Separation of Concerns in Compiler Development Using Aspect-Orientation

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

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

ABSTRACT
A major difficulty in compiler development regards the proper modularization of concerns among the various compiler phases. The traditional object-oriented development paradigm has difficulty in providing an optimal solution towards modularizing the analysis phases of compiler development, because implementation of each phase often crosscuts the class hierarchy defined by language syntax constructs. Object-oriented design patterns, such as the Visitor pattern, also cannot solve the crosscutting problem adequately because an object is not a natural representation of a collection of operations. This paper demonstrates the benefits of applying aspect-oriented programming languages (e.g., AspectJ) and principles to compiler design and implementation. The experience result shows that the various language constructs in AspectJ (e.g., inter-type declaration, pointcut-advice model, static aspect members and aspect inheritance) fit well with the various computation needs of compiler development, which results in a compiler implementation with improved modularity and better separation of concerns. The ideas utilized in this paper can also be generalized to other software systems with a tree-like structure.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *patterns, information hiding, and languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features *–patterns, classes and objects*

General Terms
Design, Languages.

Keywords
Compiler design, design patterns, aspect-oriented programming.

1. INTRODUCTION

SAC'06, April, 23-27, 2006, Dijon, France.

Compiler implementation is often an intricate task due to the complexity and interconnected nature of various stages within the compiler. 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 [1].

It is well-known that compiler development is a multi-stage process (e.g., syntax analysis, tree construction, static checking, and code generation) [2]. Each phase has a clear goal and some of them often require an independent traversal of the Abstract Syntax Tree (AST). Language implementation using traditional tools tangles these different phases of the implementation together as one module. There are few mechanisms available to separate each phase cleanly, which contributes to the difficulty in hiding the data and methods that are only relevant to specific phases; as such, the phases of a compiler implementation are often tightly coupled. The compiler writer is forced to consider all semantic phases simultaneously and the construction of one phase always "pollutes" a different phase, which makes a system hard to develop, maintain and extend. Although recently the Visitor pattern [3] has provided a way to solve this problem, its unnatural implementation and restrictions make it difficult to adopt [4], as detailed in section 2.

Aspect-Oriented Programming (AOP) [5] provides special language constructs that modularize concerns which crosscut conventional program structures, in particular the class hierarchies of object-oriented programs and design patterns [6]. Because semantic analysis in compiler design often involves traversal of various AST nodes, it is natural to explore the effect of AOP on compiler construction [7]. Particularly, in object-oriented compiler implementation, after the node types are implemented as classes, the AOP approach is an excellent technique to apply to semantic analysis. This paper describes an approach to aspect-oriented compiler implementation by applying AspectJ. Our experience results show that AOP can significantly improve the separation of concerns in compiler construction. Various language features provided by AspectJ (e.g., pointcuts, advice, inter-type declarations and aspect methods/fields) are well-suited to describe the various analysis needs in compiler design, which cannot be easily achieved by classical object-oriented design.

This paper is organized as follows. Section 2 describes the problems in object-oriented compiler design and briefly introduces how aspect-oriented development can provide better modularization. In section 3, a case study is introduced to show how aspect-orientation offers an improvement to object-orientation and the Visitor pattern. Section 4 details the aspect-oriented methodology by exploring the essential features in AOP. Section 5 discusses related work and we conclude in section 6.

2. OBJECT-ORIENTED VS. ASPECT-ORIENTED COMPILER DESIGN

In classical object-oriented development, AST nodes are usually defined as classes with various semantic operations (e.g., type-checking, symbol table loading, pretty printing, code generation) embedded as methods. Consequently, AST traversal is achieved by iteratively executing those semantic operations. However, an inherent problem in this approach is that each kind of semantic operation (defined as a method within each node class) crosscuts the various node class boundaries, thereby leading to a system that is hard to maintain and add new operations. This problem actually reflects the drawback of object-oriented programming in modularizing crosscutting concerns. It would be better if each semantic function could be specified separately, so that the node classes were independent of the operations that apply to them.

The Visitor pattern [3] is a current compiler construction technique that is often used to address the problems introduced beforehand. In applying this pattern, all the methods pertaining to one semantic pass are encapsulated inside a visitor class. Each AST node class has a general `accept` method associated with it. An abstract visitor is declared such that the general `accept` methods can pass an arbitrary visitor and dispatch to the appropriate method in the provided visitor object. The upper part of Figure 1 provides an illustration of the Visitor pattern in implementing the type-evaluation and pretty-print analysis phases for an expression language.

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 introduces a lot of extra code in the element classes and makes the code hard to understand and maintain [4]. Particularly, if a Visitor pattern has not been incorporated into the code from the beginning, the whole AST hierarchy has to be modified to implement it. To support the double-dispatch mechanism, the abstract visitor forces the return types, number of parameters and parameter types of various visiting methods of a certain node to be the same. This is a very inflexible limitation, because different semantic operations have different computing needs. This tends to make the programs difficult to understand and introduces dependencies that can impede evolution of the compiler.

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 semantic implementation can isolate crosscutting behavior in a more explicit way. Each semantic phase can be implemented as an individual aspect. Inside each aspect, the concrete semantic actions of specific nodes can be implemented as Inter-Type Declarations (ITDs) of AST classes, from which the tree iteration logic can be specified as join points, whereas those global fields and utility routines are represented as static aspect members. Aspect-oriented semantic implementation is superior to the Visitor pattern because of its unrestricted parameters and return types for different semantic operations, with additional benefits coming from the ability to introduce any field and method into a class, as well as increased flexibility in phase integration and tree iteration using pointcut-advice models. As all the semantic actions are introduced to the AST node classes, there is only one class hierarchy created. Therefore, during tree iteration, object-oriented polymorphism is sufficient to determine which node's processing method should be invoked at runtime. Explicit double-dispatch is removed and the `accept` methods defined in each node class are no longer needed, which makes the AST nodes totally oblivious to semantic operations. A visual comparison between operation redirection in the Visitor pattern and the aspect weaving in AOP implementation is illustrated in Figure 1. Section 3 will further compare these two methods with a real example.

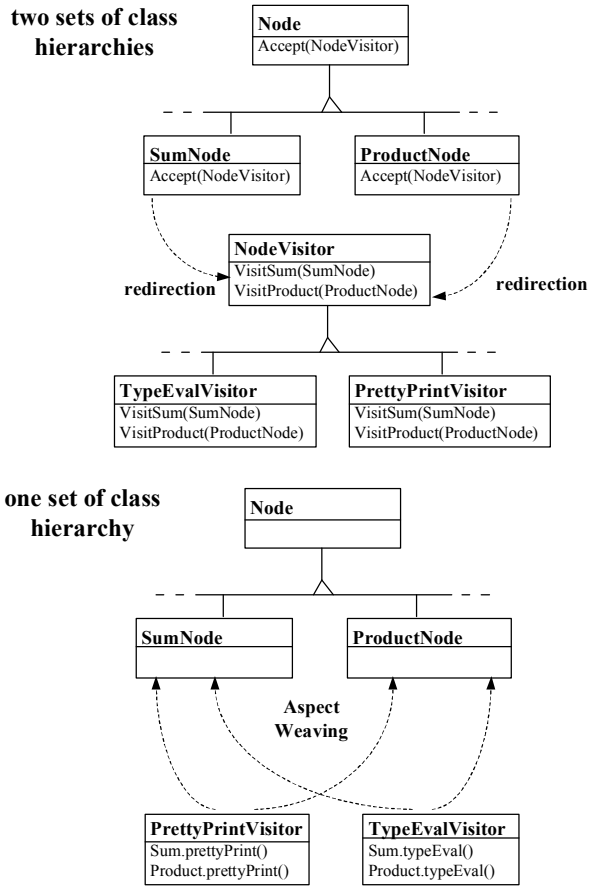

Figure 1. Operation redirection in Visitor pattern vs. aspect weaving in AOP implementation

3. CASE STUDY

In order to detail the benefits derived from applying AOP to compiler implementation, this section presents one of our previous development practices using this strategy. The case study is focused on the compiler design of a language called RelationJava, which provides *relations* as first-class computation elements along with Java classes and interfaces. The nature of the language is irrelevant with the topic, but it can help to understand how the compiler is built. Essentially, a relation is an external method owned by multiple interfaces/classes and it can use fields and methods of those interfaces/classes directly. The language was implemented by compiling it into regular Java and the key part was to translate a relation into a static Java method defined in a default class. The compiler implementation reused an existing JavaCC™ [1]+JJTree[2] sample program that can take a Java program and reproduce its source code by using an unparse visitor [3]. Consequently, the RelationJava compiler can be achieved based on this by providing additional AST nodes and their translation rules for a relation definition.

The implementation of the compiler was a three-step development process: first, in order to facilitate further aspect-oriented development, the `UnparseVisitor` class was rewritten as an `Unparse` aspect. Then, the `unparse` methods of new nodes were introduced in a `Translation` aspect derived from the `Unparse` aspect to translate each relation locally into a static Java method, with related class names converted to method parameters. This translation was achieved without using any membership information of the related classes, so the language user was required to specify the class name explicitly before each referenced class member. To eliminate this limitation such that class members can be referred to in a relation directly, it was needed to retrieve ownership information about related classes with respect to their corresponding fields and methods. This was realized in the third step by loading symbol tables with a `LoadSymbolTable` aspect. The exploration of developing three aspects is detailed in the following paragraphs.

1) Unparse aspect. The unparsed visitor by JJTree is an example of the problems associated with the Visitor pattern. The JJTree implementation contains 500 lines of handwritten code for the visiting methods and 90 automatically generated `accept` methods inside each AST node. The contents of each visiting method are identical; i.e., return the call results of a `print` method, as illustrated in Lines 9-18 of Figure 2. The reason that the Visitor pattern has so much redundancy in this case is that the abstract visitor, which is necessary for double-dispatch, forces each concrete visitor to implement a visiting method for every single node, even if their contents are identical. On the other hand, in the aspect-oriented implementation, since all the semantic operations can be naturally weaved into the class hierarchy, there is no need to define the generic abstract visitor. After rewriting the `UnparseVisitor` class as an `Unparse` aspect, *all of those redundancies (500 lines of code) are removed by a three line ITD in AspectJ* (as shown in Line 8-10 of Figure 3). Correspondingly, the `print` methods, as well as other helper routines inside `UnparseVisitor`, are rewritten as static aspect methods, and the `PrintStream` field is converted to a static aspect field (as shown in Lines 2-7 of Figure 3).

```
1.  class UnparseVisitor {
2.    protected PrintStream out = System.out;
3.    public Object print(SimpleNode node,
4.                        Object data){
5.      // ...
6.    }
7.    // Other utility routines
8.    // ...
9.    public Object visit(SimpleNode node,
10.                       Object data){
11.     return print(node, data);
12.   }
13.   public Object visit(ASTCompilationUnit node,
14.                       Object data){
15.     return print(node, data);
16.   }
17.   // same visit methods for another 83 nodes.
18.   // ...
19. }
```

Figure 2. The UnparseVisitor class

```
1.  aspect Unparse{
2.    protected static PrintStream out = System.out;
3.    public static void print(SimpleNode node){
4.      // ...
5.    }
6.    // other utility routines
7.    // ...
8.    public void SimpleNode.unparse(){
9.      Unparse.print(this);
10.   }
11. }
12. aspect Translation extends Unparse{
13.   public void RelationDefinition.unparse(){
14.     // ...
15.   }
16.   // unparsed ITDs for other new AST nodes
17. }
```

Figure 3. The ITDs and static members in Unparse aspect and the inherited Translation aspect

[1] Java Compiler Compiler - https://javacc.dev.java.net

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

[3] Available at https://javacc.dev.java.net/source/browse/javacc/examples/VTransformer/UnparseVisitor.java.

2) Translation aspect. The `Translation` aspect (as shown in Lines 12-17 of Figure 3), which defines `unparse` methods for the new AST node classes, inherits the `Unparse` aspect to reuse the helper routines (e.g., `Print`) defined in it. The new `unparse` methods override the `unparse` method defined in the `SimpleNode` so that when a `RelationDefinition` or a `RelationReference` node is processed, its translated Java methods or statements are generated instead of simply reprinting its original source code.

3) LoadSymbolTable aspect. The symbol table loading operation is written as another clearly separated aspect. To avoid traversing the whole tree for a second time, the methods defined in the `LoadSymbolTable` aspect are specified as stand-alone methods, which are glued together by poincuts with the `Translation` aspect that has the traversal logic (specified in Figure 4). As in Line 6, each `symTabLoad` method is invoked before the `unparse` method to ensure required symbol information can be obtained during the translation. Therefore, although there are two aspects, the whole AST is only traversed once. Moreover, besides invoking `symTabLoad` methods, the two advice also invoke `pushCurrentSymTab` and `popCurrentSymTab` methods (as in Line 6 and 9, respectively) to reserve and reset the current symbol table scope after traversing each AST node, which is usually specified in an error-prone fashion in YACC[4] actions, or passed back and forth between AST node classes and visitors as an `Object` type.

```
1.  aspect LoadSymbolTable{
2.    pointcut translate(SimpleNode s):
3.      target(s) && call(* *.unparse(..));
4.    before(SimpleNode s): translate(s){
5.      pushCurrentSymTab(); // reserve SymTab scope
6.      s.symTabLoad();
7.    }
8.    after(SimpleNode s):translate(s){
9.      popCurrentSymTab(); // reset SymTab scope
10.   }
11.   static SymbolTable globalSymTab;
12.   static SymbolTable currentSymTab;
13.   void pushCurrentSymTab(){
14.     // ...
15.   }
16.   void popCurrentSymTab(){
17.     // ...
18.   }
19.   // symTabLoad() ITDs
20.   // ...
21. }
```

Figure 4. The LoadSymbolTable aspect

With these three aspects, the language was fully implemented without the requirement to specify the class name before each referenced class member. The compiler can determine which class each field/method comes from in a relation definition and generate the Java code. Notice that because the extension to the compiler is achieved by using aspects, the compiler can always return to its original state, i.e., simply reprint a Java program, by not weaving the new aspects. As the language and compiler evolve, additional semantic functionality may be added as separated aspects without any change in the existing code.

4. ASPECT-ORIENTED COMPILER DEVELOPMENT IN ASPECTJ

As illustrated in previous sections, aspect-oriented programming is an ideal programming technology to solve the separation of concerns problem in compiler development. By using AOP, all the operations that belong to one semantic phase can be encapsulated as a separated aspect, providing much better modularization and abstraction at the source code level. The aspects can be composed and selectively weaved into AST node classes at compile time, without abandoning the desirable properties of object-orientation such as polymorphism and overloading. The following sub-sections summarize the aspect-oriented semantic analysis methodology using AspectJ, categorized by the language features that help model the compiler development process.

4.1 Inter-type declarations

In applying AOP to compiler design, each semantic phase is implemented as an individual aspect. The semantic artifacts that are represented as methods and fields of each AST node class can be removed to the specific semantic aspect as ITDs, which can be freely introduced to the existing AST class hierarchy. As compared to declaring these operations directly inside the Java classes, AspectJ ITDs have the following benefits:

1) Aspect-orientation can isolate crosscutting semantic behavior in an explicit way. As shown in Section 3, each semantic segment is encapsulated as one physically separated aspect; e.g., the `LoadSymbolTable` aspect contains nothing but the code related to symbol loading.

2) Each semantic aspect can be freely attached to generate AST nodes without "polluting" the parser or AST node structure. It is a common practice to generate node classes automatically in a number of tree generation systems such as JJTree and JTB[5], to which user-supplied semantic code can be added afterwards. However, the mixed generated code and handwritten code lead to a system that is hard to maintain and evolve. There are situations when the regenerated code can overwrite the user-supplied code, or the generated code is modified by users accidentally. By using ITDs, all the hand-written code exists as aspects, and hence is clearly separated from the generated code that exists as classes, as we have seen in Section 3.

3) Since each aspect is separated with other aspects, developers can always come back to the previous phase while developing a

[4] Yet Another Compiler-Compiler - http://dinosaur.compilertools.net

[5] JTB: Java Tree Builder. http://www.cs.purdue.edu/jtb/releasenotes.html

later phase. As compiler design is a multi-stage process, it is beneficial to have newly added phases independent of existing phases: if an error occurs, it would be easier to narrow the problem scope using aspects; the failure of the new phase will not affect the success of previous phases.

4) Different aspects can be selectively plugged in for different purposes. For example, the pretty print operation could be either defined to return a `String` that has the desired code format or defined as a `void` method that prints the formatted code directly. One of these two operations defined as aspects can be selectively plugged in by weaving.

4.2 Pointcut-advice model

AspectJ provides language constructs for defining pointcuts and advice around join points, which can significantly impact the compiler development process. The following areas are some sample facets facilitated by the abstraction power of join-points.

1) Phase combination. Using the pointcut-advice model, each semantic aspect can be glued together with other aspects as one phase, which eliminates unnecessary passes of the AST. For example, in the development of the RelationJava compiler, the `LoadSymbolTable` aspect was integrated with the `Translation` aspect. One key requirement of the gluing is there should be no traversal conflicts, which means only one aspect should contain the iteration logic, with others containing standalone semantic actions attached to it.

```
1.  Dfvisit algorithm:
2.  procedure dfvisit(n:node);
3.  begin
4.    for each child m of n, from left to right
5.    do begin
6.      evaluate inherited attributes of m;
7.      dfvisit(m);
8.    end;
9.    evaluate synthesized attributes of n
10. end
11. AspectJ implementation:
12. pointcut synAttrCall(Node node):
13.   target(node)"&& call (* *. synthesized());
14.   before(Node node):synAttrCall (node){
15.     Iterator iter = node.getChildren();
16.     while(iter.hasNext()){
17.     Node child = (Node)iter.next();
18.     child.inherited();
19.     child.synthesized();
20.   }
21. }
```

Figure 5. The Dfvisit algorithm and its AspectJ implementation

2) Tree traversal. Inside one semantic aspect, tree traversal algorithms are easy to implement with pointcuts and advice, especially for the attribute grammar traversal strategy. For example, the depth-first evaluation algorithm `dfvisit` for L-attributed grammars [2] can be directly implemented in AspectJ as shown in Figure 5, provided that the method `synthesized` handles synthesized attribute computations and the method `inherited` handles inherited attribute computations. The main program simply calls the `synthesized` method of the root node to activate whole tree traversal.

3) Tracing. One of the canonical uses of aspects is to facilitate tracing. For example, it is often required to add print statements during application development to debug certain functionality. Using AspectJ, such tracing or debugging statements can be easily introduced without manually adding them in several places in the source code. Such a debugging facility is especially useful in compiler development, as an AST usually contains hundreds of node types. It would be quite tedious to put tracing information in every node class to keep track of the node processing sequence. Using the join point model, the construction and traversal information of the AST can be easily specified and displayed. For example, in order to track if the AST is created as expected, the code shown in Figure 6 prints the AST node creation sequence.

```
1.  aspect PrintNodeCreation{
2.    pointcut construction(Node n): target(n)
3.      && execution((Node+ && !Node).new(..));
4.    after(Node n) returning():construction(n){
5.      System.err.println(
6.       thisJoinPointStaticPart.getSignature().
7.       getDeclaringType().getName()+"is created");
8.    }
9.  }
```

Figure 6. A sample aspect to trace the AST construction process

4.3 Aspect fields and methods

During one pass of the AST traversal, there are certain fields and methods that need to be shared by all AST node operations related to this specific phase. This type of fields includes constants and accumulated states, such as the `PrintStream` object in Figure 2 and the `SymbolTable` objects in Figure 4. The methods in this category include utility routines or a common processing method that can be reused by various nodes such as the `print` method in Figure 2 and `popCurrentSymTab` and `pushCurrentSymTab` in Figure 4. Within an aspect, these elements can be directly declared as its own static fields and methods, which are accessible by all ITD methods. Without aspects, the fields or methods would be passed as extra arguments to the semantic operations or they might appear as global elements in non-visitor implementations.

4.4 Aspect inheritance

Similar to classes in Java, aspects are extendable entities. AOP allows new functionality to be added to an object-oriented system without modifying the existing classes, as we have seen in the `Translation` aspect. Aspect inheritance further allows new

functionality to be added as a separated aspect without modifying the existing aspects. This improves the reusability of aspects. Typically, multiple semantic phases may share several common fields and routines. These constructs can be defined in a parent aspect that is reusable among other aspects. There are also cases when the implementations are separated with interfaces, where an abstract aspect can be implemented by concrete aspects.

In summary, because AOP focuses on modularizing concerns that crosscut multiple classes, aspects can be helpful in separating the compiler stages that require traversals on multiple AST nodes. Although the benefits declared in Section 4.3 and 4.4 can also be obtained by using the Visitor pattern, none of the object-oriented implementation can provide same kind of facilities as using ITDs and pointcuts in aspect-oriented compiler development. Since the semantics of AspectJ is compatible with Java, any compiler written in Java (either using or not using the Visitor pattern) could be reimplemented with AspectJ.

5. RELATED WORK

Applying AOP concepts to compiler design was first proposed by de Moor et al. [8], at a time when AOP tool support was not mature and when various definitions of aspects existed. Essentially, they introduced an aspect-oriented implementation of an attribute grammar, where each aspect represented a semantic attribute (e.g., the environment). Their notion of aspect is highly restrictive and it is a slight deviation from the general notion and terminology of aspects as described in this paper.

Similarly, TreeCC [9] invented special notations to facilitate compiler writers to specify semantic computation outside of an AST node structure. These aspects are translated by a preprocessor into C functions with those AST nodes as parameters, whereas our approach is built on object-oriented AST nodes with each semantic operation implemented as a method of the node itself. Therefore, the advantages of object-orientation are utilized in our framework.

JastAdd II [10] is a language implementation tool that supports the generation of compilers based on Rewritable Reference Attributed Grammars. In JastAdd II, static aspect-oriented programming technology was utilized to assist some of the computation of attributes, which can add features to the AST classes in a way analogous to the use of ITDs in AspectJ as described in this paper. The difference is that our approach regards all semantic actions as concerns crosscutting AST classes and consequently fully implements them using AspectJ's various constructs and features, such as ITDs, join point models, aspect static member and even aspect-inheritance.

Directly using AspectJ's ITDs in compiler implementation was also introduced in [11], without fully investigating utilizing other features of AOP languages. The ITDs are primarily used as a notation that can be transformed back and forth with object-oriented functions to support multiple evolution needs, which has a different focus than this paper.

6. CONCLUSION

This paper emphasizes the difficulties in separating concerns during compiler construction using purely object-oriented approaches and illustrates an alternative technique that uses aspect-orientation to describe semantic phases built on object-oriented AST nodes. The approach clearly separates each semantic operation as an aspect and uses various AOP language features to fulfill the computational needs of tree traversal, thereby improving the overall modularization of the system and providing flexibility for future evolution of the compiler. The approach supersedes the object-oriented Visitor pattern by its unrestricted method definitions and transparent nature to AST node classes, as well as the flexibility in phase integration and tree walking using join points. The benefits have been experimentally validated in a compiler developed in AspectJ that translates RelationJava code into regular Java code. A similar study could be done on any software system that has a tree-like structure and obtain the same benefits of using AOP.

7. REFERENCES

[1] K.-G. Doh and P. D. Mosses, Composing programming languages by combining Action-semantics modules, *Science of Computer Programming,* Vol. 47, No. 1, pp. 3-36, 2003.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[4] O. Hachani and D. Bardou. Using Aspect-Oriented Programming for Design Patterns Implementation. In *Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002, http://www-lsr.imag.fr/OOIS_Reuse_Workshop/Papers/Hachani.pdf.

[5] 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)*, 1997, pp. 220-242.

[6] 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.

[7] M. Mernik, X. Wu, B. R. Bryant. Object-Oriented Language Specification: Current Status and Future Trends, In *Proc. ECOOP Workshop on Evolution and Reuse of Language Specifications for DSLs*, June 2004.

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

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

[10] G. Hedin and E. Magnusson. JastAdd-An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, Vol. 47, No. 1, pp. 37-58, 2003.

[11] X. Wu, S. Roychoudhury, B. Bryant, J. Gray, and M. Mernik. A Two-Dimensional Separation of Concerns for Compiler Construction. In *Proc. ACM Symposium on Applied Computing (SAC),* 2005, pp. 1365-1369.