some nodes will be generated as interfaces (including the super node named *ASTNode*) and others as concrete classes (the generation mechanism and associated rules are specified in Section 4). For example, the node class for the grammar symbol *Sum* is shown in Listing 2.

Part 1 of Figure 4 (upper-left) shows the initial state with generated nodes that contain no method definitions. Due to space limitations, only 4 of the 7 nodes are presented. In the second stage, the visitor pattern is applied such that 3 aspects are added to implement each semantic operation. The matrix is modularized horizontally as shown in part 2 of Figure 4. Because each aspect is cleanly separated from the generated node classes, there is no single manual change required inside the node classes. An AspectJ implementation for the arithmetic evaluator for this language is shown in Listing 3. Since *Expression*,

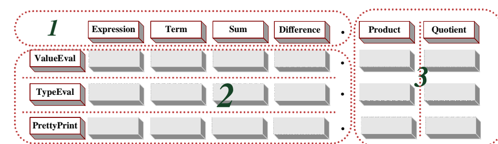

Figure 4. Two-dimensional extension for the expression language

Binary_expression and *Term* are interfaces extended from *ASTNode*, there is no need to define their abstract methods explicitly in the aspect.

Suppose that the expression language needs to be extended to handle quotient, product, as well as unary and parenthesized expressions. The grammar is extended as shown in Listing 4, with the new productions and symbols highlighted in bold.

```
1   expression ::= term | binary_expression | unary_expression
2   binary-expression ::= sum | difference | Quotient | Product
3   sum ::= expression '+' expression
4   difference ::= expression '-' expression
5   quotient ::= expression '/' expression
6   product ::= expression '*' expression
7   unary_expression ::= '-' term
8   term ::= integer_literal | real_literal | parenthesized_expression
9   parenthesized_expression ::= '('expression ')'
```

Listing 4. Extended grammar for expression language

In order to modularize the compiler matrix in a vertical way to facilitate adding new AST nodes, each semantic operation of a specific aspect is weaved into the class it belongs to and implements an instance of the inheritance pattern. For example, after the weaving process, the new node class for the grammar symbol *Sum* is shown in Listing 5, where the code in bold represents methods weaved from aspects. To extend the grammar, AST node classes are generated for the new tokens and semantic operations are manually added to each new node without changing the existing class structure. As in part 3 of Figure 4, these new AST node classes are written in the same format as the existing ones. The super interface *ASTNode* ensures that all the new nodes encapsulate all of the required methods.

```
1   class Sum implements Binary_expression, ASTNode{
2     public Expression expression1;
3      public Expression expression2;
4      public Sum (Expression expression1 , Expression expression2){
5         this.expression1 = expression1;
6         this.expression2 = expression2;
7      };
8      public Double valueEval() {
9         Double value1 = expression1.valueEval();
10        Double value2 = expression2.valueEval();
11        return new Double(value1.doubleValue()+value2.doubleValue());
12     }
13     //pretty print method
14     //type checking method
15  }
```

Listing 5. Node class in inheritance pattern

Once more, if any new operation needs to be added, the developer can transform the inheritance pattern to the visitor pattern and implement the corresponding changes. Such extensions do not change existing code.

4. IMPLEMENTATION

The generation framework for the parser and AST node classes is illustrated in Figure 5. A variation of the object-oriented Two-Level Grammar (TLG) [8] is used as the input specification for the language grammar. JLex[3] and CUP[4] are utilized as lexical and parser generation components of the framework. All the generated lexer, parser, and AST node classes are in Java. The TLG specification requires all of the productions that share the same left-hand-side (LHS) variable to be unit-productions [9]. Consequently, for those LHS variables, the framework will generate the abstract Java interfaces. This technique improves the speed of parsing and also facilitates join-point abstraction. The whole generation process is detailed in [10].

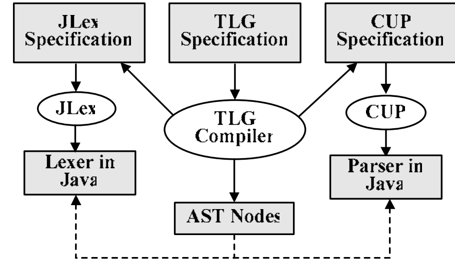

Figure 5. Parser and AST nodes generation framework

AspectJ is used for the aspect specification in this compiler construction approach. The Java-based syntax of AspectJ enables each operation method to be pluggable between Java classes and AspectJ aspects. For instance, the pretty print AspectJ specification is shown in Listing 6, which can be weaved easily into Java classes.

```
1    aspect PrettyPrint {
2      public abstract String ASTNode.prettyPrint();
3      public String Real_literal.prettyPrint(){ return lexeme; };
4      public String Integer_literal.prettyPrint(){ return lexeme; };
5      public String Difference.prettyPrint(){
6        return expression1.prettyPrint() +" - " + expression2.prettyPrint();
7      }
8      public String Sum.prettyPrint(){
9         return expression1.prettyPrint() +" + "+ expression2.prettyPrint();
10     }
11   }
```

Listing 6. AspectJ specification for pretty print

Moreover, the abstraction utilities provided by Java also can be used in AspectJ. For example, since all of the interfaces (such as *Expression*, *Binary_expression*, *Term*) only have abstract methods and they all inherit from super interface *ASTNode* (this is true for all visitor functions), there is no need to define explicitly these abstract methods. Only the abstract method of *ASTNode* and concrete methods for Java classes are defined in aspects, as illustrated in Listing 3 and Listing 6. Consider a grammar that has 100 tokens, with 30 interface tokens - this kind of abstraction will greatly reduce the overall aspect specification.

However, aspect weaving in AspectJ occurs at the byte code level without the availability of the transformed Java source code. Moreover, there is no way in AspectJ to unweave the semantic operations from Java classes (e.g. Lines 8-12 of Listing 5) to their corresponding aspects (e.g. Lines 15-20 of Listing 3). To overcome these constraints, a program transformation system was used (i.e., the Design Maintenance System (DMS) [11]) to perform source-to-source transformation in an aspect-oriented style: The inputs given to the DMS engine during the weaving process are the aspect specifications (Listings 3 and 6) and the set of node classes (Listing 2). An aspect parser is used to parse the aspect specification and a pattern matcher binds the operations' class names in the aspect specification to those AST classes. Once

[3] JLex: http://www.cs.princeton.edu/~appel/modern/java/JLex/

[4] CUP: http://www.cs.princeton.edu/~appel/modern/java/CUP/

matches are found, the related advice (semantic operations) is weaved to the corresponding node classes using the transformation system. On the other hand, unweaving is achieved using a pattern extractor that extracts the semantic operations from the existing node classes and generates several aspect specifications for each type of operation (e.g., type-checking, pretty-printing). Moreover, these operations are also removed from the existing node classes leaving them in an isolated state (free from crosscutting concerns).

5. RELATED WORK

Several papers have mentioned the use of AOP as an approach in design pattern implementation. Hannemann and Kiczales [4] use Java and AspectJ to implement all 23 design patterns in [2] and illustrate implementation details using the observer pattern as an example. Hachani and Bardou [5] further emphasize implementation of the visitor pattern using AspectJ. The benefits of using aspect-oriented techniques are described in both of these works. However, a major drawback of the visitor pattern still remains in the result implementation.

Aspect-oriented compiler design was first proposed by de Moor et al. [12], where a functional approach to attribute grammar (encapsulated as aspects) was presented. These aspects represent the semantic attributes such as the environment, which is a slight deviation from the general notion and terminology of aspects as described in this paper. The drawbacks of the inheritance and visitor patterns are discussed in [1] and the author has claimed that TreeCC can be a better alternative to both these patterns. However, the essence of TreeCC is still aspect-oriented visitors with strongly typed properties. It can't solve the major problem associated with the visitor pattern when new nodes are added to an existing node structure.

Tarr et al. first introduced the concept of multi-dimensional separation of concerns [6]. They proposed a model that allows developers to decompose the problem domain into a variety of hyperslices, which are intended to encapsulate concerns in dimensions other than the dominant one.

Our contribution differs from the above approaches in that we not only use design patterns and aspect-oriented techniques to modularize the compiler construction process and isolate the crosscutting concerns, but also make patterns and concerns interchangeable to adapt to the various development needs.

6. CONCLUSION AND FUTURE WORK

This paper analyzed the essence of the compiler matrix and presented an approach for compiler construction in two dimensions using object-orientation and aspect-orientation. The implementation of a simple expression language and its possible extension was shown using Java and AspectJ. More complex programming language designs based on this approach are under investigation. To face the challenges arising from large legacy languages, several practical programming problems will be considered. For example, one operation of a single node may not be well captured by one visitor function, and sometimes a global environment is needed for describing specific semantic actions.

There will always be multi-dimensional needs in software development, because no single design principle or pattern offers a panacea toward addressing problems of change evolution. Transformation techniques applied to design patterns offer an alternative to alleviating this problem. The inheritance pattern and visitor pattern transformation technique introduced in this paper may not be only used in compiler design, but also in other software system development. Moreover, the technique offers an initial example toward the possibilities of pattern transformation (especially between inheritance pattern and behavior patterns such as the Iterator pattern and the Observer pattern).

Due to space restrictions, several implementation details are omitted in this paper. Interested readers may refer to the source code at the project web site (http://www.cis.uab.edu/softcom/cde) for more implementation details.

7. REFERENCES

[1] R. Weatherley. TreeCC: An Aspect-Oriented Approach to Writing Compilers. http://www.southern-storm.com.au/treecc.html.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. 11th European Conf. Object-Oriented Programming (ECOOP)*, Springer-Verlag, LNCS 1241, 1997, pp. 220-242.

[4] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 2002, pp. 161–173.

[5] O. Hachani and D. Bardou. Using Aspect-Oriented Programming for Design Patterns Implementation. In *Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002.

[6] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int. Conf. Software Engineering (ICSE)*, 1999, pp. 107-119.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP),* Springer-Verlag, LNCS 2072, 2001, pp. 327–355.

[8] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In *Proc. 35th Hawaii Int. Conf. System Sciences (HICSS)*, 2002.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[10] X. Wu, B. R. Bryant, and M. Mernik. Object-Oriented Pattern-Based Language Implementation. Technical Report, http://www.cis.uab.edu/wuxi/paper/acta.pdf

[11] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformation for Practical Scalable Software Evolution. In *Proc. Int. Conf. Software Engineering (ICSE)*, 2004, pp. 625-634.

[12] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented Compilers. In *Proc. Generative and Component-Based Software Engineering (GCSE),* Springer-Verlag, LNCS 1799, 2000, pp. 121–133